LOGIC IN COMPUTER SCIENCE

by

Hantao Zhang, Jian Zhang

A Quote

When I used a word – *said Humpty Dumpty* –
it means precisely what I wish to mean
– neither more –
nor less.

May 2022

Editor: Benji Mo

ABSTRACT

This book is written based on the lecture notes used for the course entitled *Logic in Computer Science* at the University of Iowa, Iowa, U.S.A. The book focuses on logic as a practical software tool for solving various problems in computer science and engineering.

LOGIC IN COMPUTER SCIENCE

by

Hantao Zhang, Jian Zhang

Some people are always critical of vague statements.
I tend rather to be critical of precise statements.
They are the only ones which can
correctly be labeled wrong.
– Raymond Smullyan

May 2022

Supervisor: Benji Mo

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Hantao Zhang, Jian Zhang

has been approved by the Examining Committee
for the thesis requirement for the Doctor of
Philosophy degree in Computer Science at the May 2022
graduation.

The committee: _____
Supervisor

_____
Member

_____
Member

_____
Member

_____
Member

ACKNOWLEDGMENTS

The author is grateful for selfless help of Professor Hantao Zhang.

ABSTRACT

This is the first draft of a textbook on Logic in Computer Science.

TABLE OF CONTENTS

Page

LIST OF FIGURES ix

CHAPTER

1 Introduction to Logic 1

1.1 Logic is Everywhere 2
1.1.1 Statement or Proposition 3
1.1.2 A Brief History of Logic 5
1.2 Logical Fallacies in Arguments 5
1.2.1 Formal Fallacies 6
1.2.2 Informal Fallacies 8
1.3 A Brief Review of Mathematical Logic 9
1.3.1 Set Theory 9
1.3.2 Model Theory 18
1.3.3 Proof Theory 20
1.3.4 Computability Theory 23
1.4 Exercise Problems 26

2 Propositional Logic 30

2.1 Syntax 30
2.1.1 Logical Operators 30
2.1.2 Formulas 32
2.2 Semantics 33
2.2.1 Interpretations 34
2.2.2 Models, Satisfiability, and Validity 36
2.2.3 Equivalence 38
2.2.4 Entailment 41
2.2.5 Theorem Proving and the SAT Problem 43
2.3 Normal Forms 44
2.3.1 Negation Normal Form (NNF) 46
2.3.2 Conjunctive Normal Form (CNF) 47
2.3.3 Disjunctive Normal Form (DNF) 49
2.3.4 Full DNF and Full CNF from Truth Table 51
2.3.5 Binary Decision Diagram (BDD) 52
2.4 Optimization Problems 57
2.4.1 Minimum Set of Operators 57

2.4.2 Logic Minimization 59
2.4.3 Maximum Satisfiability 67
2.5 Using Propositional Logic 68
2.5.1 Bitwise Operators 68
2.5.2 Specify Problems in Propositional Logic 70
2.6 Exercise Problems 75

3 Proof Procedures for Propositional Logic 83

3.1 Semantic Tableau 85
3.1.1 Tableau: A Tree Structure for DNF 86
3.1.2 α-Rules and β-Rules 87
3.2 Deductive Systems 90
3.2.1 Inference Rules and Proofs 90
3.2.2 Hilbert Systems 92
3.2.3 Natural Deduction 94
3.2.4 Inference Graphs 97
3.3 Resolution 98
3.3.1 Resolution Rule 98
3.3.2 Resolution Strategies 101
3.3.3 Preserving Satisfiability 103
3.3.4 Completeness of Resolution 106
3.3.5 A Resolution-based Decision Procedure 109
3.3.6 Clause Deletion Strategies 110
3.4 Boolean Constraint Propagation (BCP) 113
3.4.1 BCP: a Simplification Procedure 114
3.4.2 BCP: a Decision Procedure for Horn Clauses 115
3.4.3 Unit Resolution versus Input Resolution 116
3.4.4 Head/Tail Literals for *BCP* 118
3.5 Exercise Problems 123

4 Propositional Satisfiability 126

4.1 The DPLL Algorithm 127
4.1.1 Recursive Version of *DPLL* 127
4.1.2 All-SAT and Incremental SAT Solvers 130
4.1.3 *BCPw*: Implementation of Watch Literals 131
4.1.4 Iterative Implementation of *DPLL* 134
4.2 Conflict-Driven Clause Learning (CDCL) 136
4.2.1 Generating Clauses from Conflicting Clauses 137
4.2.2 DPLL with CDCL 138
4.2.3 Unsatisfiable Cores 141
4.2.4 Random Restart 142
4.2.5 Branching Heuristics for DPLL 143
4.3 Use of SAT Solvers 145

4.3.1 Specify SAT Instances in DIMACS Format 146
4.3.2 Sudoku Puzzle 147
4.3.3 Latin Square Problems 149
4.3.4 Graph Problems 150
4.4 Local Search Methods and MaxSAT 152
4.4.1 Local Search Methods for SAT 152
4.4.2 2SAT versus Max2SAT 155
4.5 Maximum Satisfiability 157
4.5.1 Weight MaxSAT and Hybrid MaxSAT 157
4.5.2 The Branch-and-Bound Algorithm 159
4.5.3 Simplification Rules and Lower Bound 161
4.5.4 Use of Hybrid MaxSAT Solvers 164
4.6 Exercise Problems 166

5 First Order Logic 170

5.1 Syntax of First Order Languages 170
5.1.1 Terms and Formulas 170
5.1.2 The Quantifiers 174
5.1.3 Unsorted and Many-Sorted Logics 176
5.2 Semantics 178
5.2.1 Interpretation 178
5.2.2 Models, Satisfiability, and Validity 182
5.2.3 Entailment and Equivalence 184
5.3 Proof Methods 187
5.3.1 Semantic Tableau 188
5.3.2 Natural Deduction 190
5.4 Conjunctive Normal Form 191
5.4.1 Prenex Normal Form 191
5.4.2 Skolemization 193
5.4.3 Skolemizing Non-Prenex Formulas 196
5.4.4 Clausal Form 197
5.4.5 Herbrand Models for CNF 199
5.5 Exercise Problems 201

6 Unification and Resolution 205

6.1 Unification 205
6.1.1 Substitutions and Unifiers 205
6.1.2 Combining Substitutions 207
6.1.3 Rule-Based Unification 208
6.1.4 Practically Linear Time Unification Algorithm 211
6.2 Resolution 218
6.2.1 The Resolution and Factoring Rules 218
6.2.2 A Refutational Proof Procedure 220

6.3 Ordered Resolution . . . 222
6.3.1 Simplification Orders . . . 222
6.3.2 Completeness of Ordered Resolution . . . 227
6.4 Prover9: A Resolution Theorem Prover . . . 230
6.4.1 Input Formulas to Prover9 . . . 230
6.4.2 Inference Rules and Options . . . 233
6.4.3 Simplification Orders in Prover9 . . . 236
6.5 Exercise Problems . . . 237

7 Equational Logic . . . 241

7.1 Equality of Terms . . . 241
7.1.1 Axioms of Equality . . . 242
7.1.2 Semantics of "=" . . . 243
7.1.3 Theory of Equations . . . 245
7.2 Rewrite Systems . . . 247
7.2.1 Rewrite Rules . . . 247
7.2.2 Termination of Rewrite Systems . . . 249
7.2.3 Confluence of Rewriting . . . 250
7.2.4 The Knuth-Bendix Completion Procedure . . . 251
7.2.5 Rewrite Systems for Ground Equations . . . 257
7.3 Inductive Theorem Proving . . . 258
7.3.1 Inductive Theorems . . . 258
7.3.2 Structural Induction . . . 259
7.3.3 Induction on Two Variables . . . 261
7.3.4 Multi-Sort Algebraic Specifications . . . 262
7.4 Resolution with Equality . . . 265
7.4.1 Paramodulation . . . 265
7.4.2 Simplification Rules . . . 267
7.4.3 Equality in Prover9 . . . 269
7.5 Mace4: Finite Model Finding in FOL . . . 271
7.5.1 Use of Mace4 . . . 271
7.5.2 Finite Model Finding by SAT Solvers . . . 275
7.6 Exercise Problems . . . 277

8 Prolog: Programming in Logic . . . 280
8.1 Prolog's Working Principle . . . 280
8.1.1 Horn Clauses in Prolog . . . 280
8.1.2 Resolution Proofs in Prolog . . . 282
8.1.3 A Goal-Reduction Procedure . . . 284
8.2 Prolog's Data Types . . . 289
8.2.1 Atoms, Numbers, and Variables . . . 289
8.2.2 Compound Terms and Lists . . . 291
8.2.3 Popular Predicates over Lists . . . 292
8.2.4 Sorting Algorithms in Prolog . . . 295

8.3 Recursion in Prolog . . . 296
8.3.1 Program Termination . . . 297
8.3.2 Focused Recursion . . . 299
8.3.3 Tail Recursions . . . 300
8.3.4 A Prolog program for N-Queen Puzzle . . . 301
8.4 Beyond Clauses and Logic . . . 302
8.4.1 The Cut Operator ! . . . 303
8.4.2 Negation as Failure . . . 304
8.4.3 Beyond Clauses . . . 306
8.5 Exercise Problems . . . 307

9 Undecidable Problems in First-Order Logic . . . 310

9.1 Turing Machines . . . 311
9.1.1 Formal Definition of Turing Machines . . . 312
9.1.2 High-level Description of Turing Machines . . . 315
9.1.3 Recognizable vs Decidable . . . 316
9.2 Decidability of Problems . . . 316
9.2.1 Encoding of Decision Problems . . . 317
9.2.2 Decidable Problems . . . 318
9.2.3 Undecidable Problems . . . 320
9.3 Turing Completeness . . . 323
9.3.1 Turing Completeness of Prolog . . . 324
9.3.2 Turing Completeness of Rewrite Systems . . . 326
9.4 Exercise Problems . . . 327

10 Hoare Logic . . . 328

10.1 Hoare Triples . . . 329
10.1.1 Hoare Rules . . . 331
10.1.2 Examples of Formal Verification . . . 335
10.1.3 Hoare Rule for Function Calls . . . 339
10.1.4 Partial and Total Correctness . . . 339
10.2 Automated Generation of Assertions . . . 340
10.2.1 Verification Conditions . . . 341
10.2.2 Proof of Theorem 10.2.4 . . . 343
10.2.3 Implementing *vc* in Prolog . . . 347
10.3 Obtaining Good Loop Invariants . . . 351
10.3.1 Invariants from Generalizing Postconditons . . . 352
10.3.2 Program Synthesis from Invariants . . . 355
10.3.3 Choosing A Good Language for Assertions . . . 358
10.4 Exercise Problems . . . 358

11 Temporal Logic . . . 359

11.1 An Approach from Modal Logic 360
11.1.1 Modal Operators 360
11.1.2 Kripke Semantics 361
11.1.3 Restrictions and Limitations 365
11.2 Linear Temporal Logic 367
11.2.1 Timeline as Interpretation Sequence 368
11.2.2 Properties of LTL 371
11.3 Semantic Tableaux for LTL 373
11.3.1 Rules for Modal Operators 374
11.3.2 Deciding Satisfiability by Tableaux 377
11.4 Binary Temporal Operators 385
11.4.1 The *until* and *release* Operators 386
11.4.2 The *weak until* and *strong release* Operators 388
11.4.3 Extension of Semantic Tableau 389
11.5 Verification of Concurrent Programs 390
11.5.1 The BAKERY(2) Program 390
11.5.2 States and Transition Conditions 391
11.5.3 Transition Conditions in LTL 392
11.5.4 Hoare Triples and Invariants 394
11.5.5 Verification of Mutual Exclusion 396
11.5.6 Verification of Accessibility 397
11.6 Exercise Problems 398

12 Decision Procedures 401

12.1 Equality with Uninterpreted Functions (EUF) 401
12.1.1 Uninterpreted Functions 401
12.1.2 The Congruence Closure Problem 403
12.1.3 Nelson and Oppen's Algorithm 404
12.1.4 Ground Congruence by Knuth-Bendix Completion . . . 408
12.2 Linear Arithmetic 414
12.2.1 Simplex Method by Example 414
12.2.2 The Simplex Algorithm 418
12.2.3 Linear Programming 420
12.2.4 Integer Programming 421
12.2.5 Difference Arithmetic 423
12.3 Finite-length Linear Data Types 423
12.3.1 Bit Vectors 423
12.3.2 Arrays 423
12.3.3 Points 423
12.4 Satisfiability Modulo Theories (SMT) 423
12.4.1 DPLL(T): Extension of DPLL with Theory T 424
12.4.2 Combination of Theories 427
12.5 Exercise Problems 428

LIST OF FIGURES

Figure Page

2.1.1 The formula tree for $(\neg p \wedge q) \rightarrow (p \wedge (q \vee \neg r))$ 33
2.3.1 The BDD of INF $ite(x_1, ite(x_2, 1, 0), ite(x_2, ite(x_3, 1, 0), ite(x_3, 0, 1)))$ derived from $\overline{x_1 x_2 x_3} \vee x_1 x_2 \vee x_2 x_3$. 55
2.3.2 The first BDD uses $a_1 > b_1 > a_2 > b_2 > a_3 > b_3$ and the second BDD uses $a_1 > a_2 > a_3 > b_1 > b_2 > b_3$ for the same formula $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$. 55
2.4.1 K-map for $f(x, y) = x\overline{y} + \overline{x}y + \overline{xy}$. 61
2.4.2 The K-map for obtaining $f(A, B, C) = AB + \overline{A}C$. 62
2.4.3 The K-map for $f(P, Q, R, S) = m_0 + m_2 + m_5 + m_7 + m_8 + m_{10} + m_{13} + m_{15}$. The simplified formula is $f = \overline{QS} + QS$. 63
2.4.4 The K-map for $f(A, B, C) = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$. The simplified CNF is $f = (A + B + C)(\overline{A} + \overline{B})(\overline{B} + \overline{C})$. 64
3.1.1 The tableaux for $A = p \wedge (\neg q \vee \neg p)$ and $B = (p \vee q) \wedge (\neg p \wedge \neg q)$ . . 87
3.2.1 The proof tree of $\neg q$ 92
4.1.1 The result of BCP is shown in the box near each node. 128
4.3.1 A typical example of Sudoku puzzle and its solution. 147
4.4.1 The implication graph for 2CNF C and the SCC. 156
4.5.1 Finding a maximal clique in a graph. 159
6.1.1 The term graphs for t and s 213
8.1.1 $\vee$-nodes in the $\wedge$-$\vee$ tree for the query `dfs(a, N)`. 286
11.1.1 A graphical display of the Kripke frame 362
11.5.1 Program BAKERY(2) 391
11.5.2 Program BAKERY(N) 398
11.6.1 Program BAKERY(N) 400
12.1.1 DAG $G = (V, E)$ for $A = \{f(a, b) = a, f(f(a, b), b) \neq a\}$, where $V = \{1, 2, 3, 4\}$, $use(1) = \emptyset, use(2) = \{1\}$, $use(3) = \{2\}$, $use(4) = \{1, 2\}$ 406

CHAPTER 1
INTRODUCTION TO LOGIC

Do you like to solve Sudoku puzzles? If you do, how long will it take on average to solve a Sudoku puzzle appearing in a newspaper, minutes or hours? An easy exercise in Chapter 4 of this book asks you to write a small program in your favorite language that will solve a Sudoku puzzle, hard or easy, in less than a second on your laptop.

Did you ever play the *tower of Hanoi*? This puzzle consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No larger disk may be placed on top of a smaller disk.

For instance, if the three rods are named by a, b, and c, and there are three disks, each named by its size, a solution of the puzzle of moving the disks from rod a to rod b is given as follows:

```
Move disk 1 from a to b
Move disk 2 from a to c
Move disk 1 from b to c
Move disk 3 from a to b
Move disk 1 from c to a
Move disk 2 from c to b
Move disk 1 from a to b
```

This solution was produced by a Prolog program of 3 lines and 168 characters (including commas and periods). The same program can produce solutions for various numbers of disks. Prolog is a programming language based on logic and is introduced in Chapter 8 of the book.

The above two examples just illustrate that logic is not only a theoretic foundation of computer science, but also a problem-solving tool. The emphasis of the book is on how to make this tool efficient and practical. After a rigorous introduction to basic concepts, we will study various algorithms and introduce software tools which are the implementation of these algorithms.

To solve a problem using a logic-based tool, you need to think the problem in logic and specify the problem in logic. This is a skill that needs to be learned like any other, and it does take a bit of training and practice to master this skill. The same skill can be applied to abstract situations such as those encountered in formal proofs. Logic is also innate to all of you - indeed, you probably use the laws of logic unconsciously in your everyday speech and in your own internal reasoning. Because logic is innate, the logic principles that you learn should make sense - if you find yourself having to memorize one of the principles in the book, without feeling a mental "click" or comprehending why that law should work, then you will probably not be able to use that principle correctly and effectively in practice.

1.1 Logic is Everywhere

Logic has been called the *calculus of computer science*, because logic plays a fundamental role in computer science, similar to that played by calculus in the physical sciences and traditional engineering disciplines. Logic is used in almost every field of computer science: the construction of computers, such as computer architecture (digital gates, hardware verification), software engineering (specification, verification), programming languages (semantics, type theory, abstract data types, object-oriented programming), databases (relational algebra), artificial intelligence (automated theorem proving, knowledge representation), algorithms and theory of computation (complexity, computability), etc.

Logic also plays important roles in other fields, such as mathematics and philosophy. In mathematics, logic includes both the mathematical study of logic and the applications of formal logic to other areas of mathematics. The unifying themes in mathematical logic include the study of the expressive power of formal systems and the deductive power of formal proof systems. Mathematical logic is often divided into the fields of (a) set theory, (b) model theory, (c) proof theory, and (d) computability theory. These areas share basic results on logic.

Symbolic logic in the late nineteenth century and mathematical logic in the twentieth are also called *formal logic*. Topics traditionally treated by logic not being part of formal logic have tended to be termed either *philosophy of logic* or *philosophical logic* . Philosophical logic dealt with arguments in the natural language

used by humans.

> **Example**. Given the premises that (a) "all men are mortal" and (b) "Socrates is a man", we may draw the conclusion that (c) "Socrates is mortal", by the inference rule of *Modus Ponens*.

Philosophical logic is the investigation, critical analysis and intellectual reflection on issues arising in logic and is the branch of studying questions about reference, predication, identity, truth, quantification, existence, entailment, modality, and necessity.

1.1.1 Statement or Proposition

Logic comes from natural languages as most sentences in natural languages are statements or propositions. A *proposition* is a statement or assertion that expresses a judgment or opinion. We will use *statement* and *assertion* as synonyms of proposition. Here are some examples:

- The color of the apple is green.
- Today is either Monday or Tuesday.
- He is a sophomore and twenty years old.

Every statement can be either true or false. True or false are called the *truth values* of a statement. Some sentences in natural languages are not statements, such as commands or exclamatory sentences. For example, *"Run fast!" Coach shouts.* The first part of the sentence, "Run fast!", is **not** a statement; the second part, "Coach shouts", is a statement. In fact, a sentence is a statement if and only if it has a truth value.

In natural languages, we can combine, or relate statements with words such as "not" (negation), "and" (conjunction), "or" (disjunction), "if-then" (implication), etc. That is, a statement can be obtained from other statements by these words. In logic, these words are called *logical operators*, or equivalently, *logical connectives*. A statement is *composed* if it can be expressed as a composition of several simpler statements; otherwise, it is *simple*. In the above three examples of statements, the first statement is simple; the other two are composed. That is, "today is either Monday or Tuesday" is the composition of "today is Monday" and "today is Tuesday", using the logical operator "or". We often regard "the color of that apple is not green" as a composed statement: It is the negation of a simple statement.

Logical connectives are indispensable for expressing the relationship between statements. For example, the following statement is a composition of several simple statements.

> (∗) Either taxes are not increased or if expenditures rise then the debt ceiling is raised.

To see this clearly, we need to introduce symbols, such as t, e, and d, to denote them:

1. t: "taxes are increased",
2. e: "expenditures rise",
3. d: "the debt ceiling is raised".

To express that the statement (∗) is composed, let us use these symbols for logical operators: $\neg$ for negation, $\vee$ for "either-or", and $\rightarrow$ for the "if-then" relation (i.e., implication). Then (∗) becomes the following formula:

$$(\neg t) \vee (e \rightarrow d)$$

Here each of the symbols t, e and d is called *propositional variable*, which denotes a statement, either simple or composed. Naturally, these propositional variables, like statements, can take on only the truth values, i.e., true and false.

Propositional variables are also called *Boolean variables* after their inventor, the nineteenth century mathematician George Boole. Boolean logic includes any logic in which the considered truth values are *true* and *false*. The study of propositional variables with logic operators is called *propositional logic*, which is the simplest one in the family of Boolean logics. On the other hand, probability logic is not a Boolean logic because probability values are used to represent various degrees of the truth values. Statements which contain subjective adjectives, such as "that apple is delicious", are better expressed in propositional logic, as "delicious" is subjective. Questions regarding degrees of truth of subjective statements are ignored in Boolean logic. For example, "that apple is delicious" can be true in one's view and false in the other. Despite this simplification, or indeed because of it, such a method is scientifically successful. One does not even have to know exactly what the truth values true and false actually are.

1.1.2 A Brief History of Logic

Sophisticated theories of logic were developed in many cultures, including China, India, Greece and the Islamic world. Greek methods, particularly Aristotelian logic (or term logic), found wide application and acceptance in Western science and mathematics for millennia.

Aristotle (384 – 322 BC) was the first logician to attempt a systematic analysis of logical syntax, of noun (or term), and of verb. He demonstrated the principles of reasoning by employing variables to show the underlying logical form of an argument. He sought relations of dependence which characterize necessary inference, and distinguished the validity of these relations, from the truth of the premises. He was the first to deal with the principles of contradiction and excluded middle in a systematic way. Aristotle has had an enormous influence in Western thought. and he developed the theory of the *syllogism*, where three important principles are applied for the first time in history: the use of symbols, a purely formal treatment, and the use of an axiomatic system. Aristotle also developed the theory of fallacies, as a theory of non-formal logic.

Christian and Islamic philosophers such as Boethius (died 524), Ibn Sina (Avicenna, died 1037) and William of Ockham (died 1347) further developed Aristotle's logic in the Middle Ages, reaching a high point in the mid-fourteenth century, with Jean Buridan (1301-1358/62 AD).

In the 18th century, attempts to treat the operations of formal logic in a symbolic or algebraic way had been made by philosophical mathematicians, including Leibniz and Lambert, but their works remained isolated and little known.

In the beginning of the 19th century, logic was studied with rhetoric, through the syllogism, and with philosophy. Mathematical logic emerged in the mid-19th century as a field of mathematics independent of the traditional study of logic. The development of the modern "symbolic" or "mathematical" logic during this period by the likes of Boole, Frege, Russell, and Peano is the most significant in the two-thousand-year history of logic, and is arguably one of the most important and remarkable events in human intellectual history.

1.2 Logical Fallacies in Arguments

Logical fallacies are logical errors one makes when writing or speaking. In this section, we will give a brief introduction to logical fallacies, which are the subjects of philosophical logic. Knowing these fallacies will help us to write logically by avoiding them. Writing logically is somewhat related to, but not the same as, writing clearly, or efficiently, or convincingly, or informatively; ideally one would

want to do all of these at once, but one does not have the skill has to achieve all, though with practice you'll be able to achieve more of your writing objectives concurrently. The big advantage of writing logically is that one can be absolutely sure that your conclusion will be correct, as long as all your hypotheses were correct and your steps were logical; using other styles of writing one can be reasonably convinced that something is true, but there is a difference between being convinced and being sure.

In philosophical logic, an *argument* has similar meaning as a proof in formal logic, as the argument consists of premises and conclusions. A *fallacy* is the use of invalid or otherwise faulty reasoning in the construction of an argument. A fallacious argument may be deceptive by appearing to be better than it really is. Some fallacies are committed intentionally to manipulate or persuade by deception, while others are committed unintentionally due to carelessness or ignorance.

Fallacies are commonly divided into "formal" and "informal". A formal fallacy can be expressed neatly in a formal logic, such as propositional logic, while an informal fallacy cannot be expressed in a formal logic.

1.2.1 Formal Fallacies

In philosophical logic, a formal fallacy is also called *deductive fallacy*, *logical fallacy*, or *non sequitur* (Latin for "it does not follow"). This is a pattern of reasoning rendered invalid by a flaw in its logical structure.

> **Example**. Give the premises that (a) my car is some car, and (b) some cars are red, we draw the conclusion that (c) my car is red.

This is a typical example of a conclusion that does not follow logically from premises or that is based on irrelevant data. Here are some common logical fallacies:

- **Affirming the consequent**: Any argument with the invalid structure of: If A then B. B, therefore A.

  > **Example**. If I get a B on the test, then I will get the degree. I got the degree, so it follows that I must have received a B. In fact, I got an A.

- **Denying the antecedent**: Any argument with the invalid structure of: If A then B. Not A, therefore not B.

  > **Example**. If it's a dog, then it's a mammal. It's not a dog, so it must not be a mammal. In fact, it's a cat.

- **Affirming a disjunct**: Any argument with the invalid structure of: If A or B. A, therefore, not B.

  > **Example**. I am working or I am at home. I am working, so I must not be at home. In fact, I am working at home.

- **Denying a conjunct**: Any argument with the invalid structure of: It is not the case that both A and B. Not A, therefore B.

  > **Example**. I cannot be both at work and at home. I am not at work, so I must be at home. In fact, I am at a park.

- **Undistributed middle**: Any argument with the invalid structure of: Every A has B. C has B, so C is A.

  > **Example**. Every bird has a beak. That creature has a beak, so that creature must be a bird. In fact, the creature is a dinosaur.

A formal fallacy occurs when the structure of the argument is incorrect, despite of the truth of the premises. A valid argument always has a correct formal structure and if the premises are true, the conclusion must be true. A valid argument is a formally correct argument that use true premises. When we use false premises, the formal fallacies disappear, but the argument may be regarded as a fallacy as the conclusion is invalid.

As an application of modus ponens, the following example contains no formal fallacies:

> If you took that course on CD player repair right out of high school, you would be doing well and having a vacation on the moon right now.

Even though there is no logic error in the argument, the conclusion is invalid because the premise is contrary to the fact. With a false premise, you can make any conclusion, so that the composed statement is always true. However, an always true statement has no value in reasoning.

By contrast, an argument with a formal fallacy could still contain all true premises:

> (a) If someone owns the world's largest diamond, then he is rich. (b) King Solomon was rich. Therefore, (c) King Solomn owned the world's largest diamond.

Although, (a) and (b) are true statements, (c) does not follow from (a) and (b) because the argument commits the formal fallacy of "affirming the consequent".

1.2.2 Informal Fallacies

There are numerous kinds of informal fallacies that use an incorrect relation between premises and conclusion. These fallacies can be grouped approximately into four groups, and each group contains several types of fallacies:

- fallacies of improper premise;
- fallacies of faulty generalizations;
- fallacies of questionable cause; and
- relevance fallacies.

For instance, the *circular reasoning*, where the reasoner begins with what he or she is trying to end up with, belongs to the group of improper premises. An example of circular reasoning is given below.

You must obey the law, because it's illegal to break the law.

For a complete list of informal fallacies, the reader is recommended to read the *Wikipedia*'s page on the same topic for details. Some informal fallacies do not belong to the above four groups. For example, the *false dilemma* fallacy and the *middle ground fallacy*.

The false dilemma fallacy presents a choice between two mutually exclusive options, implying that there are no other options. One option is clearly worse than the other, making the choice seem obvious. Also known as the *either/or* fallacy, false dilemmas are a type of informal logical fallacy in which a faulty argument is used to persuade an audience to agree. False dilemmas are everywhere.

- Vote for me or live through four more years of higher taxes.
- America: Love it or leave it.
- Either we let every immigrant into our country, or we close the borders for everyone.
- Subscribe to our streaming service or be stuck with cable.

The middle ground fallacy assumes that a compromise between two extreme conflicting points is always true. Arguments of this style ignore the possibility that one or both of the extremes could be completely true or false – rendering any form of compromise between the two invalids as well.

Example. Mary thinks the best way to improve sales is to redesign the entire company website, but John is firmly against making any changes to the website. Therefore, the best approach is to redesign some portions of the website.

An argument could contain both an informal fallacy and a formal fallacy yet lead to a conclusion that happens to be true. For example, again affirming the consequent, now also from an untrue premise:

If a scientist makes a statement about science, it is correct. It is true that quantum mechanics is deterministic. Therefore, a scientist has made a statement about it.

Having an understanding of these basic logical fallacies can help us more confidently parse the arguments we participate in and witness on a daily basis, separating fact from sharply dressed fiction.

1.3 A Brief Review of Mathematical Logic

Mathematical Logic is roughly divided into four areas:

- set theory,
- model theory,
- proof theory, and
- computability theory.

Each area has a distinct focus, although many techniques and results are shared between multiple areas. The borderlines between these areas, and the lines between mathematical logic and other fields of mathematics, are not always sharp. Each area contains rich materials which can be studied in graduate courses of multiple levels. As an introductory book on logic, we will introduce some basic concepts, which are closely related to this book, in these areas.

1.3.1 Set Theory

A *set* is a structure, representing an unordered collection of zero or more distinct objects. All elements in a set are unequal (distinct) and unordered. Set theory is a branch of mathematical logic that studies sets and deals with operations between, relations among, and statements about sets. Although any type of object

can be collected into a set, set theory is applied most often to objects that are relevant to mathematics. The language of set theory can be used to define nearly all mathematical objects.

1.3.1.1 Basic Concepts and Notations

Let U be the collection of all objects under consideration. U is often called the *universal set* or *universe of discourse*. For example, U can be the set of natural numbers or the population of the world. Any object x under consideration is a *member* of U, written as $x \in U$. A *set* is any collection of objects from U. The *empty set* is often denoted by $\emptyset$ or simply $\{\}$. Given two sets A and B, the following are the popular set operations:

- $x \in A$: it is true iff (e.g., if and only if) x a member of A.
- $A \cup B$: the *union* of A and B. For any object x, $x \in A \cup B$ iff x is in A or B or both.
- $A \cap B$: the *intersection* of A and B. $x \in A \cap B$ iff $x \in A$ and $x \in B$.
- $A - B$: the *set difference* of A and B. $x \in A - B$ iff $x \in A$ but not $x \in B$.
- $\overline{A}$: the *complement* of A (with respect to the universal set U). $\overline{A} = S - A$. $A - B$ is also called the *complement* of B with respect to A: $A - B = A \cap \overline{B}$.
- $A \subseteq B$: A is a *subset* of B. $A \subseteq B$ is true iff every member of A is a member of B. Naturally, $A \subseteq U$ and $B \subseteq U$ for the universal set U.
- $A \subset B$: A is a *proper subset* of B. $A \subset B$ is true iff $A \subseteq B$ and $A \neq B$.
- $A = B$: $A = B$ iff $A \subseteq B$ and $B \subseteq A$, that is, A and B contain exactly the same elements.
- $A \times B$: the *Cartesian product* of A and B. An ordered pair (a, b) is in $A \times B$ iff $a \in A$ and $b \in B$.
- A^i: the *Cartesian product* of i copies of A, where i is a natural number (i.e., non-negative integers). When $i = 0$, A^0 denotes the *empty sequence*; $A^1 = A$; $A^2 = A \times A$, $A^3 = A \times A \times A$, etc.
- $|A|$: the *cardinality* of a set. When A is finite, $|A|$ is the number of members in A. $|A|$ is called a *cardinal number* if A is infinite.

- $\mathcal{P}(A)$: the *power set* of A. $\mathcal{P}(A)$ is the set whose members are all of the possible subsets of A. That is, for any set X, $X \in \mathcal{P}(A)$ iff $X \subseteq A$. When $|A| = n$, $|\mathcal{P}(A)| = 2^n$.

Elementary set theory can be studied informally and intuitively, and so can be taught in primary schools using Venn diagrams. Many mathematical concepts can be defined precisely using only set theoretic concepts, such as relations and functions.

1.3.1.2 Relations and Functions

Given two sets A and B, a relation between A and B is a subset of $A \times B$. If $A = B$, that happens in many applications, we say the relation is over set A. For example, the less-than relation, $<$, on N, is

$$< \quad = \{\langle 0,1\rangle, \langle 0,2\rangle, \langle 0,3\rangle, \langle 1,2\rangle, \langle 0,4\rangle, \langle 1,3\rangle, ...\}$$

which is a subset of $N \times N$. In general, give a relation R over set S, i.e., $R \subseteq S \times S$, we assume that $R(a,b)$ is true if $\langle a,b\rangle \in R$ and $R(a,b)$ is false if $\langle a,b\rangle \notin R$. Of course, we are used to write $<$ as an infix operator, such as $0 < 1$, instead of a prefix operator such as $< (0,1)$.

Let $\succeq$ be a relation over the set S. For any $a, b, c \in S$.

- $\succeq$ is *reflexive* if $a \succeq a$.
- $\succeq$ is *antisymmetry* if $a \succeq b$ and $b \succeq a$ imply $a = b$.
- $\succeq$ is *transitive* if $a \succeq b$ and $b \succeq c$ imply $a \succeq c$.
- $\succeq$ is *connex* (i.e., *comparable*) if either $a \succeq b$ or $b \succeq a$ is true. That is, if two elements of S are not comparable by $\succeq$, then $\succeq$ is not connex. If $\succeq$ is connex, then $a \succeq a$ must be true, that is, a connex $\succeq$ is always reflexive.
- $\succeq$ is a *total order* if $\succeq$ is antisymmetry, transitive, and connex.
- $\succeq$ is a *partial order* if $\succeq$ is antisymmetry, transitive, and reflexive. Since the connexity implies the reflexivity, a total order is always a partial order but the inverse is not always true (some pair of elements may not comparable in a partial order).
- The strict part of a partial order $\succeq$ is $\succ$: $a \succ b$ iff $a \succeq b$ and $a \neq b$. Obviously, $\succeq \quad = \quad \succ \cup \{\langle x,x\rangle \mid x \in S\}$. For convenience, people also call $\succ$ or $\preceq$ (the reverse of $\succeq$) a partial order.

- Given a partial order $\succeq$, a *minimal* element of $X \subseteq S$ is an element $m \in X$ such that if $x \in X$ and $m \succeq x$, then $x = m$.

- A partial order $\succeq$ is *well-founded* if there is a minimal element $m \in X$ for every nonempty $X \subseteq S$. Equivalently, $\succeq$ is well-founded iff there exists no infinite sequence of distinct elements $x_1, x_2, ..., x_i, ...$, such that

$$x_1 \succeq x_2 \succeq \cdots \succeq x_i \succeq \cdots$$

  For example, the ordinary relation $\geq$ on integers is well-founded on the set of natural numbers but not well-founded on the set of integers.

A *function* f is a relation R between A and B with the property that for every $a \in A$, there exists a *unique* $b \in B$, such that $\langle a, b\rangle \in R$.

- The function f is denoted by $f : A \longrightarrow B$, where A is the *domain* of f and B is the *range* of f. The domain refers to the set of possible input values for f. The range is the set of possible output values of f.

- If $f(a) = b$, we say b is the *output* of f on the *input* a. We also say b is the *image* of a under f. Let $f(A) = \{f(a) \mid a \in A\}$, the set of images of all elements in A under f. Obviously, $f(A) \subseteq B$.

- If a function g takes two values as input, say $g(a, a') = b$, we denote g by $g : A \times A' \longrightarrow B$, and the domain of g is $A \times A'$. By convention, we write $g(a, a') = b$ instead of $g(\langle a, a'\rangle) = b$, where $a \in A, a' \in A'$, and $b \in B$. This practice can be generalized to functions with more than two inputs.

- When both A and B are finite, the set F of all functions from A to B, $F = \{f \mid f : A \longrightarrow B\}$, is also finite. Let $|A| = m$ and $|B| = n$, then $|F| = n^m$, because there are m choices for $a \in A$ and there are n choices for each $f(a)$. For example, if $B = \{0, 1\}$ and $A = B \times B$, then $|B| = 2$, $|A| = 4$ and the number of functions from A to B is $2^4 = 16$.

- A function $f : A \longrightarrow B$ is said to be *total* if for every $x \in A$, there exists an element $y \in B$ such that $f(x) = y$. For instance, the division function over the integers is not total as the quotient is not defined when the divisor is zero.

- $f : A \longrightarrow B$ is said to be *surjective* if for every $y \in B$, there exists $x \in A$ such that $f(x) = y$. It is easy to check that f is surjective iff $B = f(A)$.

- $f : A \longrightarrow B$ is said to be *injective* if f is total and for any $x, y \in A$, $x \neq y$ implies $f(x) \neq f(y)$.

- f is said to be *bijective* if f is both injective and surjective. A bijective function is also called a *one-to-one correspondence* between A and B. A function f is bijective iff the inverse of f, denoted by f^{-1}, exists, where $f^{-1}(y) = x$ iff $f(x) = y$.

1.3.1.3 Countable Sets

People often use N to denote the set of natural numbers, i.e.,

$$N = \{0, 1, 2, 3, ...\},$$

Z to denote the set of integers,

$$Z = \{..., -3, -2, -1, 0, 1, 2, 3, ...\},$$

and R to denote the set of rational numbers. We also use Z^+ to denote the set of all positive integers. These are naturally infinite sets.

For finite sets, the cardinality is a number in N and we can compare them. How to compare the cardinality of infinite sets? Cantor's idea is to use injective functions: If $f : A \longrightarrow B$ is injective, then $|A| \leq |B|$. That is, for every $x \in A$, we have a distinct $f(x) \in B$, so it cannot be the case that $|A| > |B|$.

Definition 1.3.1. *A set S is* countable *if there exists an injective function $f : S \longrightarrow N$, where N is the set of natural numbers. If S is both countable and infinite, S is called* countably infinite*.*

Example 1.3.2. Let $E = \{0, 2, 4, ...\}$ be the set of all even natural numbers. Let $f : E \longrightarrow N$ with $f(x) = x$, then f is injective, so E is countable and we may conclude that $|E| \leq |N$. Since E is infinite, E is also countably infinite. If we define $h : N \longrightarrow E$ with $h(x) = 2x$, then h is bijective, hence $|N| \leq |E|$ and we end up with $|E| = |N|$. This is not an exception for all countably infinite sets. □

Proposition 1.3.3. *There is a bijective function between any two countably infinite sets.*

Proof. It suffices to show that the proposition holds for the two sets A and N (the set of natural numbers). Since A is countable, there exists an injective function $f : A \longrightarrow N$. Listing the elements of A as $a_0, a_1, a_2, ...$, such that $i < j$ iff $f(a_i) < f(a_j)$. Since A is infinite, define $g : N \longrightarrow A$ as $g(i) = a_i$, then g is a bijection between N and A. □

Today, countable sets form the foundation of *discrete mathematics*, and we first check what sets are countable.

Proposition 1.3.4. *The following sets are countable:*

1. *Any finite set;*
2. *Any subset of N;*
3. *The set $A \times B$, if both A and B are countable;*
4. *The set Z of integers;*
5. *The set of all binary strings of finite length;*
6. *Σ^*, the set of all strings of finite length built on the symbols from Σ, a finite alphabet.*

Proof. To show (1) is countable, we just need to define an injective function which map a set of n elements to the first n natural numbers.

(2) is countable because if $S \subseteq N$, then $f : S \longrightarrow N$, where $f(x) = x$, is an injective function.

(3) is countable when either A or B is finite and its proof is left as an exercise. When both A and B are countably infinite, without loss of generality, we assume $A = B = N$. Let $g(k) = k(k+1)/2$, which is the sum of the first k positive integers, and $f : N \times N \longrightarrow N$ be the function:

$$f(i, j) = g(i + j) + j$$

The inverse of f exists: For any $k \in N$, let m be the integer such that $g(m) \leq k < g(m + 1)$. Then $f^{-1}(k) = \langle i, j \rangle$, where $i = g(m) + m - k$ and $j = k - g(m)$. It is easy to check that $i + j = m$ and $f(i, j) = k$. For example, if $k = 12$, then $m = 4$, and $i = j = 2$. Table 1.3.1 shows the first few values of $k = f(i, j)$. Recall that f is bijective iff f^{-1} exists. Since the inverse of f exists, f is bijective.

$i \backslash j$	0	1	2	3	4
0	0	2	5	9	14
1	1	4	8	13	19
2	3	7	12	18	25
3	6	11	17	24	32
4	10	16	23	31	40

Table 1.3.1: $k = f(i, j) = (i + j)(i + j + 1)/2 + j$.

To show (4) is countable, we define a bijective function f from the set of integers to N: $f(x) =$ if $x \geq 0$ then $2x$ else $-2x - 1$. This function maps 0 to

0; positive integers to even natural numbers, and negative integers to odd natural numbers. f is bijective because the inverse of f exists: For any $n \in N$, $f^{-1}(n) =$ if n is even then $n/2$ else $(-n-1)/2$.

To show (5) is countable, let $S = \{0,1\}^*$ be all the binary strings, we sort S by the *canonical order* which compares the length of strings first, then the decimal value of strings. Thus, the strings will be listed as follows:

$$S = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, ...\},$$

where ϵ denotes the empty string. To find an injection h from S to N, for any string w in S, if $w = \epsilon$, let $h(\epsilon) = 0$; and $h(w)$ will replace every 0 in w by 2 and leaving 1 intact,

$$h(S) = \{0, 2, 1, 22, 21, 12, 11, 222, 221, ...\}.$$

It is easy to check that $h : S \longrightarrow N$ is injective. So S is countable.

We can do better by finding a bijection between S and N. For any binary string s, let the length of s be n, i.e., $|s| = n$. There are $1+2+2^2+...+2^{n-1} = 2^n-1$ strings shorter than s. Let $v(s)$ be the decimal value of s: $v(\epsilon) = 0$, $v(x0) = 2v(x)$ and $v(x1) = 2v(x) + 1$, then there are $v(s)$ strings of length n before s when S is listed by the canonical order. So the position of s in the sorted list is $2^n + v(s)$. Define $f : S \longrightarrow N$: $f(s) = 2^{|s|} + v(s) - 1$, then f must be a bijection.

To show (6) is countable, let $k = \lceil log_2(|\Sigma|) \rceil$. Since $|\Sigma| \leq 2^k$, for each symbol $a \in \Sigma$, let $h(a)$ be a distinct binary string of length k. For each string $w \in \Sigma^*$, let $h(w) = h(a_1) \cdots h(a_n)$ if $w = a_1 \cdots a_n$. Then $h : \Sigma^* \longrightarrow S$, where $S = \{0,1\}^*$, is an injective function. It is easy to construction an injective function $g : \Sigma^* \longrightarrow N$ by the composition of h and $f : S \longrightarrow N$ from (5), so Σ^* is countable. □

The last three cases in the above proposition are examples of countably infinite sets.

1.3.1.4 Uncountable Sets

In 1874, in his first set theory article, Georg Cantor introduced the term "countable set" and proved that the set of real numbers is uncountable by the famous diagonalization method, thus showing that not all infinite sets are countable. In 1878, he used bijection to define and compare cardinalities, contrasting sets that are countable with those that are uncountable.

Proposition 1.3.5. *The following sets are uncountable:*

1. *B: the set of binary strings of infinite length;*

$s_0 =$	**0**	0	0	0	0	0	0	0	0	0 ...
$s_1 =$	0	**0**	0	0	0	0	0	0	1	0 ...
$s_2 =$	0	0	**0**	0	0	1	0	0	0	0 ...
$s_3 =$	0	0	0	**1**	0	0	0	0	0	0 ...
$s_7 =$	0	0	0	1	**0**	1	0	0	0	0 ...
$s_6 =$	0	0	0	0	1	**0**	0	0	0	0 ...
$s_4 =$	0	0	0	1	0	1	**0**	1	0	0 ...
$s_5 =$	0	0	1	0	0	0	0	**1**	0	0 ...
$s_8 =$	0	1	0	0	1	0	0	1	**0**	0 ...
$s_9 =$	0	1	0	0	1	1	0	1	0	**1** ...
					.....					
$x =$	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0** ...

Table 1.3.2: Illustration of Cantor's diagonalization method: If B can be enumerated as $s_0, s_1, s_2, ...$, then $x = \overline{s_{00}}\,\overline{s_{11}}\,\overline{s_{22}}...$ is a binary string but cannot be in B.

2. *R_1: the set of non-negative real numbers less than 1;*

3. *$\mathcal{P}(N)$: the power set of the natural numbers;*

4. *F: The set of Boolean functions, $f : N \longrightarrow \{0, 1\}$, over the natural numbers.*

Proof. For the first set B, we will use Cantor's diagonalization method to show that B is uncountable by a refutational proof. Assume that B is countable, then there exists a one-to-one correspondence between B and N. For each $i \in N$, let the corresponding string in B be s_i, and the j^{th} symbol in s_i be s_{ij}. Now, we construct a binary string x of infinite length as follows: the j^{th} symbol of x, i.e., x_j, is the complement of s_{jj}. That is, if $s_{jj} = 0$, then $x_j = 1$; if $s_{jj} = 1$, then $x_j = 0$. Clearly, $x \in B$. Suppose x's corresponding number in N is k, i.e., $x = s_k$. However, x differs from s_k on the k^{th} symbol, because x_k is the complement of s_{kk}. The contradiction comes from the assumption that B is countable. Table 1.3.2 illustrates the idea of Cantor's diagonalization method, where $\overline{0} = 1$ and $\overline{1} = 0$.

Once we knew B is uncountable, if we have a bijection from B to R_1, then R_1 must be uncountable. The bijection $f : B \longrightarrow R_1$ can be simply $f(s) = 0.s$, as each number of R_1 can be represented by a binary number, where $s \in B$.

To show $\mathcal{P}(N)$ is uncountable, let $f : B \longrightarrow \mathcal{P}(N)$ be $f(s) = \{j \mid j^{th}$ symbol of s is 1 $\}$. Obviously f is bijective.

Similarly, to show F is uncountable, let $f : B \longrightarrow F$ be $f(s) = f$ if for every j, $f(j) =$ the j^{th} symbol of s. Obviously f is bijective, too. □

The result that $\mathcal{P}(N)$ is uncountable while N is countable can be generalized: for every set S, the power set of S, i.e., $\mathcal{P}(S)$, has a larger cardinality than S itself. This result implies that the notion of the *set of all sets* is an inconsistent notion. If S were the set of all sets then $\mathcal{P}(S)$ would at the same time be bigger than S and a subset of S.

Theorem 1.3.6. (**Cantor's Theorem**) $|A| < |\mathcal{P}(A)|$ *for any set* A.

Proof. Let us prove it by contradiction. If $|A| = |P(A)|$, there is a bijection $f : A \longrightarrow \mathcal{P}(A)$. Define $S = \{a \in A \mid a \notin f(a)\}$, then $S \subseteq A$ and $S \in \mathcal{P}(A)$. There must exist $b \in A$ such that $f(x) = S$ because f is surjective. There only two possibilities: $b \in S$ or $b \notin S$.

1. If $b \in S$, by definition of S, $b \notin f(b) = S$.
2. If $b \notin S$, by definition of S, $b \in f(b) = S$.

Both cases lead to a contradiction, so f cannot exist. It cannot be the case that $|A| > |P(A)|$ because $g(a) = \{a\}$ is an injection from A to $\mathcal{P}(A)$. □

Cantor's theorem implies that there is no such thing as the "set of all sets". Suppose A were the set of all sets. Since every element of $\mathcal{P}(A)$ is a set, so $P(A) \subseteq A$. Thus $|P(A)| \leq |A|$, a contradiction to Cantor's theorem.

Cantor chose the symbol $\aleph_0$ for the cardinality of n, $|N|$, a cardinal number. $\aleph_0$ is read as aleph-null, after the first letter of the Hebrew alphabet. The cardinality of the reals is often denoted by $\aleph_1$, or c for the continuum of real numbers. Cantor's theorem implies that there are infinitely many infinite cardinal numbers, and that there is no largest cardinal number.

$$\begin{array}{lcll} \aleph_0 & = & |N| & \\ \aleph_1 & = & |\mathcal{P}(N)| & = \ 2^{\aleph_0} > \aleph_0 \\ \aleph_2 & = & |\mathcal{P}(\mathcal{P}(N))| & = \ 2^{\aleph_1} > \aleph_1 \\ \aleph_3 & = & |\mathcal{P}(\mathcal{P}(\mathcal{P}(N)))| & = \ 2^{\aleph_2} > \aleph_2 \\ \ldots & & & \end{array}$$

The famous inconsistent example related to set theory is so-called *Russell's paradox*, discovered by Bertrand Russell in 1901:

Paradox 1.3.7. (**Russell's paradox**) Let $T = \{S \mid S \notin S\}$, the set T of all sets S that does not contain itself. What is the truth value of $T \in T$?

If "$T \in T$" is false, then $T \in T$ by the condition $S \notin S$. If "$T \in T$" is true, then $T \notin T$ because every member of T satisfies $S \notin S$. This paradox showed

that some attempted formalization of the naïve set theory created by Georg Cantor led to a contradiction. Russell's paradox shows that the concept of "a set contains itself" is invalid. If "$S \in S$" is always false, then $T = \{S \mid S \notin S\}$ is the "set of all sets", which does not exist, either.

A layman's version of Russell's paradox is called the *Barber's paradox*.

Paradox 1.3.8. (**Barber's paradox**) In a village, there is only one barber who shaves all those, and those only, who do not shave themselves. The question is, does the barber shave himself?

Answering this question results in a contradiction. The barber cannot shave himself as he only shaves those who do not shave themselves. Conversely, if the barber does not shave himself, then he fits into the group of people who would be shaved by the barber, and thus, as the barber, he must shave himself.

The discovery of paradoxes in informal set theory caused mathematicians to wonder whether mathematics itself is inconsistent, and to look for proofs of consistency. Ernst Zermelo (1904) gave a proof that every set could be well-ordered, using the *axiom of choice*, which drew heated debate and research among mathematicians and the pioneers of set theory. In 1908, Zermelo provided the first set of axioms for set theory. These axioms, together with the additional axiom of replacement proposed by Abraham Fraenkel, are now called Zermelo–Fraenkel set theory (ZF). Besides ZF, many set theories are proposed since then, to rid paradoxes from set theory.

1.3.2 Model Theory

Model theory is the study of mathematical structures (e.g. groups, fields, algebras, graphs, logics) in a formal language. Every formal language has its syntax and semantics. Models are a semantic structure associated with syntactic structures in a formal language. Theories are then introduced based on models. Following this approach, every formal logic is defined inside of a formal language.

1.3.2.1 Syntax and Semantics

The syntax of a formal language specifies how various components of the language, such as symbols, words, and sentences, are defined. For example, the language for propositional logic uses only propositional variables and logic operators as its symbols, and well-formed formulas built on these symbols as its sentences. In model theory, a set of sentences in a formal language is one of the components that form a theory. The language for logic often contains the two constant symbols,

either 1 and 0, or $\top$ and $\bot$, which are interpreted as true and false, respectively. They are usually considered to be special logical operators which take no arguments, not propositional variables.

The semantics of a language specifies the meaning of various components of the language. For example, if we use symbol q to stands for the statement "Today is Monday", and r for "Today is Tuesday", then the meaning of q is "Today is Monday"; q can be used to denote a thousand different statements, just like a thousand Hamlets in a thousand people's eyes. On the other hand, a formal meaning of the formula $q \vee r$ is the truth value of $q \vee r$, which can be decided uniquely when the truth values of q and r are given. In model theory, the formal meaning of a sentence is explored: it examines semantic elements (meaning and truth) by means of syntactical elements (formulas and proofs) of a corresponding language.

In model theory, semantics and model are synonyms. A *model* of a theory is an interpretation that satisfies the sentences of that theory. Abstract algebras are often used as models. In a summary definition, dating from 1973,

$$\text{model theory} = \text{abstract algebra} + \text{logic},$$

Abstract algebra, also called *universal algebra*, is a broad field of mathematics, concerned with sets of abstract objects associated various operations and properties.

1.3.2.2 Boolean Algebra

In abstract algebra, the most relevant algebra related to the logic discussed in this book is *Boolean algebra*, which is approximately equivalent to propositional logic. Many syntactic concepts of Boolean algebra carry over to propositional logic with only minor changes in notation and terminology, while the semantics of propositional logic are defined via Boolean algebras in a way that the tautologies (theorems) of propositional logic correspond to equational theorems of Boolean algebra.

In Boolean algebra, the values of the variables are the truth values true and false, usually denoted 1 and 0 respectively. The main operations of Boolean algebra are the multiplication $\cdot$ (conjunction), the addition $+$ (disjunction), and the inverse i (negation). It is thus a formalism for describing logical operations in the same way that elementary algebra describes numerical operations, such as addition and multiplication. In fact, a *Boolean algebra* is any set with binary operations $+$ and $\cdot$ and a unary operation i thereon satisfying the Boolean laws (equations), which define the properties of the logical operations.

Boolean algebra was introduced by George Boole in his first book *The Mathematical Analysis of Logic* (1847). According to Huntington, the term "Boolean

algebra" was first suggested by Sheffer in 1913. Sentences that can be expressed in propositional logic have an equivalent expression in Boolean algebra. Thus, Boolean algebra is sometimes used to denote propositional logic performed in this way. However, Boolean algebra is not sufficient to capture logic formulas using quantifiers, like those from first order logic.

1.3.3 Proof Theory

In a similar way to model theory, proof theory is situated in an interdisciplinary area among mathematics, philosophy, and computer science. Proof theory is a major branch of mathematical logic that represents proofs as formal mathematical objects, facilitating their analysis by mathematical techniques.

1.3.3.1 Axioms and Theorems

In a formal logic, the *axioms* are a set of sentences which are assumed to be true. Typically, the axioms contain by default the definitions of all logical operators. For example, the negation operator $\neg$ is defined by $\neg(\top) = \bot$ and $\neg(\bot) = \top$.

The *theorems* are the formulas which can be proved to be true from the axioms. For example, for any propositional variable p, $\neg(\neg(p)) = p$ is a theorem. Using the case analysis method, if the truth value of p is $\top$, then $\neg(\neg(\top)) = \neg(\bot) = \top$; if the truth value of p is $\bot$, then $\neg(\neg(\bot)) = \neg(\top) = \bot$. It is easy to see that different sets of axioms would lead to different theorems.

There are two important properties concerning a set of axioms:

- **Consistency** The axiom set is *consistent* if every formula in the axiom can be true at the same time.
- **Independence** The axiom set is *independent* if no axiom is a theorem of the other axioms.

1.3.3.2 Proof Procedures

Given a set $\mathbf{A}$ of axioms and a formula B, we need to a procedure P that can answer the question if B is a theorem of $\mathbf{A}$: If $P(\mathbf{A}, B)$ returns "yes", we say B is proved, if $P(\mathbf{A}, B)$ returns "no", we say B is disproved; if it returns "unknown", we do not know if B is a theorem or not. P is called a *proof procedure*. This procedure can be carried out by hand, or executed on a computer.

There are two important properties concerning a proof procedure P:

- **Soundness** The proof procedure P is *sound* iff whenever $P(\mathbf{A}, B)$ returns true, B is a theorem of $\mathbf{A}$; whenever $P(\mathbf{A}, B)$ returns false, B is not a theorem of $\mathbf{A}$.

- **Completeness** The proof procedure P is *complete* iff for any theorem B of $\mathbf{A}$, $P(\mathbf{A}, B)$ returns true.

If a procedure $P(\mathbf{A}, B)$ is sound and always halts on any input $\mathbf{A}$ and B with "yes" or "no", i.e., P is an algorithm, then P is a *decision procedure* for the logic. In general, a *decision procedure* is usually referred to any procedure which always halts with a correct answer to a decision question, such as "A is valid or not".

Theorem 1.3.9. *Every decision procedure is sound and complete.*

Proof. Given any axiom set $\mathbf{A}$ and any formula B, since P always halts, $P(\mathbf{A}, B)$ will return a truth value. If B is a theorem of $\mathbf{A}$, $P(\mathbf{A}, B)$ will return true because the answer returned by P must be correct. □

Some proof procedures may stop with "yes", "no", or "unknown"; they cannot be called a decision procedure. A sound and complete proof procedure may loop on a formula that is not a theorem, thus cannot be a decision procedure by definition. From the above theorem, we see that the soundness and halting properties of a proof procedure implies the completeness. A sound and complete proof procedure is also called a *semi-decision procedure* for the logic, because it will halt on any formula which is a theorem. For any logic, we always look for its decision procedure. For example, propositional logic has many different decision procedures. If a decision procedure does not exist, we look for a semi-decision procedure. For some logics, even a semi-decision procedure cannot exist. Clearly, these discussions will only make sense once we have formally defined the logic in question.

1.3.3.3 Inference Rules

In structural proof theory, which is a major area of proof theory, inferences rules are used to construct a proof. In logic, an *inference rule* is a pattern which takes formulas as premises, and returns a formula as conclusion. For example, the rule of *modus ponens* (MP) takes two premises, one in the form "If p then q" (i.e., $p \to q$) and another in the form p, and returns the conclusion q'. The rule is *sound* if the premises are true under an interpretation, then so is the conclusion.

An inference system S consists of a set of axioms A and a set of inference rules R. The soundness of S comes from the soundness of every inference rule in R. A *proof* of formula F_n in $S = (A, R)$ is a sequence of formulas $F_1, F_2, ..., F_n$ such

that each F_i is either in A or can be generated by an inference rule of R, using the formulas before F_i in the sequence as the premises.

Example 1.3.10. Let $S = (A, R)$, $A = \{p \to (q \to r), p, q\}$, and $R = \{MP\}$. A proof of r from A using S is given below:

1.	$p \to (q \to r)$	$axiom$
2.	p	$axiom$
3.	$q \to r$	$MP, 1, 2$
4.	q	$axiom$
5.	r	$MP, 3, 4$

□

One obvious advantage of such a proof is that these proofs can be checked either by hand or automatically by computer. Checking formal proofs is usually simple, whereas finding proofs (automated theorem proving) is generally hard. An informal proof in the mathematics literature, by contrast, requires weeks of peer review to be checked, and may still contain errors. Informal proofs of everyday mathematical practice are unlike the proofs of proof theory. They are rather like high-level sketches that would allow an expert to reconstruct a formal proof at least in principle, given enough time and patience. For most mathematicians, writing a fully formal proof is too pedantic and long-winded to be in common use.

In proof theory, proofs are typically presented as recursively-defined data structures such as lists (as shown here), or trees, which are constructed according to the axioms and inference rules of the logical system. As such, proof theory is syntactic in nature, in contrast to model theory, which is semantic in nature.

Every inference system can be converted to a proof procedure if a *fair strategy* is adapted to control the use of inference rules. Fair strategy ensures that if an inference rule is applicable at some point to produce a new formula, this new formula will be derived eventually. If every inference rule is sound, the corresponding proof procedure is sound. Extra effort is needed in general to show that the proof procedure is complete, or halting.

We have defined a formal notion of proof as a sequence of formulae, starting with axioms and continuing with theorems derived from earlier members of the sequence by rules of inference.

Besides structural proof theory, some of the major areas of proof theory include ordinal analysis, provability logic, reverse mathematics, proof mining, automated theorem proving, and proof complexity.

1.3.4 Computability Theory

Computability theory, used to be called *recursion theory*, is a branch of mathematical logic, of computer science, and of the theory of computation that originated in the 1930s with the study of computable functions. The field has since expanded to include the study of generalized computability and definability. In these areas, computability theory overlaps with proof theory and set theory.

1.3.4.1 Recursion

The main reason for the name of "recursion theory" is that recursion is used to construct objects and functions.

Example 1.3.11. Given a constant symbol 0 of type T and a function symbol $s : T \longrightarrow T$, the objects of type T can be recursively constructed as follows:

1. 0 is an object of type T;
2. If n is an object of type T, so is $s(n)$;
3. Nothing else will be an object of T.

If we let $s^0(0) = 0$, $s^1(0) = s(0)$, $s^2(0) = s(s(0))$, etc., then T can be expressed as

$$T = \{0, s(0), s^2(0), s^3(0), ..., s^i(0), ...\}.$$

Obviously, there exists a bijection f between T and N, the set of natural numbers. That is, for any $i \in N$, $f(i) = s^i(0)$. By giving s the meaning of *successor* of a natural number, mathematicians use T as a definition of N. □

Functions can be recursively defined in a similar way. Let *pre*, *add*, *sub*, *mul* be the predecessor, addition, subtraction, and multiplication functions over the set of natural numbers:

$$\begin{aligned}
pre(0) &= 0; \\
pre(s(x)) &= x. \\
add(0, y) &= y; \\
add(s(x), y) &= s(add(x, y)). \\
sub(x, 0) &= x; \\
sub(x, s(y)) &= sub(pre(x), y)). \\
mul(0, y) &= 0; \\
mul(s(x), y) &= add(mul(x, y), y).
\end{aligned}$$

Note that the subtraction defined above is different from the subtraction over the integers: we have $sub(s^i(0), s^j(0)) = 0$ when $i \leq j$. This function holds the property that $sub(x, y) = 0$ iff $x \leq y$.

The above functions are examples of so-called *primitive recursive functions*, which are total and computable functions. The set of *general recursive functions* include all primitive recursive functions, and some functions which are not computable.

1.3.4.2 Backus-Naur Form (BNF)

In computer science, *Backus–Naur form* (BNF) is a notation technique for *context-free grammars*, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols. John Backus and Peter Naur are the first to use this notation to describe ALGOL, an early programming language. BNF is applied wherever formal descriptions of languages are needed. We will use BNF describe the syntax of logic as a language.

BNF can be extended to define recursively constructed objects in a succinct style. For example, the set of natural numbers is expressed in BNF as follows:

$$\langle N \rangle ::= 0 \mid s(\langle N \rangle)$$

The symbol ::= can be understood as "is defined by", and "—" separates alternative parts of a definition. The above formula can be read as follows: the member of N is either 0, or s applying to (another) member of N; nothing else will be in N. From this example, we see that to define a set N, we use any member of N is denoted by $\langle N \rangle$ and is defined by the items on the right of ::=. For a recursive definition, we have a basic case (such as 0) and a recursive case (such as $s(\langle N \rangle)$). Items like $\langle N \rangle$ are called *variables* in a context-free grammar and items like 0, s, (, and) are called *terminals*. A context-free grammar specifies what set of strings of terminals can be derived for each variable. For the above example, the set of strings derived from $\langle N \rangle$ is

$$\{0, s(0), s^2(0), s^3(0), ..., s^i(0), ...\}$$

This set is the same as the objects defined in Example 1.3.11, but BNF gives us a formal and compact definition.

Example 1.3.12. To define the set Σ^* of all strings (of finite length) over the set Σ of symbols, called the alphabet, we may use BNF. For instance, if $\Sigma = \{a, b, c\}$,

then Σ^* is the set of all binary strings:

$$\begin{aligned} \langle symbol \rangle &::= a \mid b \mid c \\ \langle \Sigma^* s \rangle &::= \epsilon \mid \langle symbol \rangle \langle \Sigma^* \rangle \end{aligned}$$

That is, the empty string $\epsilon \in \Sigma^*$; if $w \in \Sigma^*$, for every symbol $a \in \Sigma$, $aw \in \Sigma^*$; nothing else will be in Σ^*. Thus, Σ^* contains exactly all the strings built on Σ:

$$\Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, ...\}.$$

□

Since there is a one-to-one corresponding between a string and a sequence, sequences can be also recursively defined in a similar way. In fact, all recursively constructed objects can be defined this way.

1.3.4.3 Computable Functions

One of the basic questions addressed by computability theory is the following:

- What does it mean for a function on the natural numbers to be computable?

The answer to the question is that the function is computable if there exists an algorithm which computes the function. Logicians and computer scientists, including Turing and Church, gave different definitions of algorithms based on different computing models. These computing models can be Turing machines, Church's λ-calculus, general recursive functions, Markov algorithms, Chomsky's grammars, Minski's counter machine, or von Neumann model. One point in common is that, by algorithm, we mean that the computing process will halt in a finite number of steps.

In section 1.3.1.4, we saw that the set of functions, which take a natural number and return a Boolean value, is uncountable. However, the set of algorithms is countable, because every algorithm can be recorded by a text file of finite length (which can be regarded as a binary string), and the set of binary strings is countable. In other words, we have only a countable set of algorithms but the functions on the natural numbers are uncountable. As a result, there exist massive number of functions on the natural numbers which are not computable.

One notable non-computable problem is to decide if a Turing machine T will halt on an input x. If we encode T and x as a single positive integer $\langle T, x \rangle$, and define the function $f(\langle T, w \rangle) = 1$ if T halts on w and 0, otherwise, then f is a non-computable function on the natural numbers. Note that the Turing machine

T can be replaced by any other equivalent computing model and the result still holds. This follows from the *Church–Turing thesis* (also known as computability thesis, the Church–Turing conjecture), which states that a function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine.

The concept of *Turing completeness* says that if a computing model has the ability to simulate a Turing machine. A computing model that is Turing complete is theoretically capable of doing all tasks done by computers; on the other hand, nearly all computers are Turing complete if the limitation of finite memory is ignored. Some logics are also Turing complete as they can also be used to simulate a Turing machine. As a result, some problems such as the halting problem for such logics are undecidable. Computability theory will help us to decide if there exist or not decision procedures for some logic, and we will see some examples in Chapter 9.

One of the main aims of this book is to present algorithms invented for logic and the software tools built up based on these algorithms. That is why we will invest a disproportionate amount of effort (by the standards of conventional logic textbooks) in the algorithmic aspect of logic. Set theory and model theory help us to understand and specify the problems; proof theory helps us to create algorithms; and computability theory helps us to know what problems have algorithms and what problems do not have algorithms at all.

1.4 Exercise Problems

1. Decide if the following sentences are statements or not. If they are, use propositional symbols and logic connectives to express them.

   (a) If I win the lottery, I'll be poor.

   (b) The man asked, "shut the door behind you."

   (c) Today is the first day of the school.

   (d) Today is either sunny or cloudy.

2. Answer the questions in the following examples regarding logical fallacies.

   (a) During their argument, Anthony tells Marie that he hates her and wished that she would get hit by a car. Later that evening, Anthony receives a call from a friend who tells him that Marie is in the hospital because she was struck by a car. Anthony immediately blames himself and reasons that if he hadn't made that comment during their fight, Marie would not have been hit. What logical fallacy has Anthony committed?

(b) Louise is running for class president. In her campaign speech she says, "My opponent does not deserve to win. She is a smoker and she cheated on her boyfriend last year." What fallacy has Louise committed?

(c) Bill: "I think capital punishment is wrong." Adam: "No it isn't. If it was wrong, it wouldn't be legal." Of which fallacy is this an example?

(d) Ricky is watching television when he sees a commercial for foot cream. The commercial announcer says, "This is the best foot cream on the market because no other foot cream makers have been able to prove otherwise!" What fallacy has the announcer committed?

(e) Maria has been working at her current job for more than 30 years at the same wage. She desperately wants a raise so she approaches her boss to ask for one. She says, "You are one of the kindest people I know. You are smart and good looking and I really love your shoes." What type of fallacy is this?

(f) Jeff's mom is concerned when she finds out that he skipped class one day. She tells him that she is concerned that since he skipped one class, he will start skipping more frequently. Then he will drop out altogether, never graduate or get into college, and end up unemployed and living at home for the rest of his life. What type of fallacy has Jeff's mom committed?

(g) Dana is trying to raise money for her university's library. In her address to the board of trustees, she says, "We must raise tuition to cover the cost of new books. Otherwise the library will be forced to close." Of what fallacy is this an example?

(h) Jeff is preparing to create a commercial for a new energy drink. He visits a local high school and surveys students in an English class about their beverage preferences. The majority of the class says they prefer grape flavored drinks, so Jeff tells his superiors that grape is the flavor favored most by high school students. What error in reasoning has Jeff made?

3. Let $A = \{0, 1\}$. $\mathcal{P}(A) = ?$ $\mathcal{P}(\mathcal{P}(A)) = ?$

4. Let a, b be real numbers and $a < b$. Define a function $f : (0, 1) \longrightarrow (a, b)$ by $f(x) = a + x(b - a)$, where (a, b) denotes the interval $\{x \mid a < x < b\}$. Show that f is bijective.

5. Answer the following questions with justification.

(a) Is the set of all odd integers countable?

(b) Is the set of real numbers in the interval $(0, 0.1)$ countable?

(c) Is the set of angles in the interval $(0^o, 90^o)$ countable?

(d) Is the set of all points in the plane with rational coordinates countable?

(e) Is the set of all Java programs countable?

(f) Is the set of all words using an English alphabet countable?

(g) Is the set of sands on the earth countable?

(h) Is the set of atoms in the solar system countable?

6. **Barber's paradox**: In a village, there is only one barber who shaves all those, and those only, who do not shave themselves. The question is, does the barber shave himself? Let the relation $s(x, y)$ be that "x shaves y" and b denote the barber.

   (a) Define A to be the set of the people shaved by the barber using $s(x, y)$ and b.

   (b) Define B to be the set of the people who does not shave himself.

   (c) Use A and B to show that the answer to the question "does the barber shave himself?" leads a contradiction.

7. Prove that any subset of a countable set is countable.

8. Prove that the set of rational numbers is countable.

9. Prove that the set $N \times \{a, b, c\}$ is countable, where N is the set of natural numbers. Your proof should not use the property that countable sets are closed under Cartesian product.

10. Decide with a proof that the set N^k is countable or not, where N is the set of natural numbers and $k \in N$.

11. Decide with a proof that the set of all functions $f : \{0, 1\} \longrightarrow N$ is countable or not.

12. Prove that if both A and B are countable, so is $A \cup B$.

13. Prove that if A is uncountable and B is countable, then $A - B$ is uncountable.

14. Prove that the set $\{a, b, c\}^*$ of all strings (of finite length) on $\{a, b, c\}$ is countable.

15. If a language is any set of strings in Σ^*, where $\Sigma \neq \emptyset$, prove that the set of all languages is uncountable.

16. Suppose the set of natural numbers is constructed by the constant 0 and the successor function s. Provide a definition of $<$ and $\leq$ over the natural numbers. No functions are allowed to use.

17. Provide a definition of power function, $pow(x, y)$, which computes x^y, over the natural numbers built up by 0 and the successor function s. You are allowed to use the functions *add* (addition) and *mul* (multiplication).

18. Provide a BNF (Backus–Naur form) for the set of binary trees, where each non-empty node contains a natural number. You may use $null : Tree$ for the empty tree and $node : Nat, Tree, Tree \longrightarrow Tree$ for a node of the tree.

CHAPTER 2
PROPOSITIONAL LOGIC

Following model theory, we will present propositional logic as a language. We first introduce the syntax of this language and then its semantics.

2.1 Syntax

The syntax of propositional logic specifies what symbols and what sentences are used in the logic. Propositional logic uses *propositional variables* and *logical operators* as the only symbols. In the first chapter, we said that propositional variables are symbols for denoting statements and take only true and false as truth values. For convenience, we will use 1 for true and 0 for false. We will treat $\top$ and $\bot$ as the two nullary logical operators such that the truth value of $\top$ is always 1 and the truth value of $\bot$ is always 0. Note that $\top$ and $\bot$ are syntactic symbols while 1 and 0 are semantic symbols.

2.1.1 Logical Operators

Mathematicians use the words "not", "and", "or", etc., for operators that change or combine propositions. The meaning of these logical operators can be specified as a function which takes Boolean values, and returns a Boolean value. These functions are also called . For example, if p is a proposition, then so is $\neg p$ and the truth value of the proposition $\neg p$ is determined by the truth value of p according to the meaning of $\neg$: $\neg(1) = 0$ and $\neg(0) = 1$. That is, if the value of p is 1, then the value of $\neg(p)$ is 0; if the value of p is 0, then the value of $\neg(p)$ is 1.

In general, the definition of a Boolean function is displayed by a table where the output of the Boolean function is given for each possible truth values of the input variables. Such a table is called *truth table*. For example, the meanings of $\wedge$, $\vee$, and $\rightarrow$ are defined by the following truth table: This table has four lines, since there are four pairs of Boolean values for the two variables:

According to this table, the truth value of $p \wedge q$ is 1 when both p and q are 1. The truth value of $p \vee q$ is 1 when either p or q, or both are true. This is not always the intended meaning of "or" in everyday dialog, but this is the standard definition in logic. So if a logician says, "You may have cake, or you may have ice cream," he means that you could have both. If you want to exclude the possibility of having

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \oplus q$	$p \leftrightarrow q$
1	1	1	1	1	0	1
1	0	0	1	0	1	0
0	1	0	1	1	1	0
0	0	0	0	1	0	1

Table 2.1.1: Definitions of $\wedge$, $\vee$, $\rightarrow$, $\oplus$, and $\leftrightarrow$

both cake and ice cream, you should combine them with the exclusive-or operator, $\oplus$, which is also defined in Table 2.1.1. The exclusive disjunction $\oplus$ corresponds to addition modulo 2 and is therefore given the symbol + when the boolean values are assumed to be 0 and 1.

In Table 2.1.1, we also give the definition of $\leftrightarrow$ (if and only if). If p denotes "Tom wears a blue jean" and q denotes "Sam wears a blue jean", then the formula $p \leftrightarrow q$ asserts that "Tom and Sam always wear a blue jean at the same time". That is, the value of $p \leftrightarrow q$ is 1 when p and q have the same truth value: either both are 1 or both are 0. Note that the $p \oplus q$ and $p \leftrightarrow q$ always have opposite values, that is, $\neg(p \oplus q)$ and $p \leftrightarrow q$ always have the same value.

The truth table for implications is also given in Table 2.1.1. This operator is worth remembering, because a large fraction of all mathematical statements are of the if-then form. Now let's figure out the truth of the following example: If elephants fly, I'll be on Mars. What is the truth value of this statement? Elephants do not fly, so the truth value of p is 0 and we fall to the last two lines of Table 2.1.1: In both lines, $p \rightarrow q$ is 1. In section 1.2.2 on Informal Fallacies, we pointed out that with a false premise, you can make any conclusion, so that the composed statement is always true. However, an always true statement has no value in reasoning, and it does not imply the causal connection between the premise and the conclusion of an implication. In logic, it is important to accept the fact that logical implications ignore causal connections.

Let $B = \{0, 1\}$, any logical operator is a function $f : B^n \longrightarrow B$. There are two nullary operators (when $n = 0$): $\top = 1$ and $\bot = 0$. There are four unary operators (when $n = 1$): $f_1(x) = 0$; $f_2(x) = 1$; $f_3(x) = x$; and $f_4(x) = \neg x$. There are 16 binary operators (when $n = 2$), and $\wedge$, $\vee$, $\rightarrow$, $\oplus$, and $\leftrightarrow$ are some of them. We may use operators when $n > 2$. For example, let $ite : B^3 \longrightarrow B$ be the if-then-else operator, if we use a truth table for the meaning of $ite(p, q, r)$, then the truth table will have eight lines. In general, there are 2^{2^n} n-ary Boolean functions, as the number of functions $f : A \longrightarrow B$ is $|B|^{|A|}$, where $|A| = |B^n| = |B|^n = 2^n$.

2.1.2 Formulas

Not every combination of propositional variables and logic operatorsis makes sense. For instance, $p\ ()\ q \vee \neg)$ is a meaningless string of symbols. The *propositional formulas* are the set of well-formed strings built up by propositional variables and logical operators by a rigorous set of rules.

We may add/delete operators or propositional variables at will in the above definition. According to the definition, every member of $Formulas$ is either the constants $\top$ or $\bot$, a propositional variable of V_P, or the negation of a well-formula, or a binary operator applying to two formulas. Here are some examples:

$$\top,\quad p,\quad \neg q,\quad (p \wedge \neg q),\quad (r \to (p \wedge \neg q)),\quad ((p \wedge \neg q) \vee r)$$

Throughout this chapter, we will use p, q, ..., for propositional variables, and A, B, ..., for formulas.

For every binary operator, we introduce a pair of parentheses, and some of them may be unnecessary and we will use a precedence relation over all the operators to remove them. The typical precedence is given as the following list, where operators with higher precedence go first. Note that we assign $\oplus$ and $\leftrightarrow$ the same precedence.

$$\neg,\ \wedge,\ \vee,\ \ \to,\ \{\oplus, \leftrightarrow\}.$$

Thus $(r \to (p \wedge \neg q))$ can be simply written as $r \to p \wedge \neg q$, and we will not take it as $(r \to p) \wedge \neg q$, because $\wedge$ has higher precedence than $\to$. Similarly, $((p \wedge \neg q) \vee r)$ is written as $p \wedge \neg q \vee r$ without ambiguity.

When a well-formed expression, like the formulas defined above, is represented by a string (a sequence of symbols), we use parentheses and commas to identify the structures of the expression. Such a structure can be easily identified by a tree, where we do not need parentheses and commas.

Definition 2.1.1. *A* tree *for formula A is defined as follows:*

1. *If A is p, $\top$, or $\bot$, the tree is a single node with A as its label.*
2. *If $A = \neg B$, then the root node is labeled with $\neg$ and has one branch pointing to the formula tree of B.*
3. *IF $A = B$ op C, then the root node is labeled with op and has two branches pointing to the formula trees of B and C, respectively.*

For example, the formula tree of $(\neg p \wedge q) \to (p \wedge (q \vee \neg r))$ is displayed in Figure 2.1.1. The tree representation of a formula does not need parentheses.

A formula and its formula tree are the two representations of the same object. We are free to choose one of the representations for the convenience of discussion.

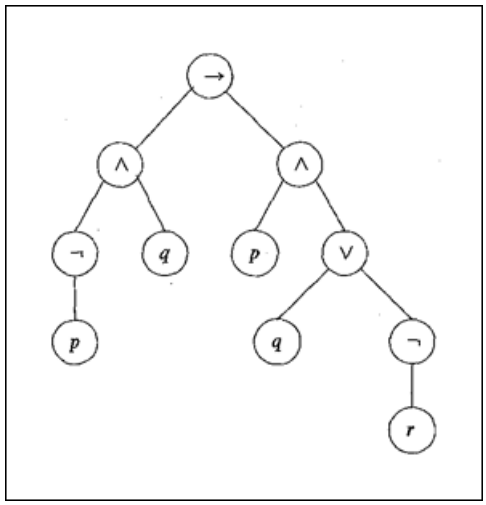


Figure 2.1.1: The formula tree for $(\neg p \wedge q) \rightarrow (p \wedge (q \vee \neg r))$

Definition 2.1.2. *Given a formula A, we say B is a* subformula *A if either $B = A$ or $A = (A_1 \; op \; A_2)$, where op is a binary operator, or $A = \neg A_1$ and B is a subformula of A_1 or A_2. If B is a subformula of A and $B \neq A$, then B is a proper subformula of A.*

For example, $\neg p \vee \neg(q \vee r)$ contains p, q, r, $\neg p$, $q \vee r$, and $\neg(q \vee r)$ as proper subformulas. Intuitively, a subformula is just a formula represented by any subtree in the formula tree.

Definition 2.1.3. *Given a subformula B of A and another formula C, we use $A[B \leftharpoonup C]$ to denote the formula obtained by substituting all occurrences of B by C in A. If B is a propositional variable, then $A[B \leftharpoonup C]$ is called an* instance *of A.*

For example, if A is $\neg p \vee \neg(q \vee r)$, B is $\neg(q \vee r)$, and C is $\neg q \wedge \neg r$, then $A[B \leftharpoonup C]$ is $\neg p \vee (\neg q \wedge \neg r)$. $A[p \leftharpoonup p \wedge q] = \neg(p \wedge q) \vee \neg(q \vee r)$ is an instance of A.

2.2 Semantics

In the previous section, we talked about Boolean values, truth tables, and meanings of logical operators, which are, strictly speaking, semantic concepts of the logic. Now, let us talk about meaning of propositional variables and formulas.

The informal meaning of a propositional variable is the statement that symbol represents. In different applications, the same symbol can represent different statements, thus has different meanings. However, each statement can have only

two truth values, i.e., true, or false. We will use these truth values of propositional variables, extending to the formulas which contain them, as the semantics of propositional logic.

2.2.1 Interpretations

A propositional variable may be assigned true or false. This truth value assignment is considered to be the *semantics* of the variable. For a formula, an assignment of truth values to every propositional variable is said be an *interpretation* of the formula. If A is a formula built on the propositional variables V_P, then an interpretation of A is the function $\sigma : V_P \longrightarrow \{1, 0\}$. It is easy to check that there are $2^{|V_P|}$ distinct interpretations.

Suppose σ is an interpretation over $V_P = \{p, q\}$ such that $\sigma(p) = 1$ and $\sigma(q) = 0$. We may write $\sigma = \{p \mapsto 1, q \mapsto 0\}$. We may use $(1, 0)$ for the same σ, assuming V_P is a list (p, q) and $\sigma(p, q) = (1, 0)$.

An alternative representation of an interpretation σ is to use a subset of V_P. Given σ, let $X_\sigma = \{x \in V_P \mid \sigma(x) = 1\}$. Then there is bijection between σ and X_σ. So we may use X_σ to represent σ. For example, if $V_P = \{p, q\}$, then the four interpretations over V_P is the power set of V_P:

$$2^{V_P} = \{\emptyset, \{p\}, \{q\}, \{p, q\}\}$$

Another alternative representation of σ is to use a set of literals. A *literal* is either a propositional variable or the negation of a variable. Let

$$Y_\sigma = X_\sigma \cup \{\neg y \in V_P, y \notin X_\sigma\}$$

then Y_σ is a set of literals such that every variable of V_P appears in Y_σ exactly once. If $V_P = \{p, q\}$ and $X_\sigma = \emptyset$, then $Y_\sigma = \{\neg p, \neg q\}$; if $X_\sigma = \{p\}$, $Y_\sigma = \{p, \neg q\}$, etc. That is, by adding the negations of the missing variables in X_σ, we obtain such a representation Y_σ for σ.

Given σ, we can check the truth value of the formula under σ, denoted by $\sigma(A)$. This notation means we treat σ as a function from the formulas to $\{0, 1\}$, as the unique extension of $\sigma : V_P \longrightarrow \{0, 1\}$. On the other hand, the notation $A\sigma$ is used to denote the formula obtained by substituting each propositional variable p by $\sigma(p)$.

Example 2.2.1. Given the formula $A = p \wedge \neg q \vee r$, and an interpretation $\sigma = \{p \mapsto 1, q \mapsto 1, r \mapsto 0\}$, we represent σ by $\sigma = \{p, q, \neg r\}$. Replace p by 1, q by 1, and r by 0 in A, we have $A\sigma = 1 \wedge \neg 1 \vee 0$. Applying the definitions of $\neg$, $\wedge$, and $\vee$, we

obtain the truth value 0. In this case, we say the formula is *evaluated* to 0 under σ, denoted by $\sigma(A) = 0$. Given another interpretation $\sigma' = \{p, q, r\}$, the same formula will be evaluated to 1, i.e., $\sigma'(A) = 1$. □

Recall that we use $A[B \leftharpoonup C]$ to denote an instance of A where every occurrence of B is replaced by C. An interpretation σ is a special case of the substitution of formulas where B is a variable and C is either 1 or 0. For example, if $\sigma = \{p, \neg q\}$, then $A\sigma = A[p \leftharpoonup 1][q \leftharpoonup 0]$. Strictly speaking, $A\sigma$ is not a propositional formula: it is the meaning of A under σ. To obtain $\sigma(A)$, we may use the algorithm *eval*.

Algorithm 2.2.2. The algorithm *eval* will take a formula A and an interpretation σ, and returns a Boolean value.

proc *eval*(A, σ)
    if $A = \top$ **return** 1;
    if $A = \bot$ **return** 0;
    if $A \in V_P$ **return** $\sigma(A)$;
    if A is $\neg B$ **return** $\neg$*eval*(B, σ);
    else A is $(B \ op \ C)$ **return** *eval*(B, σ) *op* *eval*(C, σ);

Example 2.2.3. Let $\sigma = \{p, \neg q\}$. Then the execution of *eval*$(p \to p \wedge \neg q, \sigma)$ will return 1:

eval$(p \to p \wedge \neg q, \sigma)$ calls
    eval(p, σ), which returns 1; and
    eval$(p \wedge \neg q, \sigma)$, which calls
        eval(p, σ), which returns 1; and
        eval$(\neg q, \sigma)$, which calls
            eval(q, σ), which returns 0; and
            returns $\neg 0$, i.e., 1;
        returns $1 \wedge 1$, i.e., 1;
    returns $1 \to 1$, i.e., 1.

□

What *eval* does is to travel the formula tree of A bottom-up, if the node is labeled with a variable, use σ to get the truth value; otherwise, compute the *truth value of this node* under σ using the operator at that node with the truth values from its children nodes. The process of running *eval* is exactly what we do when constructing a truth table for $p \to p \wedge \neg q$. The truth values under p and q in the

truth table give us all the interpretations σ and the truth values of $A = \neg q$, $p \wedge \neg q$, or $p \rightarrow p \wedge \neg q$ are the values of $eval(A, \sigma)$.

p	q	$\neg q$	$p \wedge \neg q$	$p \rightarrow p \wedge \neg q$
0	0	1	0	1
0	1	0	0	1
1	0	1	1	1
1	1	0	0	0

Theorem 2.2.4. *Algorithm* eval(A, σ) *returns* 1 *iff* $\sigma(A) = 1$*, and runs in* $O(|A|)$ *time, where* $|A|$ *denotes the number of symbols, excluding parentheses, in* A*.*

$|A|$ is also the number of nodes in the formula tree of A. The proof is left as an exercise.

Corollary 2.2.5. *For any formulas* A *and* B*, and any interpretation* σ*, the following equations hold.*

$$\begin{aligned} \sigma(A \vee B) &= \sigma(A) \vee \sigma(B) \\ \sigma(A \wedge B) &= \sigma(A) \wedge \sigma(B) \\ \sigma(\neg A) &= \neg\sigma(A) \end{aligned}$$

The truth value of a formula in propositional logic reflects the two foundational principles of Boolean logics: the *principle of bi-valence*, which allows only two truth values, and the *principle of extensibility* that the truth value of a general formula depends only on the truth values of its parts, not on their informal meaning.

2.2.2 Models, Satisfiability, and Validity

Interpretations play a very important role in propositional logic and introduce many important concepts.

Definition 2.2.6. Given a formula A and an interpretation σ, σ is said to be a model of A if $eval(A, \sigma) = 1$. If A has a model, A is said to be *satisfiable*.

Given a set V_P of propositional variables, we will use IV_P to denote the set of all interpretation over V_P. For example, if $V_P = \{p, q\}$, then $IV_P = \{\{p \mapsto 1, q \mapsto 1\}, \{p \mapsto 1, q \mapsto 0\}, \{p \mapsto 0, q \mapsto 1)\}, \{p \mapsto 0, q \mapsto 0\}$, or abbreviated as $IV_P = \{(1,1), (1,0), (0,1), (0,0)\}$ if p and q are understood from the context. In general, $|IV_P| = 2^{|V_P|}$, as each variable has two values. We may use *eval* to look for a model of A by examining every interpretation in IV_P.

Definition 2.2.7. *Given a formula* A*, let* $\mathcal{M}(A)$ *be the set of all models of* A*. If* $\mathcal{M}(A) = \emptyset$*,* A *is said to be* unsatisfiable*; If* $\mathcal{M}(A) = IV_P$*, i.e., every interpretation*

is a model of A, then A is said to be valid*, or* tautology*. We write* $\models A$ *if A is valid.*

Being a tautology is different from being useful or efficient. For instance, the statement $p \vee \neg p$ is true but unlikely to be very useful; it is neither efficient (the formula $\top$ is more precise).

If there is a truth table for a formula A, $\mathcal{M}(A)$ can be easily obtained from the truth table, as the truth values under the variables of A are all the interpretations and the models of A are those rows in the table where the truth value of A is 1.

Example 2.2.8. Let $V_P = \{p, q\}$. If A is $\neg p \to \neg q$, then $\mathcal{M}(A) = \{(1,1), (1,0), (0,0)\}$. If B is $p \vee (\neg q \to \neg p)$, then $\mathcal{M}(B) = IV_P$. From $\mathcal{M}(A)$ and $\mathcal{M}(B)$, we know that both A and B are satisfiable, and B is also valid, i.e., $\models B$ is true. □

A valid formula is one which is always true in every interpretation, no matter what truth values its variables may take, that is, every interpretation is its model. The simplest example is $\top$, or $p \vee \neg p$. There are many formulas that we want to know if they are valid, and this is done by so-called *theorem proving*, either by hand or automatically.

We may think about valid formulas as capturing fundamental logical truths. For example, the transitivity property of implication states that if one statement implies a second one, and the second one implies a third, then the first implies the third. In logic, the transitivity can be expressed as the formula

$$(p \to q) \wedge (q \to r) \to (p \to r)$$

The validity of the above formula confirms the truth of this property of implication. There are many properties like this for other logical operators, such as the commutativity and associativity of $\wedge$ and $\vee$, and they all can be stated by a tautology.

Theorem 2.2.9. (Substitution) *Let A be a tautology, so is any instance of A. That is, if p is a propositional variable in A and B is any formula, then* $A[p \leftarrow B]$ *is a tautology.*

Proof. For any interpretation σ of $A[p \leftarrow B]$, let $\sigma(p) = \sigma(B)$. Applying σ to the formula trees of A and $A[p \leftarrow B]$, then the truth values of all the nodes of A must be identical to those of the corresponding nodes of $A[p \leftarrow B]$. Since A is a tautology, the root node of A must be 1 under σ, so the root node of $A[p \leftarrow B]$ must the same truth value, i.e., 1. Since σ is arbitrary, $A[p \leftarrow B]$ must be a tautology. □

The significance of the above theorem is that if we have a tautology for one variable, the tautology holds when the variable is substituted by any formula. For

example, from $p \vee \neg p$, we have $A \vee \neg A$ for any A. On the other hand, when we try to prove a tautology involving symbols A, B, ..., we may treat each of these symbols as a propositional variable. For example, to prove $(A \wedge B) \leftrightarrow (B \wedge A)$, we may prove instead $(p \wedge q) \leftrightarrow (q \wedge p)$.

An unsatisfiable formula is one which does not have any model, that is, no interpretation is its model. The simplest example is 0, or $p \wedge \neg p$. Validity and unsatisfiability are dual concepts, as stated by the following proposition.

Proposition 2.2.10. *A formula A is valid iff $\neg A$ is unsatisfiable.*

From the view of theorem proving, proving the validity of a formula is as hard (or as easy) as proving its unsatisfiability.

The following theorem shows the relationship between logic and set theory.

Theorem 2.2.11. *For any formulas A and B,*

$$\begin{array}{lrcl} (a) & \mathcal{M}(A \vee B) & = & \mathcal{M}(A) \cup \mathcal{M}(B); \\ (b) & \mathcal{M}(A \wedge B) & = & \mathcal{M}(A) \cap \mathcal{M}(B); \\ (c) & \mathcal{M}(\neg A) & = & IV_P - \mathcal{M}(A). \end{array}$$

Proof. To show $\mathcal{M}(A \vee B) = \mathcal{M}(A) \cup \mathcal{M}(B)$, we need to show that, for any interpretation σ, $\sigma \in \mathcal{M}(A \vee B)$ iff $\sigma \in \mathcal{M}(A) \cup \mathcal{M}(B)$. By definition, $\sigma \in \mathcal{M}(A \vee B)$ iff $\sigma(A \vee B) = 1$. By Corollary 2.2.5, $\sigma(A \vee B) = \sigma(A) \vee \sigma(B)$. So $\sigma(A \vee B) = 1$ iff $\sigma(A) = 1$ or $\sigma(B) = 1$. $\sigma(A) = 1$ or $\sigma(B) = 1$ iff $\sigma(A) \vee \sigma(B) = 1$. $\sigma(A) \vee \sigma(B) = 1$ iff $\sigma \in \mathcal{M}(A) \cup \mathcal{M}(B)$.

The proof of the other two is left as exercise. □

Example 2.2.12. Let $V_P = \{p, q\}$. $\mathcal{M}(p) = \{(1,1),(1,0)\}$, $\mathcal{M}(q) = \{(1,1),(0,1)\}$; $\mathcal{M}(\neg p) = IV_P - \mathcal{M}(p) = \{(0,1),(0,0)\}$, and $\mathcal{M}(\neg p \vee q) = \mathcal{M}(\neg p) \cup \mathcal{M}(q) = \{(1,1),(0,1),(0,0)\}$. □

2.2.3 Equivalence

In natural language, one statement can be stated in different forms with the same meaning. For example, "if I won the lottery, I must be rich now." This meaning can be also expressed "I am not rich, because I didn't win the lottery." Introducing p for "I won the lottery" and q for "I'm rich", the first statement is $p \rightarrow q$ and the second statement is $\neg q \rightarrow \neg p$. It happens that $\mathcal{M}(p \rightarrow q) = \mathcal{M}(\neg q \rightarrow \neg p)$, as Table 2.2.1 shows that both formulas have the same set of models. Each line of the truth table defines an interpretation; if a formula is true in the same line, that

p	q	$\neg p$	$\neg q$	$p \to q$	$\neg p \vee q$	$\neg q \to \neg p$	$q \to p$
1	1	0	0	1	1	1	1
1	0	0	1	0	0	0	1
0	1	1	0	1	1	1	0
0	0	1	1	1	1	1	1

Table 2.2.1: The equivalence of the implication $p \to q$ and its contrapositive $\neg q \to \neg p$.

interpretation is a model. The table also reveals that $\mathcal{M}(p \to q) = \mathcal{M}(\neg p \vee p)$. In logic, these formulas are equivalent.

The formula $\neg q \to \neg p$ is called the *contrapositive* of the implication $p \to q$. The truth table shows that an implication and its contrapositive are equivalent: they are just different ways of saying the same thing. In contrast, the *converse* of $p \to q$ is the formula $q \to p$, which has a different set of models (as shown by the second and third lines in Table 2.2.1).

Definition 2.2.13. *Given two formulas A and B, A and B are said to be* logically equivalent *if $\mathcal{M}(A) = \mathcal{M}(B)$, denoted by $A \equiv B$.*

$A \equiv B$ means that, for every interpretation σ, $\sigma(A) = \sigma(B)$, so $\sigma \in \mathcal{M}(A)$ iff $\sigma \in \mathcal{M}(B)$. The relation $\equiv$ over formulas is an equivalence relation as $\equiv$ is reflexive, symmetric, and transitive. The relation $\equiv$ is also a congruence relation as $A \equiv C$ and $B \equiv D$ imply that $\neg A \equiv \neg C$ and $A \; o \; B \equiv C \; o \; D$ for any binary operator o. This property allows us to obtain an equivalent formula by replacing a part of the formula by an equivalent one. The relation $\equiv$ plays a very important role in logic as it is used to simplify formulas, to convert formulas into equivalent normal forms, or to provide alternative definitions for logical operators.

In arithmetic one writes simply $s = t$ to express that the terms s, t represent the same function. For example, $(x + y)^2 = x^2 + 2xy + y^2$ expresses the equality of values of the left-and right-hand terms for all x, y. In propositional logic, however, we use the equality sign like $A = B$ only for the syntactic identity of the formulas A and B. Therefore, the equivalence of formulas must be denoted differently, such as $A \equiv B$.

Theorem 2.2.14. *For any formulas A, B, and C, where B is a subformula of A, and $B \equiv C$, then $A \equiv A[B \leftharpoonup C]$.*

Proof. For any interpretation σ, $\sigma(B) = \sigma(C)$, since $B \equiv C$. Apply σ to the formula trees of A and $A[B \leftharpoonup C]$ and compare the truth values of all corresponding nodes

of the two trees, ignoring the proper subtrees of B and C. Since $\sigma(B) = \sigma(C)$, they must have the same truth values, that is, $\sigma(A) = \sigma(A[B \leftharpoonup C])$. □

The equivalence relation is widely used to simplify formulas and has real practical importance in computer science. Formula simplification in software can make a program easier to read and understand. Simplified programs may also run faster, since they require fewer operators. In hardware, simplifying formulas can decrease the number of logic gates on a chip because digital circuits can be described by logical formulas. Minimizing the logical formulas corresponds to reducing the number of gates in the circuit. The payoff of gate minimization is potentially enormous: a chip with fewer gates is smaller, consumes less power, has a lower defect rate, and is cheaper to manufacture.

Suppose a formula A contains k propositional variable, then A can be reviewed as one of Boolean function $f : \{1,0\}^k \longrightarrow \{1,0\}$. For example, $\neg p \vee q$ contains two variables and can be regarded as a Boolean function $f(p,q)$. The truth table (Table 2.2.1) reveals that $f(p,q)$, i.e., $\neg p \vee q$, always has the same truth value as $p \rightarrow q$, so f and $\rightarrow$ are the same function. As a result, we may use $\neg p \vee q$ as the definition of $\rightarrow$. As another example, the if-then-else function, $ite : \{1,0\}^3 \longrightarrow \{1,0\}$, can be defined by $ite(1,B,C) = B$ and $ite(0,B,C) = C$, instead of using a truth table of eight lines.

The following proposition lists many equivalent pairs of the formulas.

Proposition 2.2.15. *Assume A, B, and C are any formulas.*

$A \vee A \equiv A$;	$A \wedge A \equiv A$;
$A \vee B \equiv B \vee A$;	$A \wedge B \equiv B \wedge A$;
$(A \vee B) \vee C \equiv A \vee (B \vee C)$;	$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$;
$A \vee \bot \equiv A$;	$A \wedge \top \equiv A$;
$A \vee \neg A \equiv \top$;	$A \wedge \neg A \equiv \bot$;
$A \vee \top \equiv \top$;	$A \wedge \bot \equiv \bot$;
$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$;	$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$;
$\neg\top \equiv \bot$; $\quad \neg\bot \equiv \top$;	$\neg\neg A \equiv A$;
$\neg(A \vee B) \equiv \neg A \wedge \neg B$;	$\neg(A \wedge B) \equiv \neg A \vee \neg B$;
$A \rightarrow B \equiv \neg A \vee B$;	$A \rightarrow B \equiv \neg B \rightarrow \neg A$;
$A \oplus B \equiv (A \vee B) \wedge (\neg A \vee \neg B)$;	$A \leftrightarrow B \equiv (A \vee \neg B) \wedge (\neg A \vee B)$.

One way to show that two formulas are equivalent if we can prove the validity of a formula, as given by the following proposition:

Proposition 2.2.16. *For any formulas A and B, $A \equiv B$ iff $\models A \leftrightarrow B$; $\models A$ iff $A \equiv \top$.*

Proof. $A \equiv B$ means, for any interpretation σ, $\sigma(A) = \sigma(B)$, which means $\sigma(A \leftrightarrow B) = 1$, i.e., any σ is a model of $A \leftrightarrow B$. The proof of the second part is omitted. □

The above proposition shows the relationship between $\equiv$, which is a semantic notation, and $\leftrightarrow$, which is a syntactical symbol. Moreover, Theorem 2.2.9 allows us to prove the equivalences using the truth table method, treating A, B, and C as propositional variables. For example, from $(p \to q) \equiv (\neg p \vee q)$, we know $(p \to q) \leftrightarrow (\neg p \vee q)$ is tautology; from $(p \to q) \leftrightarrow (\neg p \vee q)$, we know $(A \to B) \leftrightarrow (\neg A \vee B)$ is a tautology for any formulas A and B, thus $(A \to B) \equiv (\neg A \vee B)$.

2.2.4 Entailment

In natural languages, given a set of premises, we like to know what conclusions can be drawn from the premises. If we represent these premises by a single formula, it will be the conjunction of the premises. Since $\wedge$ is commutative and associative, a conjunction of formulas can be conveniently written as a set S of formulas. That is, we assume that $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ is equivalent to $S = \{A_1, A_2, \cdots, A_n\}$. For example, $\{A, B\} \equiv A \wedge B$.

To catch the relation between the premises and the conclusion in logic, we have the notion of *entailment*.

Definition 2.2.17. *Given two formulas A and B, we say A* entails *B, or B is a* logical consequence *of A, denoted by $A \models B$, if $\mathcal{M}(A) \subseteq \mathcal{M}(B)$.*

Thus, $\models A$ and $\top \models A$ have the same meaning, i.e., A is valid.

The above definition allows many irrelevant formulas to be logical consequences of A, including all tautologies and logically equivalent formulas. Despite of this irrelevant relationship between A and B, the concept of entailment is indispensable in logic. For instance, an inference rule is a pattern which takes formulas as premises, and returns a formula as conclusion. To check the soundness of the inference rule, we let the premises be represented by formula A and the conclusion represented by B, and check if $A \models B$, i.e., B is a logical consequence of A. If the number of variables is small, we may use the truth table method to check that if every model of A remains to be a model for B.

Example 2.2.18. The modus ponens rule says that given the premises $p \to q$ and p, then we draw the conclusion q. Let A be $(p \to q) \wedge p$, then $\mathcal{M}(A) = \{(1,1)\}$, and $\mathcal{M}(q) = \{(1,1),(0,1)\}$. Since $\mathcal{M}(A) \subseteq \mathcal{M}(q)$, so q is a logical consequence of A, or $A \models B$. □

Proposition 2.2.19. *The relation $\models$ is transitive, that is, if $A \models B$ and $B \models C$, then $A \models C$.*

The proof is left as an exercise.

Definition 2.2.20. *Given a formula A, the set $\{B \mid A \models B\}$ is called the* theory *of A and is denoted by $\mathcal{T}(A)$. Every member of $\mathcal{T}(A)$ is called a theorem of A.*

In the above definition, A can be a set of formulas to represent the conjunction of its members.

Proposition 2.2.21. *The following three statements are equivalent: (a) $A \models B$; (b) $\mathcal{M}(A) \subseteq \mathcal{M}(B)$; and (c) $\mathcal{T}(B) \subseteq \mathcal{T}(A)$.*

Proof. By definition, $A \models B$ iff $\mathcal{M}(A) \subseteq \mathcal{M}(B)$. Also by definition, $B \models C$ for any $C \in \mathcal{T}(B)$. If $A \models B$, then $A \models C$ because of the transitivity of $\models$, so $C \in \mathcal{T}(A)$. If $\mathcal{T}(B) \subseteq \mathcal{T}(A)$, since $B \in \mathcal{T}(B)$, so $B \in \mathcal{T}(A)$, thus $A \models B$. □

Corollary 2.2.22. *For any formula A, $\mathcal{T}(\top) \subseteq \mathcal{T}(A) \subseteq \mathcal{T}(\bot)$.*

Proof. This holds because $\bot \models A$ and $A \models \top$. Note that $\mathcal{T}(\top)$ contains every tautologies and $\mathcal{T}(\bot)$ contains every formula. □

Corollary 2.2.23. *For any formulas A and B, the following statements are equivalent: (a) $A \equiv B$; (b) $\mathcal{M}(A) = \mathcal{M}(B)$; and (c) $\mathcal{T}(A) = \mathcal{T}(B)$.*

Proof. Since $A \equiv B$ iff $A \models B$ and $B \models A$, this corollary holds by applying Proposition 2.2.21 twice. □

Intuitively, we may regard a formula A as a constraint to mark some interpretations as "no-model" and $\mathcal{M}(A)$ records those interpretations which are models of A. The stronger the constraint, the less the set of models. The strongest constraint is $\bot$ (no interpretations left) and the weakest constraint is $\top$ (no constraints). On the other hand, $\mathcal{T}(A)$ collects all constraints which do not mark any model in $\mathcal{M}(A)$ as "no-model".

Proposition 2.2.24. *Let A and B be sets of formulas. If $A \subseteq B \subseteq \mathcal{T}(A)$, then $\mathcal{M}(A) = \mathcal{M}(B)$.*

Proof. Let $C = B - A$. Since $B \subseteq \mathcal{T}(A)$, so $C \subseteq \mathcal{T}(A)$, or $A \models C$, which implies $\mathcal{M}(A) \subseteq \mathcal{M}(C)$. Thus $\mathcal{M}(B) = \mathcal{M}(A \cup C) = \mathcal{M}(A \wedge C) = \mathcal{M}(A) \cap \mathcal{M}(C) = \mathcal{M}(A)$, because $\mathcal{M}(A) \subseteq \mathcal{M}(C)$. □

Corollary 2.2.25. *If $C \in \mathcal{T}(A)$, then $\mathcal{T}(A) = \mathcal{T}(A \wedge C)$.*

Proof. Let $B = A \cup \{C\}$, then $A \subseteq B \subseteq \mathcal{T}(A)$, so $\mathcal{M}(A) = \mathcal{M}(B) = \mathcal{M}(A \wedge C)$. Hence $\mathcal{T}(A) = \mathcal{T}(A \wedge C)$ by Corollary 2.2.23. □

Corollary 2.2.26. *If $\bot \in \mathcal{T}(A)$, then A is unsatisfiable.*

Proof. Let $B = A \cup \{\bot\}$, then $A \subseteq B \subseteq \mathcal{T}(A)$, so $\mathcal{M}(A) = \mathcal{M}(B) = \mathcal{M}(A \wedge \bot) = \mathcal{M}(\bot) = \emptyset$. □

One task of theorem proving is to show that, given a set A of axioms, B is a theorem of A, that is, $B \in \mathcal{T}(A)$. If we use the operator $\rightarrow$, to show that B is a logical consequence of A by showing $(A \rightarrow B) \in \mathcal{T}(\top)$, as given by the following proposition.

Proposition 2.2.27. *For any formulas A and B, $A \models B$ iff $\models A \rightarrow B$.*

Proof. $A \models B$ means, for any interpretation σ, if $\sigma(A) = \top$, then $\sigma(B)$ must be $\top$; if $\sigma(A) = \bot$, we do not care about the value of $\sigma(B)$. The two cases can be combined as $\sigma(A) \rightarrow \sigma(B) = 1$, or $\sigma(A \rightarrow B) = 1$, which means $A \rightarrow B$ is valid, because σ is any interpretation. □

The above proposition shows the relationship between $\models$ (a semantic notation) and $\rightarrow$ (a syntactical symbol). The closure of the entailment relation under substitution generalizes the fact that from $p \wedge q \models p$, all entailment of the form $A \wedge B \models A$ arise from substituting A for p and B for q, just like the closure of the equivalence relation under substitution.

2.2.5 Theorem Proving and the SAT Problem

The concepts of validity, unsatisfiability, or entailment give rise difficult problems of computation. These three problems have the same degree of difficulty and can be represented by theorem proving, which asks to check if a formula $A \in \mathcal{T}(\top)$ (A is valid), or $\bot \in \mathcal{T}(A)$ (A is unsatisfiable), or A is in $\mathcal{T}(B)$ (B entails A). Theorem proving by truth table runs out of steam pretty quickly: a formula with n variables has a truth table of 2^n interpretations, so the effort grows exponentially with the number of variables. For a formula with just 30 variables, that is already over a billion interpretations to check!

The general problem of deciding whether a formula is A satisfiable is called the SAT problem. One approach to SAT is to construct a truth table and check whether or not a $\top$ ever appears, but as with testing validity, this approach quickly

bogs down for formulas with many variables because truth tables grow exponentially with the number of variables.

The good news is that SAT belongs to the class of NP problems, where the solutions of these problems can be efficiently checked. For example, given an interpretation σ, we can check in linear time (in terms of the size of A) if σ is a model of A, thus showing A is satisfiable. This can be done using $\mathit{eval}(A, \sigma)$). If a problem can be solved in polynomial time (with respect to the size of inputs), it belongs to the class P. SAT is not known to be in P, and theorem proving is not known to be in NP.

The bad news is that SAT is known to be the hardest problem in NP. That is, if there exists a polynomial-time algorithm to solve SAT, then every problem in NP can be solved in polynomial-time. These problems include many other important problems involving scheduling, routing, resource allocation, and circuit verification across multiple disciplines including programming, algebra, finance, and political theory. This is the famous result of Cook and Levin who proved that SAT is the first NP-complete problem. Does this mean that we cannot find an algorithm which takes $O(n^{100})$ time, where n is the size of A, to decide A is satisfiable? No one knows. This is the most famous open problem in computer science: $P = NP$? It is also one of the seven Millennium Problems: the Clay Institute will award you $1,000,000 if you solve the $P = NP$ problem. If $P = NP$, then SAT will be in P; theorem proving will be also in P. If $P \neq NP$, then theorem proving won't be in NP: it belongs the complement of the NP-complete problems. An awful lot hangs on the answer and the general consensus is $P \neq NP$.

In the last twenty years, there have been exciting progress on SAT-solvers for practical applications like digital circuit verification. These programs find satisfying assignments with amazing efficiency even for formulas with thousands of variables. In the next two chapters, we will study the methods for theorem proving and SAT-solvers.

2.3 Normal Forms

By "normal form", we mean every formula can be converted into an equivalent formula with certain syntactical restriction on the use and position of Boolean operators. If all equivalent formulas has a unique normal form, that normal form is also called "canonical form".

Since it is expensive to use truth table to prove the equivalence relation when the number of variables is big, an alternative approach is to use algebra to prove equivalence. A lot of different operators may appear in a propositional formula, so a useful first step is to get rid of all but three: $\wedge, \vee$ and $\neg$. This is easy because

each of the operators is equivalent to a simple formula using only these three. For example, $A \to B$ is equivalent to $\neg A \vee B$.

Using the equivalence relations discussed in the previous section, any propositional formula can be proved equivalent to a canonical form. What has this got to do with equivalence? That's easy: to prove that two formulas are equivalent, convert them both to canonical forms over the set of variables that appear in any formula. Now if two formulas are equivalent to the same canonical form, then the formula are certainly equivalent. Conversely, if two formulas are equivalent, they will have the same canonical form. We can also use canonical form to show a formula is valid or unsatisfiable: it is valid if its canonical form is $\top$; it is unsatisfiable if its canonical form is $\bot$.

In this section, we present four normal forms:

- negation normal form (NNF);
- conjunctive normal form (CNF);
- disjunctive normal form (DNF); and
- ITE normal form (INF).

Of the four normal forms, there exist canonical forms for the last three. However, the canonical forms derived from DNF or CNF are essentially copies of truth tables, thus not effective enough for showing the equivalence relation. Only the last one, INF, may have a compact size so that equivalence of two formulas can be conveniently established in this canonical form.

We will define a normal form by giving a set of transformation rules, that is a set of pairs of formulas, one is the left hand side and the other is the right hand side of the rule. Most importantly, the left side of each rule is logically equivalent to its right side. Each rule deals with a formula which is not in normal form and the application of the rule on the formula will result in a new formula closer to normal form. To convert any formula into a formula in normal form, we pick a rule, matching the left side of the rule to a subformula and replacing the subformula by the right side of the rule.

A general procedure for obtaining normal forms goes as follows:

1. If the formula is already in normal form then stop with success.
2. Otherwise it contains a subformula violating the normal form criteria.
3. Pick such a subformula and find the left hand side of a rule that matches.

4. Replace the subformula by the right hand side of the rule and continue.

2.3.1 Negation Normal Form (NNF)

A formula A is said to be a *literal* if A is a variable (positive literal) or A is $\neg p$ (negative literal), where p is a variable.

Definition 2.3.1. *A propositional formula A is a* negation norm form *(NNF) if the negation symbol $\neg$ appears in literals (as subformulas of A).*

In a NNF, if we replace negative literals by a variable, there will be no $\neg$ in the resulting formula.

Proposition 2.3.2. *Every propositional formula can be transformed into an equivalent NNF.*

The proof is done by the following algorithm, which uses only equivalence relations in the transformation.

Equivalence rules for obtaining NNF

Algorithm 2.3.3. The algorithm to convert a formula into NNF takes the following three groups of rules:

1. Use the following equivalences to remove $\rightarrow$, $\oplus$, and $\leftrightarrow$ from the formula.

$$\begin{aligned} A \rightarrow B &\equiv \neg A \vee B; \\ A \oplus B &\equiv (A \vee B) \wedge (\neg A \vee \neg B); \\ A \leftrightarrow B &\equiv (A \vee \neg B) \wedge (\neg A \vee B). \end{aligned}$$

2. Use the de Morgan laws to push $\neg$ down to variables:

$$\begin{aligned} \neg\neg A &\equiv A; \\ \neg(A \vee B) &\equiv \neg A \wedge \neg B; \\ \neg(A \wedge B) &\equiv \neg A \vee \neg B. \end{aligned}$$

3. Use $A \vee \bot \equiv A$, $A \vee \top \equiv \top$, $A \wedge \bot \equiv \bot$, and $A \wedge \top \equiv A$, to simplify the formula.

For example, $\top$ and $\neg p \vee q$ are NNF. $\neg(\neg p \vee q)$ is not NNF; we may transform it into $p \wedge \neg q$, which is NNF. If the input formula contains only $\neg$, $\vee$, $\wedge$, and $\rightarrow$, it takes only linear time to obtain NNF by working recursively top-down. If $\leftrightarrow$ or $\oplus$ are present, the computing time and the size of the output formula may be exponential in terms of the original size, because the arguments of $\leftrightarrow$ and $\oplus$ are duplicated during the transformation.

Definition 2.3.4. *Given a NNF A, if we replace every occurrence of $\wedge$ by $\vee$, $\vee$ by $\wedge$, positive literal by its negation, and negative literal $\neg p$ by its counterpart p, we obtain a formula B, which is called the* dual *of A, denoted by $B = dual(A)$.*

For example, if A is $p \wedge (q \vee (\neg p \wedge \neg r))$, then $dual(A) = \neg p \vee (\neg q \wedge (p \vee r))$. Both A and $dual(A)$ are NNF.

Proposition 2.3.5. *If A is in NNF, then $dual(A)$ is also in NNF and $dual(A) \equiv \neg A$.*

The proposition can be easily proved by DeMorgan's law.

2.3.2 Conjunctive Normal Form (CNF)

Definition 2.3.6. *In propositional logic, a* clause *is either $\perp$ (the empty clause), a literal (unit clause) or a disjunction of literals.*

A clause is also called *maxterm* in the community of circuit designs. Since $\vee$ is commutative and associative, we may represent a clause by a list of literals. We may use $A \vee A \equiv A$ to remove duplicate literals in a clause.

Definition 2.3.7. *A formula is said to be a* conjunctive normal form *(CNF) if it is a conjunction of clauses.*

A CNF is also called a *product of sums* (POS) expression. The idea of conjunctive normal form is that each type of connective appears at a distinct height in the formula. The negation signs, $\neg$, are lowest, with only propositions as their arguments. The disjunction signs, $\vee$, are in the middle, with only literals as their arguments. And the conjunction signs, $\wedge$, are at the top, taking the disjunctions as their arguments. Of course, some operators may be missing in CNF. For example, $p \wedge q$ is a CNF, which contains two unit clauses: p and q.

Equivalence rules for obtaining CNF

- stage 1: The equivalence rules for obtaining NNF.
- stage 2: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, where the arguments of $\wedge$ and $\vee$ can be switched.

Proposition 2.3.8. *Every propositional formula can be transformed into an equivalent CNF.*

To show the proposition is true, we first convert the formula into NNF, and then apply the distribution law to move $\vee$ under $\wedge$. This process will terminate because we have only a finite number of $\vee$ and $\wedge$.

Example 2.3.9. To convert $p_1 \leftrightarrow (p_2 \leftrightarrow p_3)$ into CNF, we first use $A \leftrightarrow B \equiv (\neg A \vee B) \wedge (A \vee \neg B)$ to remove the second $\leftrightarrow$:

$$p_1 \leftrightarrow ((\neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_3)),$$

and then remove the first $\leftrightarrow$:

$$(\neg p_1 \vee ((\neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_3))) \wedge (p_1 \vee \neg((\neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_3))).$$

Its NNF is

$$(\neg p_1 \vee ((\neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_3))) \wedge (p_1 \vee ((p_2 \wedge \neg p_3) \vee (\neg p_2 \wedge p_3))).$$

We then obtain the following non-tautology clauses:

$$\neg p_1 \vee \neg p_2 \vee p_3, \quad \neg p_1 \vee p_2 \vee \neg p_3, \quad p_1 \vee p_2 \vee p_3, \quad p_1 \vee \neg p_2 \vee \neg p_3.$$

□

Since $\vee$ is commutative and associative and $X \vee X \equiv X$, a clause can be represented by a set of literals. In this case, we use $\mid$ for $\vee$ and $\overline{p}$ for $\neg p$ in a clause. For example, the above clauses will be displayed as

$$(\overline{p_1} \mid \overline{p_2} \mid p_3), \quad (\overline{p_1} \mid p_2 \mid \overline{p_3}), \quad (p_1 \mid p_2 \mid p_3), \quad (p_1 \mid \overline{p_2} \mid \overline{p_3}).$$

Definition 2.3.10. *If a clause includes every variable exactly once, the clause is said to be* full. *A CNF is called a* full CNF *if every clause of the CNF is full.*

If a clause C misses a variable r, we may replace C by two clauses: $\{(C \mid r), (C \mid \overline{r})\}$. Repeating this process for every missing variable, we will obtain an equivalent full CNF. Transforming a set of clauses into an equivalent full CNF is sound because for any formula A and B, the following relation is true:

$$A \equiv (A \vee B) \wedge (A \vee \neg B),$$

which can be proved by truth table.

Equivalence rules for obtaining full CNF

- stage 1: The equivalence rules for NNF.
- stage 2: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, where the arguments of $\wedge$ and $\vee$ can be switched.
- stage 3: $A \equiv (A \vee p) \wedge (A \vee \neg p)$, if A is a clause and p does not appear in A.

Proposition 2.3.11. *Every propositional formula can be transformed into an equivalent and unique full CNF (up to the commutativity and associativity of $\wedge$ and $\vee$). That is, full CNF is a canonical form.*

Given a CNF A, each clause can be regarded as a constrain on what interpretations cannot be a model of A. For example, if A contains the unit clause $(\neg p)$, then all the interpretations where $p \mapsto 1$ cannot be a model of A and these are half of all interpretations. If A contains a binary clause $(p \mid q)$, then all interpretations where $p \mapsto 0, q \mapsto 0$ (there are a quarter of such interpretations) cannot be a model of A. If A has a clause C containing every variable exactly once, then this clause removes only interpretation, i.e., $\neg C$, as a model of A. If A has m models over V_P, i.e., $|\mathcal{M}(A)| = m$, then a full CNF of A contains $2^{|V_P|} - m$ clauses, as each clause removes exactly one interpretation as a model of A. That is exactly the idea for a proof of the above proposition.

The concept of full CNF is useful for theoretic proofs. Unfortunately, the size of a full CNF is exponential in terms of number of variables, too big for any practical application. CNFs are useful to specify many problems in practice, because practical problems are often specified by a set of constraints. Each constraint can be specified by a set of clauses, and the conjunction of all clauses from each constraint gives us a complete specification of the problem.

2.3.3 Disjunctive Normal Form (DNF)

If we regard $\wedge$ as multiplication, and $\vee$ as addition, then a CNF is a "product of sums. On the other hand, a "sum of products" is a disjunctive normal form (DNF). A formula A is said to be a *minterm* if A is either $\top$, a literal, or a conjunction of literals. Product is also called *product* or *product term* in the community of circuit designs.

Like clauses, we remove duplicates in a minterm and represent them by a set of literals.

Definition 2.3.12. *A formula A is a* disjunctive normal form *(DNF) if A is a disjunction of products.*

A DNF is also called a *sum of products* (SOP) expression. A formula like $p \wedge \neg q$ or $p \vee \neg q$ can be both a CNF and a DNF.

Equivalence rules for obtaining DNF

- stage 1: The equivalence rules for NNF.
- stage 2: $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$, where the arguments of $\wedge$ and $\vee$ can be switched.

Proposition 2.3.13. *Every propositional formula can be transformed into an equivalent DNF.*

To show the proposition is true, we first convert the formula into NNF in the first stage, and then apply the distribution law in the second stage to move $\wedge$ under $\vee$.

Once a formula A is transformed into DNF, it is very easy to check if A is satisfiable: If DNF contains a product in which each variable appears at most once, then A is satisfiable, because we can simply assign 1 to each literal of this product and the product becomes true. Since it is a hard problem to decide if a formula is satisfiable, it is also hard to obtain DNF, due to its large size.

Like CNF, DNF is not a canonical form. For example, the following equivalence is true:

$$p \wedge \neg q \vee \neg p \wedge \neg r \vee q \wedge r \equiv \neg p \wedge q \vee p \wedge r \vee \neg q \wedge \neg r$$

However, we cannot convert the both sides to the same formula.

Definition 2.3.14. *A product is* full *if it contains every variable exactly once. A DNF is* full *if every product in the DNF is full.*

A full minterm in a DNF A is equivalent to a model of A. For example, if $V_P = \{p, q, r\}$, then the product $p \wedge q \wedge \neg r$ specifies the model $\{p \mapsto 1, q \mapsto 1, r \mapsto 0\}$. Obviously, a full DNF A has m full products iff A has m models. Like full CNF, every propositional formula is equivalent to a unique full DNF, that is, full DNF is a canonical form.

Equivalence Rules for obtaining full DNF

- stage 1: The equivalence rules for NNF.
- stage 2: $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$, where the arguments of $\wedge$ and $\vee$ can be switched.

- stage 3: $A \equiv (A \wedge p) \vee (A \wedge \neg p)$, if A is a product and p does not appear in A.

Proposition 2.3.15. *Every propositional formula can be transformed into an equivalent and unique full DNF (up to the commutativity and associativity of $\wedge$ and $\vee$).*

2.3.4 Full DNF and Full CNF from Truth Table

A truth table is often used to define a Boolean function. In the truth table, the truth value of the function is given for every interpretation of the function. Each interpretation corresponds to a full minterm and the function can be defined as a sum (disjunction) of all the minterms for which the function is true in the corresponding interpretation.

Example 2.3.16. Suppose we want to define a Boolean function $f : B^3 \longrightarrow B$, where $B = \{0, 1\}$ and the truth value of $f(a, b, c)$ is given in the following truth table. Following the convention of EDA, we will use $+$ for $\vee$, $\cdot$ for $\wedge$, and $\overline{A}$ for $\neg A$.

a	b	c	f	minterms	clauses
0	0	0	0	$m_0 = \overline{a}\overline{b}\overline{c}$	$M_0 = a + b + c$
0	0	1	1	$m_1 = \overline{a}\overline{b}c$	$M_1 = a + b + \overline{c}$
0	1	0	0	$m_2 = \overline{a}b\overline{c}$	$M_2 = a + \overline{b} + c$
0	1	1	1	$m_3 = \overline{a}bc$	$M_3 = a + \overline{b} + \overline{c}$
1	0	0	0	$m_4 = a\overline{b}\overline{c}$	$M_4 = \overline{a} + b + c$
1	0	1	1	$m_5 = a\overline{b}c$	$M_5 = \overline{a} + b + \overline{c}$
1	1	0	0	$m_6 = ab\overline{c}$	$M_6 = \overline{a} + \overline{b} + c$
1	1	1	0	$m_7 = abc$	$M_7 = \overline{a} + \overline{b} + \overline{c}$

For each interpretation σ of (a, b, c), we define a minterm m_i, where i is the decimal value of σ, when σ is viewed as a binary number. For each m_i, we also define a clause M_i and the relation between m_i and M_i is that $\neg m_i \equiv M_i$. For the function f defined in the truth table, $f = m_1 + m_3 + m_5$, or $f = \overline{a}\overline{b}c + \overline{a}bc + a\overline{b}c$. □

In the above example, for each model of f, we create a minterm (product term) in its full DNF: If the variable is true in the model, the positive literal of this variable is in the product; if a variable is false, the negative literal of the variable is in the product. f is then defined by a full DNF consisting of these minterms. In other words, every Boolean function can be defined by a DNF. We can also define the same function by a full CNF. This can be processed as follows: At first, we define the negation of f by a full DNF from the truth table:

$$\overline{f} = m_0 + m_2 + m_4 + m_6 + m_7$$

Apply the negation on the both sides of the above equation, we have

$$f = M_0 \cdot M_2 \cdot M_4 \cdot M_6 \cdot M_7$$

where $\cdot$ stands for $\wedge$ and $\overline{m_i} \equiv M_i$. In other words, f can be defined by a full CNF. Note that $M_0 \cdot M_2 \cdot M_4 \cdot M_6 \cdot M_7$ is the dual of $m_0 + m_2 + m_4 + m_6 + m_7$. In general, the dual of DNF is CNF and the dual of CNF is DNF.

Once we know the principle, we can construct a full CNF directly from the truth table of a formula. That is, for each non-model interpretation in the truth table, we create one full clause: If a variable is true in the interpretation, the negative literal of this variable is in the clause; if a variable is false, the positive literal of the variable is in the clause.

Example 2.3.17. The truth table of $(p \vee q) \rightarrow \neg r$ has eight lines and three interpretations of $\langle p, q, r\rangle$ are non-models: $\langle 1, 1, 1\rangle$, $\langle 1, 0, 1\rangle$, and $\langle 0, 1, 1\rangle$. From these three interpretations, we obtain three full clauses:

$$(\neg p \vee \neg q \vee \neg r) \wedge (\neg p \vee q \vee \neg r) \wedge (p \vee \neg q \vee \neg r)$$

□

2.3.5 Binary Decision Diagram (BDD)

Let $ite : \{1, 0\}^3 \longrightarrow \{1, 0\}$ be the "if-then-else" operator such that, for any formula A and B, $ite(1, A, B) \equiv A$ and $ite(0, A, B) \equiv B$. In fact, ite, 1 and 0 can be used to represent any logical operator. For example, $\neg y \equiv ite(y, 0, 1)$. The following table shows that every binary logical operator can be represented by ite. Following the convention on BDD, we will use 1 for $\top$ and 0 for $\bot$ in every propositional formula.

Definition 2.3.18. *A formula is said be an* ite normal form *(INF) if the formula uses logical operators no more than ite,* 1*, and* 0*, and, for each occurrence of* $ite(C, A, B)$ *in the formula,* C *is a propositional variable, and* A *and* B *are INFs.*

By definition, 1 and 0 are INFs, but p is not; $ite(p, 1, 0)$ is.

Proposition 2.3.19. *Every propositional formula can be transformed into an equivalent INF.*

For example, let A be $\neg p \vee q \wedge \neg r$, its equivalent INF is $ite(p, ite(q, ite(r, 0, 1), 0), 1)$. To prove the above proposition, we provide the algorithm below to do the transformation.

output	**formula**	**ite formula**
0000	0	0
0001	$x \wedge y$	$ite(x, y, 0)$
0010	$x \wedge \neg y$ $(x > y)$	$ite(x, \neg y, 0)$
0011	x	$ite(x, 1, 0)$
0100	$\neg x \wedge y$ $(x < y)$	$ite(x, 0, y)$
0101	y	$ite(y, 1, 0)$
0110	$x \oplus y$	$ite(x, \neg y, y)$
0111	$x \vee y$	$ite(x, 1, y)$
1000	$x \downarrow y$	$ite(x, 0, \neg y)$
1001	$x \leftrightarrow y$	$ite(x, y, \neg y)$
1010	$\neg y$	$ite(y, 0, 1)$
1011	$x \vee \neg y$ $(x \geq y)$	$ite(x, 1, \neg y)$
1100	$\neg x$	$ite(x, 0, 1)$
1101	$\neg x \vee y$ $(x \leq y)$	$ite(x, y, 1)$
1110	$x \uparrow y$	$ite(x, \neg y, 1)$
1111	1	1

Table 2.3.1: Binary logical operator by *ite*. The first column shows the output of the function on the input pairs $(x, y) = (0, 0), (0, 1), (1, 0)$, and $(1, 1)$. The second column shows its conventional form and the last column gives the equivalent of ite form. In column 3, y stands for $ite(y, 1, 0)$ and $\neg y$ for $ite(y, 0, 1)$.

Algorithm 2.3.20. The algorithm to convert a formula into INF is a recursive procedure *convertINF*:

proc *convertINF*(A)
  if $A = 1$ **return** 1;
  if $A = 0$ **return** 0;
  else if A contains $p \in V_P$
    return $ite(p, convertINF(A[p \leftharpoonup 1]), convertINF(A[p \leftharpoonup 0]))$;
  else return *simplify*(A).

Note that $A[p \leftharpoonup 1]$ (or $A[p \leftharpoonup 0]$) stands for the formula resulting from replacing every occurrence of p by 1 (or 0) in A. The subroutine *simplify*(A) will return the truth value of formula A which has no propositional variables. The formula returned by *convertINF* contains only *ite*, 0, and 1 as the logical operators. All the propositional variables appear as the first argument of *ite*.

Example 2.3.21. let A be $\neg p \vee q \wedge \neg r$, then $A[p \leftharpoonup 1] = \neg 1 \vee q \wedge \neg r \equiv q \wedge \neg r$; $A[p \leftharpoonup 0] = \neg 0 \vee q \wedge \neg r \equiv 1$. So $convertINF(A) = ite(p,\ convertINF(q \wedge \neg r), 1)$. $convertINF(q \wedge \neg r) = ite(q,\ convertINF(\neg r), 0)$, and $convertINF(\neg r) = ite(r, 0, 1)$. Combining all, we have $convertINF(A) = ite(p, ite(q, ite(r, 0, 1), 0), 1)$. □

A *binary decision diagram* (BDD) is a directed acyclic graph (DAG) G, which represents an INF by maximally sharing common subformulas. G has exactly two leaf nodes (no outgoing edges) with labels 1 and 0, respectively, and a set of internal nodes (with out-going edges); each internal node represents INF $ite(p, A, B)$, that is, the node is labeled with p and has the two out-going edges to the nodes representing A and B, respectively.

Given an order on the propositional variables, if we use this order to choose each variable in the procedure *convertINF*, the resulting INF is an *ordered BDD* (OBDD). For the above example, the order $p > q > r$ is used. If we use $r > q > p$, then for $A = \neg p \vee q \wedge \neg r$, $convertINF(A) = ite(r, ite(p, 0, 1), ite(q, 1, ite(p, 1, 0)))$. The choice of orders has a big impact on the sizes of OBDD, as one order gives you the linear size (in terms of number of variables); another order may give you an exponential size. The example in Figure 2.3.2 shows the two OBDDs for $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$ using two orders. If we generalize this example to $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee ... \vee (a_n \wedge b_n)$, the size of the first OBDD will be $O(n)$, while the second OBDD will have an exponential size.

Since $ite(C, A, A) \equiv A$, we may use this equivalence repeatedly to simplify an INF, until no more simplification can be done. The resulting INF is called *reduced*. The corresponding BDD is called *reduced* if, additionally, identical subformulas of an INF are represented by a unique node.

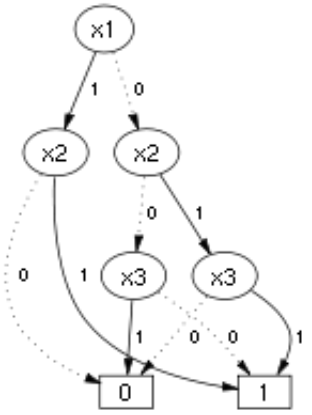


Figure 2.3.1: The BDD of INF $ite(x_1, ite(x_2, 1, 0), ite(x_2, ite(x_3, 1, 0), ite(x_3, 0, 1)))$ derived from $\overline{x_1x_2x_3} \vee x_1x_2 \vee x_2x_3$.

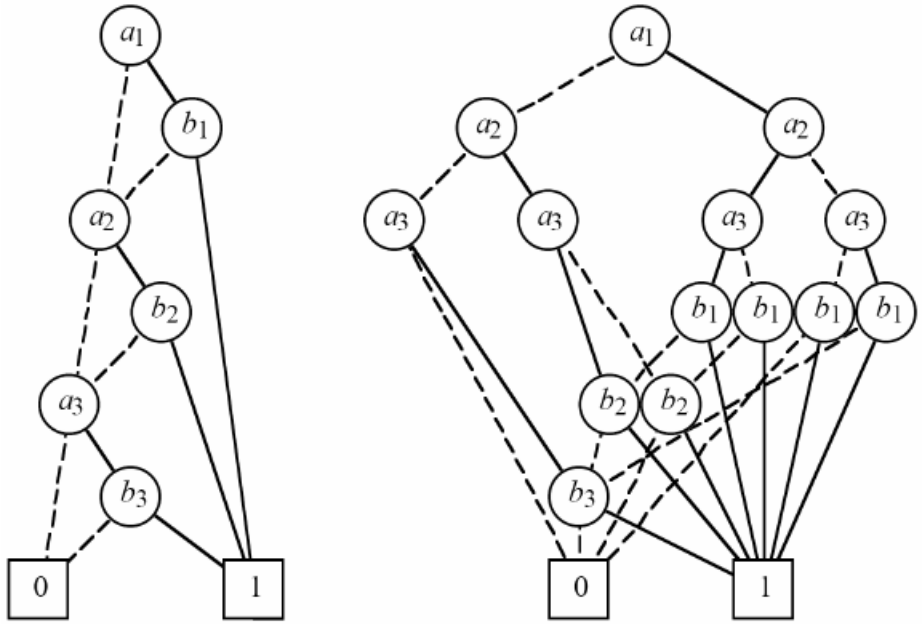


Figure 2.3.2: The first BDD uses $a_1 > b_1 > a_2 > b_2 > a_3 > b_3$ and the second BDD uses $a_1 > a_2 > a_3 > b_1 > b_2 > b_3$ for the same formula $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$.

Proposition 2.3.22. *All logical equivalent INFs, using the same order on propositional variables, have a unique reduced OBDD (ROBDD).*

Proof. Since a ROBDD is a graph representing an INF A, we may do an induction proof on the number of variables in A. If A contains no variables, then 1 or 0 are represented by the unique nodes. Let $p_n > p_{n-1} > \cdots > p_2 > p_1$ be the order on the n variables in A. Then $A \equiv ite(p_n, B, C)$ for INFs B, C, and $B \not\equiv C$. If $n = 1$, then $\{B, C\} = \{0, 1\}$, and A is represented by a unique node. If $n > 1$, because B and C are INFs without variable p_n, by induction hypothesis, B and C are represented by the unique ROBDDs, respectively. Thus, $A = ite(p_n, B, C)$ has a unique ROBDD representation. □

It is easy to apply $ite(p, A, A) \equiv A$ as a reduction rule on INF. To implement the sharing of common subformulas, we may use a hash table which contains the triple $\langle v, x, y\rangle$ for each node $ite(v, x, y)$ in a ROBDD, where v is a propositional variable and both x and y, $x \neq y$, are nodes in the ROBDD. The hash table provides two operations: *lookupHashTable*(v, x, y) checks if $\langle v, x, y\rangle$ exists in the hash table: if yes, the node is returned; otherwise, *null* is returned. The other operation is *saveCreateNode*(v, x, y), which creates a new node for the triple $\langle v, x, y\rangle$ and saves the node in the hash table. The pseudo-code of the algorithm for creating ROBDD is given below.

Algorithm 2.3.23. The algorithm *ROBDD* takes a formula A and returns a ROBDD node for A. $vars(A)$ returns the set of propositional variables of A, *topVariable*(S) returns the maximal variable of S by the given variable order. *simplify*(A) returns the truth value of a formula without variables.

proc *ROBDD*(A)
  if $vars(A) = \emptyset$ **return** *simplify*(A);
  $v :=$ *topVariable*$(vars(A))$;
  $x :=$ *ROBDD*$(A[v \leftarrow 1])$;
  $y :=$ *ROBDD*$(A[v \leftarrow 0])$;
  if $(x = y)$ **return** x; // reduction
  $p :=$ *lookupHashTable*(v, x, y);
  if $(p \neq null)$ **return** p; // sharing
  return *saveCreateNode*(v, x, y); // a new node is created

The input A to *ROBDD* can contain the nodes of a ROBDD, so that we can perform various logical operations on ROBDDs.

Example 2.3.24. Assume $a > b > c$, and let A be $ite(F, G, H)$, where $F = ite(a, 1, b)$, $G = ite(a, c, 0)$, and $H = ite(b, 1, d)$. Then $ROBDD(A)$ will call $ROBDD(A[a \leftarrow 1])$ and $ROBDD(A[a \leftarrow 0])$, which returns c and $J = ite(b, 0, H)$, respectively. Finally, $ROBDD(A)$ returns $ite(a, c, J)$. □

ROBDDs have used for presenting Boolean functions, symbolic simulation of combinational circuits, equivalence checking and verification of Boolean functions, and finding and counting models of propositional formulas.

2.4 Optimization Problems

There exist many optimization problems in propositional logic. For example, one of them is to find a variable order for a formula, so that the resulting ROBDD is minimal. In this section, we introduce three optimization problems.

2.4.1 Minimum Set of Operators

To represent the sentences like A provided B, which represent the converse implication, we do not need a formula like $A \Leftarrow B$, because we may use simply $B \rightarrow A$. This and similar reasons explain why only a few of the sixteen binary Boolean functions require notation. In CNF or DNF, we need only three Boolean operators: $\neg$, $\vee$, and $\wedge$; all other Boolean operators can be defined in terms of them.

Definition 2.4.1. *A set S of Boolean operators is said to be* sufficient *if every other Boolean operators can be defined using only the operators in S. S is said to be* minimally sufficient *if S is sufficient and no proper subset of S is a sufficient set of operators.*

Obviously, $\{\neg, \vee, \wedge\}$ is a sufficient set of operators. For example, $A \oplus B \equiv (A \vee \neg B) \wedge (\neg A \vee B)$ and $ite(A, B, C) \equiv (A \wedge B) \vee (\neg A \wedge C)$. Is the set $\{\neg, \vee, \wedge\}$ minimally sufficient? The answer is no, because $A \vee B \equiv \neg(\neg A \wedge \neg B)$. Thus, $\{\neg, \wedge\}$ is a sufficient set of operators. $\{\neg, \vee\}$ is another sufficient set, because $A \wedge B \equiv \neg(\neg A \wedge \neg B)$. To show that $\{\neg, \wedge\}$ is minimally sufficient, we need to show that neither $\{\neg\}$ nor $\{\wedge\}$ is sufficient. Since $\neg$ takes only one argument and cannot be used alone to define $\wedge$, $\{\neg\}$ cannot be sufficient. We will see later that $\{\wedge\}$ is not sufficient.

From IND (ITE normal form), we might think $\{ite\}$ is minimally sufficient. That is not true, because we also use 1 and 0 (stand for $\top$ and $\bot$) which are nullary operators. For the set $\{\neg, \wedge\}$, $\bot \equiv p \wedge \neg p$ for any propositional variable p.

Do we have a minimally sufficient set of Boolean operators which contains a single operator? The answer is yes and there are two binary operators as candidates. They are $\uparrow$ (nand, the negation of and) and $\downarrow$ (nor, the negation of or):

$$A \uparrow B \equiv \neg(A \wedge B) \qquad\qquad A \downarrow B \equiv \neg(A \vee B)$$

Proposition 2.4.2. *Both $\{\uparrow\}$ and $\{\downarrow\}$ are minimally sufficient.*

Proof. To show $\{\uparrow\}$ is sufficient, we just need to define $\neg$ and $\wedge$ using $\uparrow$: $\neg(A) \equiv A \uparrow A$ and $A \wedge B \equiv \neg(A \uparrow B)$. The proof that $\{\downarrow\}$ is sufficient is left as an exercise. □

In fact, among all the sixteen binary operators, only $\uparrow$ and $\downarrow$ have such property.

Theorem 2.4.3. *For any binary operator which can be used to define both $\neg$ and $\wedge$, that operator must be either $\uparrow$ or $\downarrow$.*

Proof. Let o be any binary operator. If $\neg$ can be defined by o, then $\neg A \equiv t(o, A)$, where $t(o, A)$ is the formula represented by a binary tree t whose leaf nodes are labeled with A and whose internal nodes are labeled with o. Furthermore, we assume that t is such a binary tree with the least number of nodes. If we replace A by 1, then $t(o, 1)$ should have the truth value 0, because $\neg 1 = t(o, 1) = 0$. If any internal node of $t(o, 1)$ has the truth value 1, we may replace that node by a leaf node 1; if any internal node other than the root of t has truth value 0, we may use the subtree at this internal node as t. In both cases, this is a violation to the assumption that t is such a tree with the least of nodes. As a result, t should have only one internal node, i.e., $1\ o\ 1 = \neg 1 = 0$. Similarly, we should have $0\ o\ 0 = \neg 0 = 1$.

There are four cases regarding the truth values of $1\ o\ 0$ and $0\ o\ 1$.

- case 1: $1\ o\ 0 = 0$ and $0\ o\ 1 = 1$. Then $A\ o\ B = \neg A$ and we cannot use this o to define $A \wedge B$, as the output from o on A and B will be one of A, $\neg A$, B, or $\neg B$.

- case 2: $1\ o\ 0 = 1$ and $0\ o\ 1 = 0$. Then $A\ o\ B = \neg B$, and we still cannot use this o to define $A \wedge B$ for the same reason as in Case 1.

- case 3: $1\ o\ 0 = 0\ o\ 1 = 1$. o is $\uparrow$.

- case 4: $1\ o\ 0 = 0\ o\ 1 = 0$. o is $\downarrow$.

p	q	$p\ o_1\ q$	$p\ o_2\ q$	$p\ o_3\ q$	$p\ o_4\ q$
1	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	1	0
0	0	1	1	1	1

Table 2.4.1: $p\ o_1\ q \equiv \neg p$ (the negation of projection on the first argument); $p\ o_2\ q \equiv \neg q$ (the negation of projection on the second argument); $p\ o_3\ q \equiv p \uparrow q$ (NAND); and $p\ o_4\ q \equiv p \downarrow q$ (NOR).

The last two cases give us the definitions of $\uparrow$ and $\downarrow$, as shown in Table 2.4.1. □

Corollary 2.4.4. *Both* $\{\neg, \wedge\}$ *and* $\{\neg, \vee\}$ *are minimally sufficient.*

In Boolean algebra, the property of any sufficient set of operators can be specified using equations with variables and the operator. To specify the property of $\uparrow$, we may use three equations: $0 \uparrow Y = 1$, $X \uparrow 0 = 1$, and $1 \uparrow 1 = 0$. For $\uparrow$, what is the minimal number of equations needed? And what is the minimal number of symbols in the equations? These questions have been answered firmly: only one equation is needed and only three variables (for a total of either occurrences) and six uses of $\uparrow$ are needed. We state below the result without proof.

Proposition 2.4.5. *The Boolean algebra can be generated by the following equation:*

$$((x \uparrow y) \uparrow z) \uparrow (x \uparrow ((x \uparrow z) \uparrow x)) = z$$

The above equation contains three variables and 14 symbols (excluding parentheses). For the set $\{\neg, \vee\}$, a single axiom also exists to specify the Boolean algebra:

Proposition 2.4.6. *The Boolean algebra can be generated by the following equation:*

$$\neg(\neg(\neg(x \vee y) \vee z) \vee \neg(x \vee \neg(\neg z \vee \neg(z \vee u)))) = z.$$

The above equation contains four variables and 21 symbols (excluding parentheses).

2.4.2 Logic Minimization

Logic minimization, also called *logic optimization*, is a part of circuit design process and its goal is to obtain the smallest combinational circuit that is represented by a Boolean formula. Logic minimization seeks to find an equivalent representation

of the specified logic circuit under one or more specified constraints. Generally the circuit is constrained to minimum chip area meeting a pre-specified delay. Decreasing the number of gates will reduce the power consumption of the circuit. Choosing gates with less transistors will reduce the circuit area. Decreasing the number of nested levels of gates will reduce the delay of the circuit. Logic minimization will reduce substantially the cost of circuits and improve its quality.

2.4.2.1 Karnaugh Maps for Minimizing DNF and CNF

In the early days of electronic design automation (EDA), a Boolean function is often defined by a truth table and we can derive full DNF (sum of products) or full CNF (product of sums) directly from the truth table, as shown in Example 2.3.16. The obtained DNF or CNF need to be minimized, in order to use the least number of gates to implement this function.

Following the convention of EDA, we will use $+$ for $\vee$, $\cdot$ for $\wedge$, and $\overline{A}$ for $\neg A$.

Example 2.4.7. Let $f : \{0,1\}^2 \longrightarrow \{0,1\}$ be defined by $f(A,B) = \overline{A}\overline{B} + A\overline{B} + \overline{A}B$. We need three AND gates and one OR gate of 3-input. However, $f(A,B) \equiv \overline{A} + \overline{B}$. Using $\overline{A} + \overline{B}$, we need only one OR gate. □

The equivalence relations used in the above simplification process are $AB + A\overline{B} \equiv A$ and $A + \overline{A}B \equiv A + B$ (or $\overline{A} + AB \equiv \overline{A} + B$. For the above example, $\overline{A}\overline{B} + A\overline{B} \equiv \overline{B}$ and $\overline{B} + \overline{A}B \equiv \overline{B} + \overline{A}$.

We want to find an equivalent circuit of minimum size possible, and this is a difficult computation problem. There are many tools such as *Karnaugh maps* available to achieve this goal.

The Karnaugh map (K-map) is a popular method of simplifying DNF or CNF formulas, originally proposed by Maurice Karnaugh in 1953. The Karnaugh map reduces the need for extensive calculations by displaying the output of the function graphically and taking advantage of humans' pattern-recognition capability. It is suitable for Boolean functions with 2 to 4 variables. When the number of variables is greater than 4, it is better to use an automated tool.

The Karnaugh map uses a two-dimensional grid where each cell represents an interpretation (or equivalently, a full minterm) and the cell contains the truth value of the function for that interpretation. The position of each cell contains all the information of an interpretation and the cells are arranged such that adjacent cells differ by exactly one truth value in the interpretation. Adjacent 1s in the Karnaugh map represent opportunities to simplify the formula. The minterms for the final formula are found by encircling groups of 1s in the map. Minterm groups must be rectangular and must have an area that is a power of two (i.e., 1, 2, 4, 8, ...).

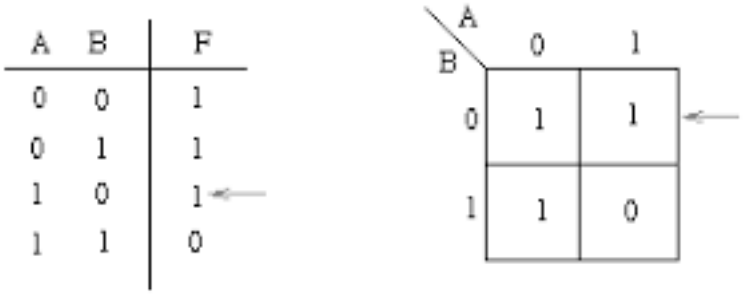


Figure 2.4.1: K-map for $f(x, y) = x\overline{y} + \overline{x}y + \overline{xy}$.

Minterm rectangles should be as large as possible without containing any 0s (it may contain don't-care cells). Groups may overlap in order to make each one larger. A least number of groups that cover all 1s will represent the minterms of a DNF of the function. These minterms can be used to write a minimal DNF representing the required function and thus implemented by the least number of AND gates feeding into an OR gate. The map can also be used to obtain a minimal CNF that is implemented by OR gates feeding into an AND gate.

Let $f(x, y) = x\overline{y} + \overline{x}y + \overline{xy}$. There are four minterms of x and y, the K-map has four cells, that is, four minterms corresponding to the four interpretations on x and y. As the output of f, three cells have value 1 and one cell (i.e., xy) has value 0. Initially, each cell belongs to a distinct group of one member (Figure 2.4.1).

> **The merge operation on K-map**: Merge two adjacent and disjoint groups of the same size and of the same truth value into one larger group.

This operation creates groups of size 2, 4, 8, etc., and cells are allowed to appear in different groups at the same time. For example, cells $x\overline{y}$ and $\overline{xy}$ in Figure 2.4.1 are adjacent, they are merged into one group $\{x\overline{y}, \overline{xy}\}$, which can be represented by $\overline{y}$ as a shorthand of the group. Similarly, cells $\overline{x}y$ and $\overline{xy}$ can be merged into $\{\overline{x}y, \overline{xy}\}$, which can be represented by $\overline{x}$. Note that $\overline{xy}$ is used twice in the two merge operations. No more merge operations can be performed, and the final result is $f = \overline{x} + \overline{y}$ (f is the NAND function). This result can be proved logically as follows:

$$\begin{aligned} f &= x\overline{y} + \overline{x}y + \overline{xy} \\ &= (x\overline{y} + \overline{xy}) + (\overline{x}y + \overline{xy}) \\ &= (x + \overline{x})\overline{y} + \overline{x}(y + \overline{y}) \\ &= \overline{y} + \overline{x} \end{aligned}$$

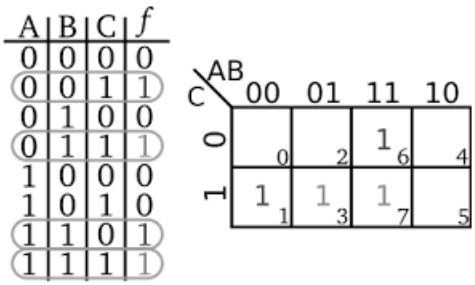


Figure 2.4.2: The K-map for obtaining $f(A, B, C) = AB + \overline{A}C$.

Example 2.4.8. Consider the function f defined by the truth table in Figure 2.4.2. The corresponding K-map has four cells with truth value 1 as f has four models shown in the truth table. There are three possible merge operations: (i) $\overline{A}B\overline{C}$ and $\overline{A}BC$ merge into $\overline{A}C$; (ii) $AB\overline{C}$ and ABC merge into AB; and (iii) $\overline{A}BC$ and ABC merge into BC. Since all the cells with value 1 are in the groups represented by $\overline{A}C$ and AB, BC is redundant. So the simplified function is $f = \overline{A}C + AB$, instead of $f = \overline{A}C + AB + BC$. On the other hand, if $g = AB + AC + BC$, then none of the three minterms in g is redundant, because if you delete one, you will miss some 1s, even though the full minterm ABC is covered by all the three minterms. □

From the above examples, it is clear that after each merge operation, two minterms are merged into one with one less variable. That is, the larger the group, the less the number of variables in the minterm to represent the group. This is possible because adjacent cells represent interpretations which differ by exactly one truth value.

Because K-maps are two-dimensional, some adjacent relations of cells are not shown on K-maps. For example, in Figure 2.4.2, $m_0 = \overline{A}\overline{B}\overline{C}$ and $m_4 = A\overline{B}\overline{C}$ are adjacent, $m_1 = \overline{A}\overline{B}C$ and $m_5 = A\overline{B}C$ are adjacent. Strictly speaking, K-maps are *toroidally* connected, which means that rectangular groups can wrap across the edges. Cells on the extreme right are actually "adjacent" to those on the far left, in the sense that the corresponding interpretations only differ by one truth value. We need a 3-dimensional graph to completely show the adjacent relation of 3 variables; a 4-dimensional graph to completely show the adjacent relation of 4 variables; etc.

Based on the idea of K-maps, an automated tool will create a graph of 2^n nodes for n variables, each node represents one interpretation, and two nodes have an edge iff their interpretations differ by one truth value. The merge operation is then implemented on this graph to generate minterms for a least size DNF. The same graph is also used for generating Gray code (by finding a Hamiltonian path

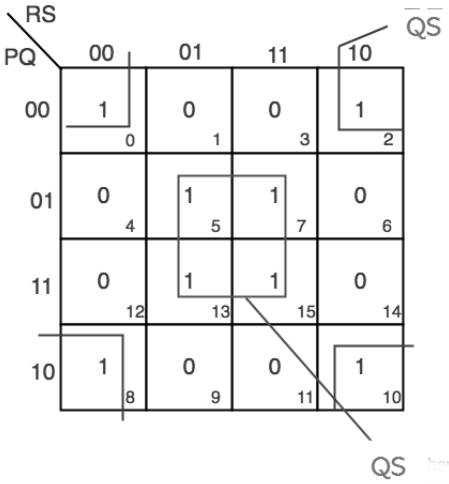


Figure 2.4.3: The K-map for $f(P,Q,R,S) = m_0 + m_2 + m_5 + m_7 + m_8 + m_{10} + m_{13} + m_{15}$. The simplified formula is $f = \overline{QS} + QS$.

in the graph).

Example 2.4.9. The K-map for $f(P,Q,R,S) = m_0 + m_2 + m_5 + m_7 + m_8 + m_{10} + m_{13} + m_{15}$, is given in Figure 2.4.3, where $m_0 = \overline{PQRS}$, $m_2 = \overline{PQR}\overline{S}$, $m_5 = \overline{P}Q\overline{R}S$, $m_7 = \overline{P}QRS$, $m_8 = P\overline{QRS}$, $m_{10} = P\overline{QR}\overline{S}$, $m_{13} = PQ\overline{R}S$, and $m_{15} = PQRS$ (i in m_i is the decimal value of $PQRS$ in the corresponding interpretation). At first, m_0 and m_2 are adjacent and can be merged into $\overline{PQS}$; m_8 and m_{10} are also adjacent and can be merged into $P\overline{QS}$. Then $\overline{PQS}$ and $P\overline{QS}$ can be further merged into $\overline{QS}$. Similarly, m_5 and m_7 merge into $\overline{P}QS$; m_{13} and m_{15} merge into PQS. Then $\overline{P}QS$ and PQS merge into QS. The final result is $f = \overline{QS} + QS$. It would also have been possible to derive this simplification by carefully applying the equivalence relations, but the time it takes to do that grows exponentially with the number of minterms. □

Note that the formula obtained from a K-map is not unique in general. For example, $g(P,Q,R,S) = P\overline{Q} + R\overline{S} + \overline{PQS}$ or $g(P,Q,R,S) = P\overline{Q} + R\overline{S} + \overline{QRS}$ are possible outputs for g from K-maps. This means the outputs of K-maps are not canonical.

K-maps can be used to generate simplified CNF for a function. In Example 2.3.16, we have shown how to obtain a full CNF from a full DNF. Using the same idea, if we simplify the full DNF first, the CNF obtained by negating the simplified DNF will a simplified one.

Example 2.4.10. Let $f(A,B,C) = \overline{AB}C + \overline{A}B\overline{C} + A\overline{BC} + A\overline{B}C$ and its K-map

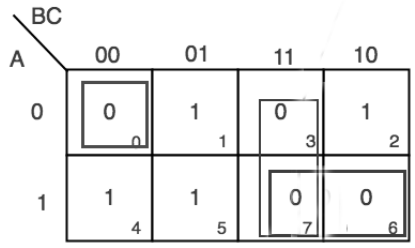


Figure 2.4.4: The K-map for $f(A,B,C) = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$. The simplified CNF is $f = (A+B+C)(\overline{A}+\overline{B})(\overline{B}+\overline{C})$.

is given in Figure 2.4.4. The negation of f is

$$\overline{f} = \overline{A}\overline{B}\overline{C} + \overline{A}BC + AB\overline{C} + ABC$$

Since $\overline{A}BC$ and ABC can merge into BC (shown in the figure), $AB\overline{C}$ and ABC can merge into AB, the simplified $\overline{f}$ is $\overline{f} = \overline{A}\overline{B}\overline{C} + AB + BC$. The simplified CNF for f is thus $f = (A+B+C)(\overline{A}+\overline{B})(\overline{B}+\overline{C})$. □

Karnaugh maps can also be used to handle "don't care" values and can simplify logic expressions in software design. Boolean conditions, as used for example in conditional statements, can get very complicated, which makes the code difficult to read and to maintain. Once minimized, canonical sum-of-products and product-of-sums expressions can be implemented directly using AND and OR logic operators.

2.4.2.2 Other Issues in Logic Minimalization

Simplified Boolean functions obtained from K-maps are DNFs or CNFs. To implement them in a circuit using CMOS (complementary metal–oxide–semiconductor), we need further optimization. One of the most important characteristics of CMOS devices is low static power consumption. CMOS devices draw significant power only momentarily during switching between on and off states. Consequently, CMOS devices do not produce as much waste heat as other forms of logic, like transistor–transistor logic (TTL), which normally has some standing current even when not changing state. These characteristics allow CMOS to integrate a high density of logic functions on a chip. It was primarily for this reason that CMOS became the most widely used technology to be implemented in VLSI (very large scale integration) chips.

Using CMOS, the NAND/NOR/NOT gates are cheaper to implement than the AND/OR gates. For a basic unbuffered NAND/NOR/NOT gate, we need only

two transistors per input. It works when that gate is one among many others, driving a few similar gates. Many practical discrete gates, however, will be buffered and they will need additional transistors to improve the output strength and prevent the output strength from depending on active inputs. Typically, AND/OR gates will be implemented using the NAND/NOR/NOT gates and need more transistors than NAND/NOR gates.

Example 2.4.11. If we directly implement $f(A,B,C) = \overline{A}C + \overline{B}$ in CMOS, we need two inverters (for negation), one OR gate (as the negation of $\downarrow$) and one AND gate (as the negation of $\uparrow$). That is, we need four inverters (each uses two transistors), one NOR gate and one NAND gate (each uses four transistors), for a total of 16 transistors. We may convert $f(A,B,C) = \overline{A}C + \overline{B}$ into the simplified and equivalent formula using only $\uparrow$ and $\neg$: $f(A,B,C) = (\overline{A} \uparrow C) \uparrow B$, which uses a total of 10 transistors (two for the inverter; four each for the two NAND gates). □

To convert DNF or CNF into a formula using only NAND/NOR/NOT, we may use the algorithm *NNF2NOR*.

Algorithm 2.4.12. The algorithm *NNF2NOR* takes a formula A in NNF as input and returns a formula B which is equivalent to A and uses only NOR/NAND/NOT operators. It calls a mutually recursive algorithm *neg2NOR*, which takes a formula A in NNF as input and returns a formula B which is equivalent to $\neg A$ and uses only NOR/NAND/NOT operators.

proc *NNF2NOR*(A)
    if $A \in V_P$ or $A = \neg B$ **return** A;
    if $A = B \vee C$ **return** *neg2NOR*(B) $\uparrow$ *neg2NOR*(C);
    if $A = B \wedge C$ **return** *neg2NOR*(B) $\downarrow$ *neg2NOR*(C);

proc *neg2NOR*(A)
    if $A \in V_P$ **return** $\neg A$;
    if $A = \neg B$ **return** B;
    if $A = B \vee C$ **return** *NNF2NOR*(B) $\downarrow$ *NNF2NOR*(C);
    if $A = B \wedge C$ **return** *NNF2NOR*(B) $\uparrow$ *NNF2NOR*(C);

For input $A = (\overline{p} \vee q) \vee r$, *NNF2NOR*($A$) returns $(\overline{p} \downarrow q) \uparrow \overline{r}$, which needs 12 transistors to implement. However, this result is not optimal, because another equivalent formula, i.e., $p \uparrow (q \downarrow r)$, needs eight transistors to implement in CMOS. Relatively speaking, finding optimal formulas from DNF/CNF is an easy task. The

trick is to get rid of as many negation in literals as possible, by switching between $\uparrow$ and $\downarrow$, using the relation

$$(\neg A \uparrow B) \downarrow \neg C \equiv A \downarrow (B \uparrow C);$$
$$(\neg A \downarrow B) \uparrow \neg C \equiv A \uparrow (B \downarrow C).$$

If a 3-input NAND gate is available, the same A can be represented by $\uparrow (p, \overline{q}, \overline{r})$, which needs 10 transistors to implement. Note that $\uparrow (A, B, C)$, $(A \uparrow B) \uparrow C$, and $A \uparrow (B \uparrow C)$ are all logically different.

DNFs (sum of products) and CNFs (product of sums) are called *2-level* formulas. A *3-level formula* is a product of DNFs or a sum of CNFs. A *4-level formula* is a product of 3-level formulas or a sum of 3-level formulas, and so on. Finding equivalent formulas of high levels can possibly reduce the size of formulas.

Example 2.4.13. Let $f(a, b, c, d, f, g) = adf + aef + bdf + bef + cdf + cef + g$. f is a DNF and we need 6 AND gates of 3-input and 3 OR gates of 3-input (to implement the 7 input OR) for a total of 9 gates. If we allow 3-level formulas, then $f(a, b, c, d, f, g) \equiv (a + b + c)(d + e)f + g$, which needs one OR gate of 3 input, 2 OR gates of 3 input, one AND gate of 3 input, for a total of 4 gates. □

Besides using higher level formulas, CMOS technology allows to use other types of gates besides OR, AND, and inverters (for negation). For example, if we want to implement a function $f(a, b) = a\overline{b}+\overline{a}b$, since $f(a, b) \equiv a \oplus b$, one XOR gate will suffice. Many circuit design systems provide a complex of libraries with a large number of gates. For example, the EE271 library allows the use of the following gates for combinational circuits:

> inv, nand2, nand3, nand4, nor2, nor3, xor, xnor, aoi2, aoi22, oai2, oai22, xnor

where "xnor" is the negation of "xor", i.e., the logical equal ($\leftrightarrow$), "aoi" stands for "AND-OR-Invert", "oai" stands for "OR-AND-Invert", and they define the following functions:

aoi2:	$F = \overline{(A \wedge B) \vee C}$
aoi22:	$F = \overline{(A \wedge B) \vee (C \wedge D)}$
oai2:	$F = \overline{(A \vee B) \wedge C}$
oai22:	$F = \overline{(A \vee B) \wedge (C \vee D)}$

AOI and OAI gates can be readily implemented in CMOS circuitry. AOI gates are particularly advantaged in that the total number of transistors is less than

if the AND, NOT, and OR functions were implemented separately. This results in increased speed, reduced power, smaller area, and potentially lower fabrication cost. For example, an AOI2 gate can be constructed with 6 transistors in CMOS compared to 10 transistors using a 2-input NAND gate (4 transistors), an inverter (2 transistors), and a 2-input NOR gate (4 transistors), because $F = \overline{(A \wedge B) \vee C} \equiv \overline{A \uparrow B \downarrow C}$.

Finding an equivalent and minimal circuit with high-level formulas of a rich set of gates becomes even more challenging and is an on-going research area in EDA.

2.4.3 Maximum Satisfiability

When a CNF is unsatisfiable, we are still interested in finding an interpretation which makes most clauses to be true. This is so-called the *maximum satisfiability problem*, which is the problem of determining the maximum number of clauses, of a given Boolean formula in CNF, that can be made true by an interpretation (an assignment of truth values to the variables of the formula). It is a generalization of the Boolean satisfiability problem, which asks whether there exists a truth assignment that makes all clauses true.

Example 2.4.14. Let A be a set of unsatisfiable clauses $\{(\overline{p}), (p \vee \overline{q}), (p \vee q), (p \vee \overline{r}), (p \vee r)\}$. If we assign 0 to p, two clauses will be false no matter what values are assigned to q and r. On the other hand, if p is assigned 1, only clause $(\overline{p})$ will be false. Therefore, if A is given as an instance of the MAX-SAT problem, the solution to the problem is 4 (four clauses are true in one interpretation). □

More generally, one can define a weighted version of MAX-SAT as follows: given a conjunctive normal form formula with non-negative weights assigned to each clause, find truth values for its variables that maximize the combined weight of the satisfied clauses. The MAX-SAT problem is an instance of weighted MAX-SAT where all weights are 1.

Find the final exam schedule for a big university is a challenging job. There are hard constraints such as no students can take two exams at the same, and no classrooms can be hold two exams at the same time. There are also software constraints such that one student is preferred to take at most two exams in one day, or two exams are separated by two hours. To specify such a problem as an instance of weighted MAX-SAT is easy: We just give a very large weight to hard constraints and a small weight to soft constraints.

Various methods have developed to solve MAX-SAT or weighted MAX-SAT problems. These methods are related to solving SAT problems and we will present

them later in the book.

2.5 Using Propositional Logic

Propositions and logical operators arise all the time in computer programs. All programming languages such as C, C++, and Java use symbols like "&&", "||", and "!" in place of $\wedge$, $\vee$, and $\neg$. Consider the Boolean expression appearing in an if-statement

```
if ( x > 0 || (x <= 0 && y > 100 )) ...
```

If we let p denote `x > 0` and q be `y > 100`, then the Boolean expression is abstracted as $p \vee \neg p \wedge q$. Since $(p \vee \neg p \wedge q) \equiv (p \vee q)$, the original code can be rewritten as follows:

```
if ( x > 0 || y > 100 ) ...
```

In other words, the knowledge about the logic may help us to write neater codes.

Consider another piece of the code:

```
if ( x >= 0 && A[x] == 0 ) ...
```

Let p be `x >= 0` and q be `A[x] == 0`, then the Boolean expression is $p \wedge q$. Since $p \wedge q \equiv q \wedge p$, it is natural to think that `( x >= 0 && A[x] == 0 )` can be replaced by `( A[x] == 0 && x >= 0 )` and the program will be the same. Unfortunately, this is not the case: the execution of `( A[x] == 0 && x >= 0 )` will be aborted when `x < 0`. This example shows that "&&" in programming languages are not commutative in general. Neither is "||" as `( x < 0 || A[x] == 0 )` is different from `( A[x] == 0 || x < 0 )`. These abnormalities of the logic not only appear in array indices, but also when assignments are allowed in Boolean expressions. For example, `x > 0 || (x = c) > 10` is different from `(x = c) > 10 || x > 0` in C, where `x = c` is an assignment command. As a programmer we need to be careful about these abnormalities.

2.5.1 Bitwise Operators

In programming languages like C, C++, and Java, an integer is represented by a list of 16-, 32-, or 64-bits (for C, a char is regarded as an 8-bit integer or a byte). A bitwise operation operates on one or two integers at the level of their individual bits and several of them are logical operations on bits. It is a fast and simple action, directly supported by the processor, and should be used whenever possible.

$(a)_2$	=	$(0,1,0,0,0,0,1,1)$
$(b)_2$	=	$(0,0,0,1,0,0,1,0)$

name	**symbol**	**example**
NOT	~	$(\sim a)_2 = (1,0,1,1,1,1,0,0)$
OR	|	$(a \mid b)_2 = (0,1,0,1,0,0,1,1)$
AND	&	$(a \,\&\, b)_2 = (0,0,0,0,0,0,1,0)$
XOR	^	$(a \wedge b)_2 = (0,1,0,1,0,0,0,1)$
shift right	>>	$(a >> 2)_2 = (0,0,0,1,0,0,0,0)$
shift left	<<	$(b << 2)_2 = (0,1,0,0,1,0,0,0)$

Table 2.5.1: Examples of bitwise operators.

For illustration purpose, we will use 8-bit integers as examples. In general, for any 8-bit integer x, the binary representation of x is a 8-bit vector $(x)_2 = (b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$, where $b_i \in \{0,1\}$ and $x = \sum_{i=0}^{7} b_i * 2^i$. For example, for $a = 67$, its bit vector is $(a)_2 = (0,1,0,0,0,0,1,1)$, because $67 = 64 + 2 + 1 = 2^6 + 2^1 + 2^0$, and for $b = 18$, $(b)_2 = (0,0,0,1,0,0,1,0)$, since $18 = 16 + 2 = 2^4 + 2^1$. In fact, $(a)_2$ and $(b)_2$ are internal representation of a and b in computer.

The following bitwise operators are provided, treating each bit as a Boolean value and performing on them simultaneously. We will use the bit vectors of $a = 67$, and $b = 18$ as examples.

If necessary, other bitwise operations can be implemented using the above four operations. For example a bitwise implication on a and b can be implemented as $\sim a \mid b$.

The bit shifts are also considered bitwise operations, because they treat an integer as a bit vector rather than as a numerical value. In these operations the bits are moved, or shifted, to the left or right. Registers in a computer processor have a fixed width, so some bits will be "shifted out" of the register at one end, while the same number of bits are "shifted in" from the other end. Typically, the "shifted out" bits are discarded and the "shifted in" bits are all 0's. When the leading bit is the sign of an integer, the left shift operation becomes more complicated and its discussion is out of the scope of this book. In the table, we see that the shift operators take two integers: $x << y$ will shift $(x)_2$ left by y bits; $x >> y$ will shift $(x)_2$ right by y bits.

The value of the bit vector $x << y$ is equal to $x * 2^y$, provided that this multiplication does not cause an overflow. E.g., for $b = 18$, $b << 2 = 18 * 4 = 72$. Similarly, the value of $x >> y$ is equal to the integer division $x/2^y$. E.g., for $a = 67$,

$a >> 2 = \lfloor 67/4 \rfloor = 16$. Shifting provides an efficient way of multiplication and division when the second operand is a power of two. Typically, bitwise operations are substantially faster than division, several times faster than multiplication.

Suppose you are the manager for a social club of 64 members. For each gathering, you need to record the attendance. If each member has a unique member id $\in \{0, 1, ..., 63\}$, you may use a set of member ids to record the attendance. Instead of a list of integers, you may use a single 64-bit integer, say s, to store the set. That is, if a member id is i, then the i^{th} bit of s is 1. For this purpose, we need a way to set a bit in s to be 1. Suppose $s = 0$ initially, we may use

$$s = s \mid (1 << i)$$

to set the i^{th} bit of s to be 1. To set the value of the i^{th} bit of s to be 0, use

$$s = s \ \& \ ^{\sim}(1 << i).$$

To get the value of the i^{th} bit of s, use $s \ \& \ (1 << i)$. For another gathering, we use another integer t. Now it is easy to compute the union or intersection of these sets: $s \mid t$ is the union and $s \ \& \ t$ is the intersection. For example, if you want to check that nobody attended both gatherings, you can check if $s \ \& \ t = 0$, which is more efficient than using lists for sets.

2.5.2 Specify Problems in Propositional Logic

Propositional logic is a very simple language but rich enough to specify most decision problems in computer science, because these problems belong to the class of NP problems and SAT is NP-complete. In fact, propositional logic is a formal language much more rigorous than natural languages. Since natural languages are ambiguous, expressing a sentence in logic helps us to understand the exact meaning of the sentence. For example, the two sentences

1. Students and seniors pay half price,

2. Students or seniors pay half price

evidently have the same meaning.

How to explain this? Let "being student" and "being senior" be abbreviated by p and q, and "pay half price" by r. Let A be $(p \to r) \wedge (q \to r)$, B be $(p \vee q) \to r$, then A and B express somewhat more precisely the factual content of sentences 1 and 2, respectively. It is easy to show that the formulas A and B are logically equivalent. The everyday-language statements of A and B obscure the structural

difference of A and B through an apparently synonymous use of the words "and" and "or".

We will show to use specify various problems in propositional logic through examples.

Example 2.5.1. The so-called n-queen example is stated as follows: Given a chessboard of size $n \times n$, where $n > 0$, how to place n queens on the board such that no two queens attack each other? For $n = 2, 3$, there is no solution; for other n, there exists a solution. To specify this problem in logic, we need n^2 propositional variables. Let the variables be $p_{i,j}$, where $1 \leq i, j \leq n$, and the meaning of $p_{i,j}$ is that $p_{i,j}$ is true iff there is a queen at row i and column j. The following properties ensure that there are n queens on the board such that no two queens attack each other:

1. There is at least one queen in each row.

$$\wedge_{1 \leq i \leq n}(p_{i,1} \vee p_{i,2} \vee \cdots \vee p_{i,n})$$

2. No more than one queen in each row.

$$\wedge_{1 \leq i \leq n}(\wedge_{1 \leq j < k \leq n}(\neg p_{i,j} \vee \neg p_{i,k}))$$

3. No more than one queen in each column.

$$\wedge_{1 \leq i \leq n}(\wedge_{1 \leq j < k \leq n}(\neg p_{j,i} \vee \neg p_{k,i}))$$

4. No more than one queen on each diagonal.

$$\wedge_{1 \leq i < k \leq n}(\wedge_{1 \leq j,l \leq n, k-i=|l-j|}(\neg p_{i,j} \vee \neg p_{k,l}))$$

The formula following each property expresses the property formally, and can be easily converted into clauses. When $n = 2, 3$, the formulas are unsatisfiable. For $n \geq 4$, the formulas are satisfiable and each model gives us a solution. For example, when $n = 4$, we have two solutions: $\{p_{1,2}, p_{2,4}, p_{3,1}, p_{4,3}\}$ and $\{p_{1,3}, p_{2,1}, p_{3,4}, p_{4,2}\}$. If we assign the four variables in a solution to be true and the other variables false, we obtain a model. Note that we ignore the property that "there is at least one queen in each row," because it is redundant. □

The next example is a logical puzzle.

Example 2.5.2. Three people are going to the beach, each using a different mode of transportation (car, motorcycle, and boat) in a different color (blue, orange, green). Who's using what? The following clues are given:

1. Abel loves orange, but she hates travel on water.
2. Bob did not use the green vehicle.
3. Carol drove the car.

□

The first step to solve a puzzle using propositional logic is to make sure what properties are assumed and what conditions are provided by clues. For this puzzle, the first assumed property is that "everyone uses a unique transportation tool".

To specify this property in logic, there are several ways of doing it. One way is to use a set of propositional variables to specify the relation of "who uses what transportation": Let $p_{x,y}$ denote a set of nine variables, where $x \in \{a, b, c\}$ (a for Abel, b for Bob, and c for Carol), and $y \in \{c, m, b\}$ (c for car, m for motorcycle, and b for boat). Now the property that "everyone uses a unique transportation tool" can be specified formally as the following clauses:

(1)	$(p_{a,c} \mid p_{a,m} \mid p_{a,b})$	Abel uses c, m, or b;
(2)	$(p_{b,c} \mid p_{b,m} \mid p_{b,b})$	Bob uses c, m, or b;
(3)	$(p_{c,c} \mid p_{c,m} \mid p_{c,b})$	Carol uses c, m, or b;
(4)	$(\overline{p_{a,c}} \mid \overline{p_{b,c}})$	Abel and Bob cannot both use c;
(5)	$(\overline{p_{a,m}} \mid \overline{p_{b,m}})$	Abel and Bob cannot both use m;
(6)	$(\overline{p_{a,b}} \mid \overline{p_{b,b}})$	Abel and Bob cannot both use b;
(7)	$(\overline{p_{a,c}} \mid \overline{p_{c,c}})$	Abel and Carol cannot both use c;
(8)	$(\overline{p_{a,m}} \mid \overline{p_{c,m}})$	Abel and Carol cannot both use m;
(9)	$(\overline{p_{a,b}} \mid \overline{p_{c,b}})$	Abel and Carol cannot both use b;
(10)	$(\overline{p_{b,c}} \mid \overline{p_{c,c}})$	Bob and Carol cannot both use c;
(11)	$(\overline{p_{b,m}} \mid \overline{p_{c,m}})$	Bob and Carol cannot both use m;
(12)	$(\overline{p_{b,b}} \mid \overline{p_{c,b}})$	Bob and Carol cannot both use b.

It appears quite cumbersome that 12 clauses are needed to specify one property. Note that the goal of using logic is to find a solution automatically. For computers, 12 million clauses are not too many.

From the clues of the puzzle, we have the following two unit clauses:

(13)	$(p_{c,c})$	Carol drove the car;
(14)	$(\overline{p_{a,b}})$	Abel hates travel on water.

These two unit clauses forces $p_{c,c}$ to be assigned 1 and $p_{a,b}$ to be 0. Then we can decide the truth values of the rest seven variables:

$$\begin{array}{lll}
p_{a,c} \mapsto 0 & \text{from} & p_{c,c} \mapsto 1, (7); \\
p_{b,c} \mapsto 0 & \text{from} & p_{c,c} \mapsto 1, (11); \\
p_{a,m} \mapsto 1 & \text{from} & p_{a,c} \mapsto 0, p_{a,b} \mapsto 0, (1); \\
p_{b,m} \mapsto 0 & \text{from} & p_{a,m} \mapsto 1, (5); \\
p_{c,m} \mapsto 0 & \text{from} & p_{a,m} \mapsto 1, (8); \\
p_{b,b} \mapsto 1 & \text{from} & p_{b,c} \mapsto 0, p_{b,m} \mapsto 0, (2); \\
p_{c,b} \mapsto 0 & \text{from} & p_{b,b} \mapsto 1, (12).
\end{array}$$

The second assumed property is that "every transportation tool has a unique color". If we read the puzzle carefully, the clues involve the relation that "who will use what color of the transportation". So it is better to use $q_{x,z}$ to represent this relation that person x uses color y, where $x \in \{a, b, c\}$ and $z \in \{b, o, g\}$ (b for blue, o for orange, and g for green). We may specify the property that "everyone uses a unique color" in a similar way as we specify "everyone uses a unique transportation tool". The clues give us the following unit clauses: $(q_{a,o})$, and $(\overline{q_{b,g}})$. From these two unit clauses, we can derive the truth values of 12 $q_{x,y}$ variables.

The next puzzle looks quite different from the previous one, but its specification is similar.

Example 2.5.3. Four people sit on a bench of four seats for a group photo: two Americans (A), a Briton (B), and a Canadian (C). They asked that (i) two Americans do not want to sit next to each other; (ii) the Briton likes to sit next to the Canadian. Suppose we use propositional variables X_y, where $X \in \{A, B, C\}$ and $1 \leq y \leq 4$, with the meaning that "X_y is true iff the people of nationality X sits at seat y of the bench". How do you specify the problem in CNF over X_y such that the models of the CNF matches exactly all the sitting solutions of the four gentlemen?

The CNF will specify the following conditions:

1. Every seat takes at least one person: $(A_1 \mid B_1 \mid C_1), (A_2 \mid B_2 \mid C_2)$, $(A_3 \mid B_3 \mid C_3)$, $(A_4 \mid B_4 \mid C_4)$. The truth of these clauses implies that there are at least four people using these seats.

2. Each seat can take at most one person: $(\neg A_1 \mid \neg B_1), (\neg A_1 \mid \neg C_1)$, $(\neg B_1 \mid \neg C_1)$, $(\neg A_2 \mid \neg B_2)$, $(\neg A_2 \mid \neg C_2)$, $(\neg B_2 \mid \neg C_2)$, $(\neg A_3 \mid \neg B_3)$, $(\neg A_3 \mid \neg C_3)$, $(\neg B_3 \mid \neg C_3)$, $(\neg A_4 \mid \neg B_4)$, $(\neg A_4 \mid \neg C_4)$, $(\neg B_4 \mid \neg C_4)$. Combining with the first condition, this condition implies that there are exactly four persons sitting on the bench.

3. At most two Americans on the bench: $(\neg A_1 \mid \neg A_2 \mid \neg A_3)$, $(\neg A_1 \mid \neg A_2 \mid \neg A_4)$, $(\neg A_1 \mid \neg A_3 \mid \neg A_4)$, $(\neg A_2 \mid \neg A_3 \mid \neg A_4)$. These clauses say that for every three seats, one of them must not be taken by an American.

4. At most one Briton on the bench: $(\neg B_1 \mid \neg B_2), (\neg B_1 \mid \neg B_3)$, $(\neg B_1 \mid \neg B_4)$, $(\neg B_2 \mid \neg B_3)$, $(\neg B_2 \mid \neg B_4)$, $(\neg B_3 \mid \neg B_4)$. These clauses say that for every two seats, one of them must not be taken by the Canadian.

5. At most one Canadian on the bench: $(\neg C_1 \mid \neg C_2), (\neg C_1 \mid \neg C_3)$, $(\neg C_1 \mid \neg C_4)$, $(\neg C_2 \mid \neg C_3)$, $(\neg C_2 \mid \neg C_4)$, $(\neg C_3 \mid \neg C_4)$.

6. The Americans do not want to sit next to each other: $(\neg A_1 \mid \neg A_2)$, $(\neg A_2 \mid \neg A_3)$, $(\neg A_3 \mid \neg A_4)$.

7. The Briton likes to sit next to a Canadian: $(\neg C_1 \mid B_2)$, $(\neg C_2 \mid B_1 \mid B_3)$, $(\neg C_3 \mid B_2 \mid B_4)$, $(\neg C_4 \mid B_3)$, $(\neg B_1 \mid C_2)$, $(\neg B_2 \mid C_1 \mid C_3)$, $(\neg B_3 \mid C_2 \mid C_4)$, $(\neg B_4 \mid C_3)$. If we read $(\neg C_2 \mid B_1 \mid B_3)$ as $C_2 \rightarrow B_1 \vee B_3$, then it means that if the Canadian takes the second seat, then the Briton must sit at either seat 1 or seat 3.

Some conditions such as there are at least two Americans, and at least one Briton and one Canadian, can also be easily specified by clauses. However, these clauses are logical consequence of the other clauses, as conditions 1 and 2 ensure that there are four persons sitting on the bench. If there are at most one Briton and one Canadian, then condition 2 is also redundant, and we may remove them safely. Note that the input clauses serve as a set of axioms. This set is minimal if no clauses are logical consequence of other clauses. Of course, without a through search, we cannot identify all the clauses that are logical consequences of the other clauses. □

The next problem is called *Knights and Knaves*, which was coined by Raymond Smullyan in his 1978 work *What Is the Name of This Book?* The problem is actually a collection of similar logic puzzles where some characters can only answer questions truthfully, and others only falsely. The puzzles are set on a fictional island where all inhabitants are either *knights*, who always tell the truth, or *knaves*, who always lie. The puzzles involve a visitor to the island who meets small groups of inhabitants. Usually the aim is for the visitor to deduce the inhabitants' type from their statements, but some puzzles of this type ask for other facts to be deduced. The puzzles may also be to determine a yes-no question which the visitor can ask in order to discover a particular piece of information.

Example 2.5.4. You meet two inhabitants: Zoey and Mel. Zoey tells you that "Mel is a knave". Mel says, "Neither Zoey nor I are knaves." Can you determine who is a knight and who is a knave? □

Let p stands for "Zoey is knight", with the understanding that if p is true, then Zoey is a knight; if p is false, then Zoey is a knave. Similarly, define q as "Mel is knight". Using a function $says$, what they said can be expressed as

$$(1)\ \ says(Zoey, \neg q); \qquad\qquad (2)\ \ says(Mel, p \wedge q),$$

together with the two axioms:

$$says(knight, X) \equiv X; \qquad says(knave, X) \equiv \neg X;$$

Now using a truth table to check all the truth values of p and q, only when $p \mapsto 1$ and $q \mapsto 0$, both (1) and (2) are true. That is, we replace Zoey by knight and Mel by knave in (1) and (2), to obtain $\neg q$ and $\neg(p \wedge q)$, respectively. Then $\{p \mapsto 1, q \mapsto 0\}$ is a model of $\neg q \wedge \neg(p \wedge q)$. Thus the solution is "Zoey is a knight and Mel is a knave."

Example 2.5.5. You meet two inhabitants: Peggy and Zippy. Peggy tells you that "of Zippy and I, exactly one is a knight". Zippy tells you that only a knave would say that Peggy is a knave. Can you determine who is a knight and who is a knave? □

Let p be "Peggy is knight" and q be "Zippy is knight". Then what they said can be expressed as

$$(1)\ \ says(Peggy, p \oplus q); \qquad\qquad (2)\ \ says(Zippy, says(knave, \neg p)).$$

The only solution is $p \mapsto 0$ and $q \mapsto 0$, i.e., both Peggy and Zippy are knaves.

2.6 Exercise Problems

1. How many logical operators which take five arguments? That is, how many functions of type $f : B^5 \longrightarrow B$, where $B = \{0, 1\}$?

2. Suppose $V_P = \{p, q, r\}$. Then $\mathcal{M}((p \to q) \wedge r) =?$ and $\mathcal{M}((p \to q) \wedge \neg q) =?$

3. Prove Theorem 2.2.11, for any formulas A and B, $\mathcal{M}(A \wedge B) = \mathcal{M}(A) \cap \mathcal{M}(B)$ and $\mathcal{M}(\neg A) = IV_P - \mathcal{M}(A)$.

4. Answer the following questions with explanation:

(a) Suppose that you have shown that whenever X is true, then Y is true, and whenever X is false, then Y is false. Have you now demonstrated that X and Y are logically equivalent?

(b) Suppose that you have shown that whenever X is true, then Y is true, and whenever Y is false, then X is false. Have you now demonstrated that X and Y are logically equivalent?

(c) Suppose you know that X is true iff Y is true, and you know that Y is true iff Z is true. Is this enough to show that X, Y, Z are all logically equivalent?

(d) Suppose you know that whenever X is true, then Y is true; that whenever Y is true, then Z is true; and whenever Z is true, then X is true. Is this enough to show that X, Y, Z are all logically equivalent?

5. Provide the truth table for defining $ite : B^3 \longrightarrow B$, the if-then-else operator.

6. Provide a BNF grammar which defines the well-formed formulas without unnecessary parentheses, assuming the precedence relation from the highest to the lowest: $\neg,\ \wedge,\ \vee,\ \ \rightarrow,\ \{\oplus, \leftrightarrow\}$.

7. Prove by the truth table method that the following formulas are valid.

$$\begin{array}{ll} (a) & (A \wedge (A \rightarrow B)) \rightarrow B \\ (b) & (\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B) \\ (c) & (A \oplus B) \leftrightarrow ((A \vee B) \wedge (\neg A \vee \neg B)) \\ (d) & (A \leftrightarrow B) \leftrightarrow ((A \vee \neg B) \wedge (\neg A \vee B)) \end{array}$$

8. Prove by the truth table method that the following entailments hold.

$$\begin{array}{ll} (a) & (A \wedge (A \rightarrow B)) \models B \\ (b) & (A \rightarrow (B \rightarrow C)) \models (A \rightarrow B) \rightarrow (A \rightarrow C) \\ (c) & (\neg B \rightarrow \neg A) \models (A \rightarrow B) \\ (d) & ((A \vee B) \wedge (\neg A \vee C)) \models (B \vee C) \end{array}$$

9. Prove by the truth table method that the following logical equivalences hold.

$$\begin{array}{ll} (a) & \neg(A \vee B) \equiv \neg A \wedge \neg B \\ (b) & (A \vee B) \wedge (A \vee \neg B) \equiv A \\ (c) & (A \leftrightarrow B) \equiv ((A \wedge B) \vee (\neg A \wedge \neg B)) \\ (d) & (A \oplus B) \equiv ((A \wedge \neg B) \vee (\neg A \wedge B)) \\ (e) & (\overline{A} \downarrow B) \uparrow \overline{C} \equiv A \uparrow (B \downarrow C) \\ (f) & (\overline{A} \uparrow B) \downarrow \overline{C} \equiv A \downarrow (B \uparrow C) \end{array}$$

10. Prove the equivalence of the following formulas by using other equivalence relations:

    (a) $A \oplus B \equiv \neg(A \leftrightarrow B) \equiv \neg A \leftrightarrow B \equiv A \leftrightarrow \neg B$
    (b) $(A \to B) \wedge (A \to C) \equiv \neg A \vee (A \wedge B \wedge C)$
    (c) $A \to (B \to (C \vee D)) \equiv (A \to C) \vee (B \to D)$.

11. Prove Theorem 2.2.4. **Hint**: Induction on the structure of A.

12. Prove that for any formulas A, B, and C, if $A \models B$, then $A \wedge C \models B \wedge C$.

13. Define all the sixteen binary Boolean functions by formulas using only $\top, \bot$, $\neg$, and $\wedge$.

14. Prove that the `nor` operator $\downarrow$, where $x \downarrow y = \neg(x \vee y)$, can serve as a minimally sufficient set of Boolean operators.

15. Convert the following formulas into equivalent CNFs:

    (a) $p \to (q \wedge r)$;
    (b) $(p \to q) \to r$;
    (c) $\neg(\neg p \vee q) \vee (r \to \neg s)$;
    (d) $p \vee (\neg q \wedge (r \to \neg p))$;
    (e) $\neg(((p \to q) \to p) \to q)$;
    (f) $(p \to q) \leftrightarrow (p \to r)$;

16. Convert the following formulas into equivalent DNFs:

    (a) $p \to (q \wedge r)$;
    (b) $(p \to q) \to r$;
    (c) $\neg(\neg p \vee q) \vee (r \to \neg s)$;
    (d) $p \vee (\neg q \wedge (r \to \neg p))$;
    (e) $\neg(((p \to q) \to p) \to q)$;
    (f) $(p \to q) \leftrightarrow (p \to r)$;

17. Construct full CNF for the following formulas, where $\overline{a}$ is $\neg a$, $+$ is $\vee$ and $\wedge$ is omitted as production.

    (a) $F = \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}\overline{c} + \overline{a}\overline{b}\overline{c}$;

(b) $G = abc + \overline{a}bc + a\overline{b}c + ab\overline{c}$;

(c) $H = a \oplus b \oplus c$;

(d) $I = (a \wedge b) \rightarrow (b \rightarrow c)$.

18. Construct full DNF for the following formulas, where $\overline{a}$ is $\neg a$, $+$ is $\vee$ and $\wedge$ is omitted as production.

(a) $F = (\overline{a} + \overline{b} + c)(\overline{a} + b + \overline{c})(a + \overline{b} + \overline{c})(\overline{a} + \overline{b} + \overline{c})$;

(b) $G = (a + b + c)(\overline{a} + b + c)(a + \overline{b} + c)(a + b + \overline{c})$;

(c) $H = a \oplus b \oplus c$;

(d) $I = (a \wedge b) \rightarrow (b \rightarrow c)$.

19. Assume $a > b > c$, construct ROBDDs for the following formulas, where $'$ is $\neg$, $+$ is $\vee$ and $\wedge$ is omitted as production.

(a) $F = abc + \overline{a}\overline{b}c + a\overline{b}\overline{c}$;

(b) $G = \overline{a}bc + abc + a\overline{b}c$;

(c) $H = a \oplus b \oplus c$;

(d) $I = (a \wedge b) \rightarrow (b \rightarrow c)$.

20. Let $\mathcal{F}$ be the set of all formulas built on the set of n propositional variables. Prove that there are exactly 2^{2^n} classes of equivalent formulas among $\mathcal{F}$.

21. (a) Prove that $\{\rightarrow, \neg\}$ is a minimally sufficient set of Boolean operators.

(b) Prove that $\{\rightarrow, \bot\}$ is a minimally sufficient set of Boolean operators.

22. Prove that $\{\wedge, \vee, \rightarrow, \top\}$ is not a sufficient set of Boolean operators.

23. Prove that (a) $\{\wedge, \vee, \top, \bot\}$ is not a sufficient set of Boolean operators; (b) if we add any function o which cannot be expressed in terms of $\wedge$, $\vee$, $\top$, and $\bot$, then $\{\wedge, \vee, \top, \bot, o\}$ is sufficient.

24. Identify the pairs of equivalent formulas from the following candidates:

$(a)\ p \uparrow (q \uparrow r),\ (b)\ (p \uparrow q) \uparrow r,\ (c)\ \uparrow (p, q, r),\ (d)\ p \downarrow (q \downarrow r),\ (e)\ (p \downarrow q) \downarrow r,$
$(f)\ \downarrow (p, q, r),\ (g)\ p \uparrow (q \downarrow r),\ (h)\ (p \uparrow q) \downarrow r,\ (i)\ p \downarrow (q \uparrow r),\ (j)\ (p \downarrow q) \uparrow r$

25. Try to find an equivalent DNF of the minimal size for the following DNF formulas using K-maps:

(a) $f(A,B,C) = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B + A\,B\,\overline{C} + A\,C$;

(b) $f(A,B,C) = \overline{A}\,B + B\,\overline{C} + B\,C + A\,\overline{B}\,\overline{C}$;

(c) $f(A,B,C,D) = \overline{A}\,\overline{B}\,CD + \overline{A}\,BC + B\,\overline{C}\,\overline{D} + B\,\overline{C}\,D$;

(d) $f(A,B,C,D) = m_0 + m_1 + m_5 + m_7 + m_8 + m_{10} + m_{14} + m_{15}$, where m_i is a full minterm of A, B, C and D, and i is the decimal value of the binary string $ABCD$.

26. Using the K-map method and the procedure *NNF2NOR*(A), to find an equivalent formula, which uses only NAND/NOR/NOT operators, for each of the formulas in the previous question.

27. Try to find an equivalent CNF of the minimal size for the following Boolean functions using K-maps:

(a) $f(A,B,C) = \overline{A}\,\overline{B} + \overline{A}\,B + A\,B\,\overline{C} + A\,C$;

(b) $f(A,B,C) = \overline{A}\,B\,C + A\,B\,\overline{C} + A\,\overline{B}\,C + \overline{A}\,\overline{B}\,\overline{C}$;

(c) $f(A,B,C,D) = m_0 + m_1 + m_3 + m_4 + m_5 + m_7 + m_{12} + m_{13} + m_{15}$;

(d) $f(A,B,C,D) = m_0 + m_1 + m_2 + m_4 + m_5 + m_6 + m_8 + m_9 + m_{12} + m_{13} + m_{14}$.

28. Try to find a minimal and equivalent formula, which uses only NAND/NOR/NOT operators, for each of the formulas in the previous question.

29. A function $f : B^n \longrightarrow B$, where $B = \{0, 1\}$, is called *linear* if $f(x_1, x_2, ..., x_n) = a_0 + a_1x_1 + \cdots + a_nx_n$ for suitable coefficients $a_0, ..., a_n \in B$. Here + denotes the addition modulo 2, and the not written multiplication is the multiplication over integers.

(a) Show that the above representation of a linear function f is unique.

(b) Determine the number of n-ary linear functions.

30. Suppose x is a 32-bit positive integer. Provide a java program for each of the following programming tasks, using as much bit-wise operations as possible:

(a) Check whether x is even or odd.

(b) Get the position of the highest bit of 1 in x.

(c) Get the position of the lowest bit of 1 in x.

(d) Count trailing zeros in x as a binary number.

(e) Shift x right by 3 bits and put the last three bits of x at the beginning of the result from the shifting. The whole process is called "rotate x right" by 3 bits.

(f) Convert a string of decimal digits into a binary number without using multiplication by 10.

31. Suppose a formula A contains 15 variables, each variable is represented by an integer from 0 to 14. We use the last 15 bits of an integer variable x to represent an interpretation of A, such that i^{th} bit is 1 iff we assign 1 to variable i. Modify the algorithm $eval(A, \sigma)$, where σ is replaced by x using bit-wise operations. Design an algorithm which lists all the interpretations of A and apply each interpretation to A using $eval(A, x)$. The pseudo-code of the algorithm should be like a java program.

32. The meaning of the following propositional variables is given below:

t: "taxes are increased",
e: "expenditures rise",
d: "the debt ceiling is raised",
c: "the cost of collecting taxes rises",
g: "the government borrows more money",
i: "interest rates increase".

(a) express the following statements as propositional formulas; (b) convert the formulas into CNF.

> Either taxes are increased or if expenditures rise then the debt ceiling is raised. If taxes are increased, then the cost of collecting taxes rises. If a rise in expenditures implies that the government borrows more money, then if the debt ceiling is raised, then interest rates increase. If taxes are not increased and the cost of collecting taxes does not increase then if the debt ceiling is raised, then the government borrows more money. The cost of collecting taxes does not increase. Either interest rates do not increase or the government does not borrow more money.

33. Specify the following puzzle in propositional logic and convert the specification into CNF. Find the model of your CNF and use this model to construct a solution of the puzzle.

Four kids, Abel, Bob, Carol, and David, are eating lunch on a hot summer day. Each has a big glass of water, a sandwich, and a different type of fruit (apple, banana, orange, and grapes). Which fruit did each child have? The given clues are:

(a) Abel and Bob have to peel their fruit before eating.

(b) Carol doesn't like grapes.

(c) Abel has a napkin to wipe the juice from his fingers.

34. Specify the following puzzle in propositional logic and convert the specification into CNF. Find the model of your CNF and use this model to construct a solution of the puzzle.

Four electric cars are parked at a charge station and their mileages are 10K, 20K, 30K, and 40K, respectively. Their charge costs are \$15, \$30, \$45, and \$60, respectively. Figure out how the charge cost for each car from the following hints:

(a) The German car has 30K miles.

(b) The Japanese car has 20K miles and it charged \$15.

(c) The French car has less miles than the Italian car.

(d) The vehicle that charged \$30 has 10K more mileages than the Italian car.

(e) The car that charged \$45 has 20K less mileage than the German car.

35. Specify the following puzzle in propositional logic and convert the specification into CNF. Find the model of your CNF and use this model to construct a solution of the puzzle.

Four boys are waiting for Thanksgiving dinner and each of them has a unique favorite food. Their names are Larry, Nick, Philip, and Tom. Their ages are all different and fall in the set of $\{8, 9, 10, 11\}$. Find out which food they are expecting to eat and how old they are.

(a) Larry is looking forward to eating turkey.

(b) The boy who likes pumpkin pie is one year younger than Philip.

(c) Tom is younger than the boy that loves mashed potato.

(d) The boy who likes ham is two years older than Philip.

36. Six people sit on a long bench for a group photo. Two of them are Americans (A), two are Canadians (C), one Italian (I), and one Spaniard (S). Suppose we use propositional variables X_y, where $x \in \{A, C, I, S\}$ and $1 \leq y \leq 6$, and X_y is true iff the people of nationality X sits at seat y of the bench. Please specify the problem formally in a propositional formula using X_y, and convert the formula into CNF such that each model of the CNF gives us one solution for all the six people to sit on the bench, enforcing the rules that (a) the people from the same country will not sit beside one another; (b) the Italian must be surrounded by Americans; and (c) each American sits next to a Canadian. Please list all the models of your CNF.

37. A very special island is inhabited only by knights and knaves. Knights always tell the truth, and knaves always lie. You meet two inhabitants: Sally and Zippy. Sally claims, "I and Zippy are not the same." Zippy says, "Of I and Sally, exactly one is a knight." Can you determine who is a knight and who is a knave?

38. A very special island is inhabited only by knights and knaves. Knights always tell the truth, and knaves always lie. You meet two inhabitants: Homer and Bozo. Homer tells you, "At least one of the following is true: that I am a knight or that Bozo is a knight." Bozo claims, "Homer could say that I am a knave." Can you determine who is a knight and who is a knave?

39. A very special island is inhabited only by knights and knaves. Knights always tell the truth, and knaves always lie. You meet two inhabitants: Bart and Ted. Bart claims, "I and Ted are both knights or both knaves." Ted tells you, "Bart would tell you that I am a knave." Can you determine who is a knight and who is a knave?

CHAPTER 3
PROOF PROCEDURES FOR PROPOSITIONAL LOGIC

In Chapter 1, we introduced the concept of "proof procedure": Given a set A of axioms (i.e., the formulas assumed to be true) and a formula B, a proof procedure $P(A, B)$ will answer the question whether B is a theorem of A: If $P(A, B)$ returns true, we say B is proved, if $P(A, B)$ returns false, we say B is disproved. For propositional logic, the axiom set A is regarded to be equivalent to the conjunction of formulas and a theorem is any formula B such that $A \models B$, i.e., $B \in \mathcal{T}(A)$. That is, a proof procedure is also called a *theorem prover*. The same procedure P can be used to show B is valid (i.e., B is a tautology) by calling $P(\top, B)$, or that C is unsatisfiable by calling $P(C, \bot)$. That is, proving the entailment relation is no harder than proving validity or proving unsatisfiability, as the following theorem shows. The three problems are equivalent and belong to the class of co-NP-complete problems. In practice, a proof procedure is designed to solve one of the three problems as its direct goal. A proof procedure is called

- *theorem prover* $P(X, Y)$ if it is to show that formula Y is a theorem (an entailment) of X (assuming a set of formulas is equivalent to the conjunction of all its members);
- *tautology prover* $T(A)$ if it is to show formula A is valid (a tautology);
- *refutation prover* $R(B)$ if it is to show formula B is unsatisfiable.

If we have one of the above three procedures, i.e., $P(X, Y)$, $T(A)$, and $R(B)$, we may implement the other two as follows.

- If $P(X, Y)$ is available, implement $T(A)$ as $P(\top, A)$, and $R(B)$ as $P(B, \bot)$.
- If $T(A)$ is available, implement $P(X, Y)$ as $T(X \rightarrow Y)$, and $R(B)$ as $T(\neg B)$.
- If $R(B)$ is available, implement $P(X, Y)$ as $R(X \wedge \neg Y)$, and $T(A)$ as $R(\neg A)$.

The correctness of the above implementations is based on the following theorem: Given formulas A and B, the following three statements are equivalent:

1. $A \models B$;

2. $A \rightarrow B$ is valid;

3. $A \wedge \neg B$ is unsatisfiable.

Thus, one prover can do all the three, either directly or indirectly. In Chapter 2, we have discussed several concepts and some of them can be used to construct proof procedures as illustrated below.

- Truth table: It can be used to construct a theorem prover, a tautology prover, and a refutation prover.
- CNF: It can be used to construct a tautology prover, as a CNF is valid iff every clause is valid and it is very easy to check if a clause is valid.
- DNF: It can be used to construct a refutation prover, as a DNF is unsatisfiable iff every minterm is unsatisfiable and it is very easy to check if a minterm is unsatisfiable.
- ROBDD or INF: It can be used to construct a tautology prover or a refutation prover, because ROBDD is a canonical form.
- Algebraic substitution: It can be used to construct a tautology prover or a refutation prover, as substituting "equal by equal" preserves the equivalence relation. If we arrive at $\top$ or $\bot$, we can claim the original formula is valid or unsatisfiable.

In terms of its working principle, a proof procedure may have one of three styles: enumeration, reduction, and deduction.

- *enumeration-style*: enumerate all involved interpretations;
- *reduction-style*: use the equivalence relations to transform a set of formulas into a simpler set;
- *deduction-style*: use inference rules to deduce new formulas.

By definition, $A \models B$ iff $\mathcal{M}(A) \subseteq \mathcal{M}(B)$, that is, every model of A is a model of B. We may use a truth table to enumerate all the interpretations involving A and B, to see if every model of A is a model of B. A proof procedure based truth tables is an enumeration-style proof procedure which uses exhaustive search. This type of proof procedures are easy to describe but highly inefficient. For example, if A contains 30 variables, we need to construct a truth table of 2^{30} rows.

Reduction-style proof procedures use the equivalence relations to transform a set of formulas into desired, simplified formulas, such as reduced ordered binary decision diagrams (ROBDD). Since ROBDD is canonical, we decide that a formula is valid if its ROBDD is $\top$ (or 1), or unsatisfiable, if its ROBDD is $\bot$ (or 0). We may also use disjunctive normal forms (DNF) to show a formula is unsatisfiable: Its simplified DNF is $\bot$. In the following, we will introduce the semantic tableau method, which is a reduction-style proof procedure and works in the same way as obtaining DNF from the original formula.

Deduction-style proof procedures use inference rules to generate new formulas, until the desired formula is generated. An inference rule typically takes one or two formulas as input and generates one or two new formulas as output. Some of these rules come from the equivalence relations between the input and the output formulas. In general, we require that the output of an inference rule is entailed by the input. There are many deduction-style proof procedures and in this chapter we will present several of them. The goal of this presentation is not for introducing efficient proof procedures, but for introducing a formalism such that other logics can use. This comment applies to the semantics tableau method, too.

Many proof procedures are not clearly classified as enumeration, reduction, or deduction styles. Yet, typically their working principles can be understood as performing enumeration, or reduction, or deduction, or all actions implicitly. For example, converting a formula into DNF can be a reduction, however, converting a formula into full DNF can be an enumeration, as every term in a full DNF corresponds to one model of the formula. In practice, reduction is often used in most deduction-style proof procedures for efficiency.

In Chapter 1, we say that a proof procedure is decision procedure if the procedure terminates for every input, that is, the procedure is an algorithm. Fortunately for propositional logic, every proof procedure introduced in this chapter is a decision procedure.

3.1 Semantic Tableau

The semantic tableau method, also called the "truth tree" method, is a proof procedure not just for propositional logic, but also for many other logics, including first-order logic, temporal and modal logics. A tableau for a formula is a tree structure, each node of the tree is associated with a list of formulas derived from the original formula. The tableau method constructs this tree and uses it to prove/refute the satisfiability of the original formula. The tableau method can also determine the satisfiability of finite sets of formulas of various logics. It is the most popular

proof procedure for modal logics.

3.1.1 Tableau: A Tree Structure for DNF

For propositional logic, the semantic tableau checks whether a formula is satisfiable or not, by "breaking" complex formulas into smaller ones as we do for transforming the formula into DNF (disjunctive normal form). The method works on a tree whose nodes are labeled with a list of formulas. At each step, we may extend a leaf node by adding one or two successors to this node.

- Each node of the tree is associated with a list of formulas and the comma "," in the list is a synonym to $\wedge$;
- A node is called *closed* if it contains a complementary pair of literals (or formulas), or $\bot$, or $\neg\top$.
- A node is called a *opwn* node if it contains only a set of consistent literals, which is equivalent to a minterm in DNF.
- If a node is neither closed nor open and has no children (a leaf in the current tree), then it is *expandable*. We may apply the transformation rules for DNF on one of the formulas in an expandable node to create children nodes. Note that the rules apply only to the top logical operation of a formula. The expansion stops when the tree has no expandable nodes.
- The tree links indicate the equivalence relation among formulas. If the result of the transformation rule is a disjunction, two successors of the current node are created and the formulas of the current node is equivalent to the disjunction of those formulas in the two successors. This type of the transformation rules is called β-rules; the other rules are called α-rules, which generate only one child and the formulas of the current node is equivalent to those of the child node.

To prove the unsatisfiability of a formula, the method starts by generating the tree and check that every leaf node is closed, that is, no open node and no expandable nodes as leaves. In this case, we say the tree is *closed*.

Example 3.1.1. For the formulas $A = p \wedge (\neg q \vee \neg p)$ and $B = (p \vee q) \wedge (\neg p \wedge \neg q)$, their tableaux are shown in Figure 3.1.1. For A, the tableau has one open node (marked by $\odot$) and one closed node (marked by $\times$). For B, it has only two closed nodes. Thus A is satisfiable as it has an open node; and B is unsatisfiable as its tableau is closed. $\square$

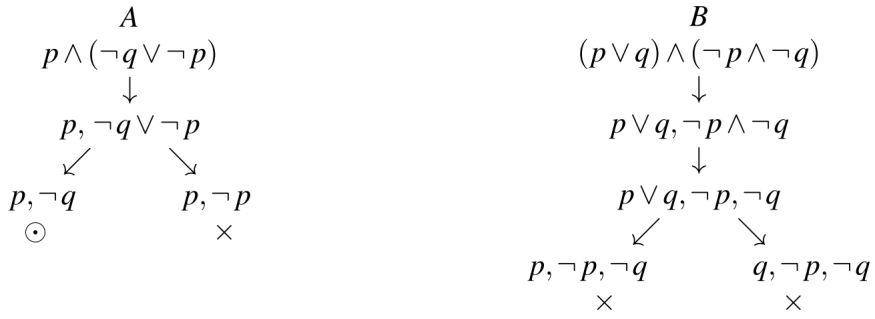


Figure 3.1.1: The tableaux for $A = p \wedge (\neg q \vee \neg p)$ and $B = (p \vee q) \wedge (\neg p \wedge \neg q)$

The tableaux can be also drawn linearly, using proper labels for the parent-child relation: The root node has the empty label. If a node has label x, its first child has label $x1$ and its second child has label $x2$, and so on. For example, we may display the tableaux of A and B in Figure 3.1.1 as follows:

	$p \wedge (\neg q \vee \neg p)$	
1 :	$p, (\neg q \vee \neg p)$	
11 :	$p, \neg q$	$\odot$
12 :	$p, \neg p$	$\times$

(a) Proof of A

	$(p \vee q) \wedge (\neg p \wedge \neg q)$	
1 :	$(p \vee q), (\neg p \wedge \neg q)$	
11 :	$(p \vee q), \neg p, \neg q$	
111 :	$p, \neg p, \neg q$	$\times$
112 :	$q, \neg p, \neg q$	$\times$

(b) Proof of B

The labels of nodes can uniquely identify the positions (or address) of these nodes in a tableau.

3.1.2 α-Rules and β-Rules

Here is the listing of α-rules and β-rules, which apply to the top logical operators in a formula.

α	α_1, α_2
$A \wedge B$	A, B
$\neg(A \vee B)$	$\neg A, \neg B$
$\neg(A \rightarrow B)$	$A, \neg B$
$\neg(A \oplus B)$	$(A \vee \neg B), (\neg A \vee B)$
$A \leftrightarrow B$	$(A \vee \neg B), (\neg A \vee B)$
$\neg(A \uparrow B)$	A, B
$A \downarrow B$	$\neg A, \neg B$
$\neg\neg A$	A

β	β_1	β_2
$\neg(A \wedge B)$	$\neg A$	$\neg B$
$A \vee B$	A	B
$A \rightarrow B$	$\neg A$	B
$A \oplus B$	$A, \neg B$	$\neg A, B$
$\neg(A \leftrightarrow B)$	$A, \neg B$	$\neg A, B$
$A \uparrow B$	$\neg A$	$\neg B$
$\neg(A \downarrow B)$	A	B

If we start with a formula in NNF (negation normal form), all the α-rules and β-rules with the leading symbol $\neg$ are not needed.

For every α-rule, we transform formula α into α_1, α_2; for every β-rule, we transform formula β into β_1 and β_2. It is easy to check that $\alpha \equiv \alpha_1 \wedge \alpha_2$ and $\beta \equiv \beta_1 \vee \beta_2$. Those rules are what we used for transforming a formula into DNF. Since a tableau maintains the property that the formulas of the current node is equivalent to the disjunction of the formulas in its children nodes (the formulas in the same node are considered in conjunction), a tableau is a tree-like representation of a formula that is a disjunction of conjunctions, i.e., a DNF.

Example 3.1.2. To show that Frege's formula, $(p \to (q \to r)) \to (p \to q) \to (p \to r)$, is valid, we show that the semantic tableau of its negation is closed.

$$
\begin{array}{rcl}
 & \neg((p \to (q \to r)) \to (p \to q) \to (p \to r)) & \alpha\ \neg\ \to \\
1: & (p \to (q \to r)), \neg(p \to q) \to (p \to r) & \alpha\ \neg\ \to \\
11: & (p \to (q \to r)), (p \to q), \neg(p \to r) & \alpha\ \neg\ \to \\
111: & (p \to (q \to r)), (p \to q), p, \neg r & \beta\ \to \\
1111: & \neg p, (p \to q), p, \neg r & \times \\
1112: & (q \to r), (p \to q), p, \neg r & \beta\ \to \\
11121: & (q \to r), \neg p, p, \neg r & \times \\
11122: & (q \to r), q, p, \neg r & \beta\ \to \\
111221: & \neg q, q, p, \neg r & \times \\
111222: & r, q, p, \neg r & \times
\end{array}
$$

□

In Chapter 1, we pointed out that there are two important properties concerning a proof procedure: soundness and completeness. For a refutation prover like semantic tableau, the soundness means that if the procedure says the formula is unsatisfiable, then the formula must be unsatisfiable. The completeness means that if the formula is unsatisfiable, then the procedure should be able to give us this result.

Theorem 3.1.3. *The semantic tableau is a decision procedure for propositional logic.*

Proof. Suppose the list of formulas at one node is α, β, and γ and the formula represented by this node is $A = \alpha \wedge \beta \wedge \gamma$. If α is transformed by a α-rule to α_1 and α_2, then from $\alpha \equiv \alpha_1 \wedge \alpha_2$, $A \equiv \alpha_1 \wedge \alpha_2 \wedge \beta \wedge \gamma$. If *beta* is transformed by a β-rule to β_1 and β_2, then from $\beta \equiv \beta_1 \vee \beta_2$, $A \equiv (\alpha \wedge \beta_1 \wedge \gamma) \vee (\alpha \wedge \beta_2 \wedge \gamma)$. In other words, the logical equivalence is maintained for every parent-child link in the semantic tableau. By a simple induction on the structure of the tree, the formula at the root of the

tree is equivalent to the disjunction of all the formulas represented at the leaf nodes. That is, the procedure modifies the tableau in such a way that the disjunction of the formulas represented by the leaf nodes of the resulting tableau is equivalent to the original one. One of these conjunctions may contain a pair of complementary literals, in which case that conjunction is equivalent to false. If all conjunctions are equivalent to false, the original formula is unsatisfiable. This shows the soundness of the procedure.

If a formula is unsatisfiable, then every term in its DNF must be equivalent to false and the corresponding node in the tableau is closed. Thus we will have a closed tableau. This shows the completeness of the procedure.

Since neither α-rules nor β-rules can be used forever (each rule reduces either one occurrence of a binary operator or two negations), the procedure must be terminating. □

Of course, semantic tableau can be used to show the satisfiability of a formula. If there is an open node in the tableau, we may assign 1 to every literal in this node so that the formula represented by this node is true under this interpretation. The original formula will be true under this interpretation because it is equivalent to the disjunction of the formulas of the leaf nodes. Thus, this interpretation is a model of the original formula. This shows that the procedure can be used to find a model when the formula is satisfiable.

Semantic tableau specifies what rules to be used but it does not specify which rule should be used first or which node should be expanded first. In general, α-rules are better be applied before β-rules, so that the number of nodes in a tree can be reduced. In practice, the order of formulas in a list can be changed and the order of β_1 and β_2 can be switched. This gives the user the freedom to construct trees of different shapes, or to find an open node quicker. The procedure may terminate once an open node is found, if the goal is to show that the original formula is satisfiable.

Semantic tableaux are much more expressive and easy to use than truth-tables, though that is not the reason for their introduction. Sometimes, a tableau uses more space than a truth table. For example, an unsatisfiable full CNF of n variables will generate a tableau of more than $n!$ nodes, which is larger than a truth table of 2^n lines. The beauty of semantic tableaux lies on the simplicity of presenting a proof procedure using a set of α and β rules. We will see in later chapters how new rules are added into this procedure so that a proof procedure for other logics can be obtained.

3.2 Deductive Systems

In logic, a deductive system $\mathcal{S}$ consists of a set of inference rules, where each rule takes premises as input and returns a conclusion (or conclusions) as output. Popular inference rules in propositional logic include modus ponens (MP), modus tollens (MT), and contraposition (CP), which can be displayed, respectively, as follows:

$$\frac{A \to B \quad A}{B}\ (\text{MP}) \qquad \frac{A \to B \quad \neg B}{\neg A}\ (\text{MT}) \qquad \frac{A \to B}{\neg B \to \neg A}\ (\text{CP})$$

3.2.1 Inference Rules and Proofs

In general, an inference rule can be specified as

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_k}{C}$$

where $P_1, P_2, ..., P_k$ are the premises and C is the conclusion. We may also write it as

$$P_1, P_2, \cdots, P_k \vdash C$$

If the premises are empty ($k = 0$), we say this inference rule is an *axiom rule*.

An inference rule is *sound* if its premises entail the conclusion, that is,

$$P_1, P_2, \cdots, P_k \models C$$

for every inference rule $\{P_1, P_2, \cdots, P_k\} \vdash C$ (or equivalently, $(P_1 \wedge P_2 \wedge \cdots \wedge P_k) \to C$ is a tautology). For example, MP (modus ponens) is sound because $\{A \to B, A\} \models B$; modus tollens is sound because $\{A \to B, \neg B\} \models \neg A$.

Given a set A of axioms, which are the formulas assumed to be true, a *proof* of formula B in $\mathcal{S}$ is a sequence of formulas $F_1, F_2, ..., F_n$ such that $F_n = B$ and each F_i is either a formula in A or can be generated by an inference rule of $\mathcal{S}$, using the formulas before F_i in the sequence as the premises. If such a proof exits, we denote it by $A \vdash_{\mathcal{S}} B$; the subscript $\mathcal{S}$ can be dropped if $\mathcal{S}$ is understood from the context. The proof procedure $P(A, B)$ based on $\mathcal{S}$ is simply to show $A \vdash_{\mathcal{S}} B$. Obviously, this is a deduction-style proof procedure.

Example 3.2.1. Let $A = \{p \to (q \to r), p, \neg r\}$, and $\mathcal{S} = \{MP, CT\}$. A proof of $A \vdash_{\mathcal{S}} \neg q$ is given below:

1.	$p \to (q \to r)$	assumed
2.	p	assumed
3.	$q \to r$	MP, 1, 2
4.	$\neg r$	assumed
5.	$\neg q$	MT, 3, 4

□

The formulas in a proof can be rearranged so that all axioms appear in the beginning of the sequence. For example, the above proof can be rewritten as the following.

1.	$p \to (q \to r)$	assumed
2.	p	assumed
3.	$\neg r$	assumed
4.	$q \to r$	MP, 1, 2
5.	$\neg q$	MP, 3, 4

Every proof can be represented by a directed graph where each node is a formula in the proof and if a formula B is derived from A_1, A_2, ..., A_k by an inference rule, then there are edges (A_i, B), $1 \leq i \leq k$, in the graph. This proof graph must be acyclic as we assume that no formula can be removed from a proof without destroying the proof. Every topological sort of the nodes of this graph should give us a linear proof. Since every acyclic direct graph (DAG) can be converted into a tree (by duplicating some nodes) and still preserve the parent-child relation, we may present a proof by a tree as well. The proof tree for the above proof example is given in Figure 3.2.1. It is clear that all assumed formulas (axioms) do not have the incoming edges. The final formula in the proof does not have outgoing edges. All intermediate formulas have both incoming and outgoing edges.

The soundness of an inference system comes from the soundness of all the inference rules. In general, we require all inference rules be sound to preserves truth of all the derived formulas. For a sound inference system $\mathcal{S}$, every derived formula $B \in \mathcal{T}(A)$. If A is a set of tautologies, then B is a tautology, too.

Theorem 3.2.2. *If an inference system $\mathcal{S}$ is sound and $A \vdash_{\mathcal{S}} B$, then $A \models B$.*

Proof. $A \vdash_{\mathcal{S}} B$ means there is a proof of B in $\mathcal{S}$. Suppose the proof is $F_1, F_2, ..., F_n = B$. By induction on n, we show that $A \models F_i$ for all $i = 1, ..., n$. If F_i is in A, then $A \models F_i$. If F_i is derived from an inference rule $P_1, P_2, ..., P_m \vdash C$, using $F_{j_1}, F_{j_2}, ..., F_{j_m}$ as premises, by induction hypotheses, $A \models F_{j_k}$, because $j_k < i$, for all $1 \leq k \leq m$. That is, $A \models \wedge_{k=1}^{m} F_{j_k}$.

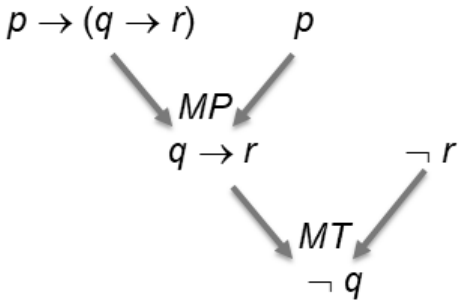


Figure 3.2.1: The proof tree of $\neg q$

Since the inference rule is sound, $\wedge_{k=1}^{m} P_k \models C$. Applying the substitution theorem, $\wedge_{k=1}^{m} F_{j_k} \models F_i$. Because $\models$ is transitive, $A \models F_i$. □

Different inference systems are obtained by changing the axioms or the inference rules. In propositional logic, all these systems are equivalent in the sense that they are sound and complete.

3.2.2 Hilbert Systems

Hilbert systems, sometimes called Hilbert calculus, or Hilbert-style inference systems, are a type of deduction system attributed to Gottlob Frege and David Hilbert. Variants of Hilbert systems for propositional logic exist and these deductive systems are extended to first-order logic and other logics.

Definition 3.2.3. *The Hilbert system $\mathcal{H}$ contains three axiom rules plus MP (modus ponens):*

$$\begin{aligned} H_1: &\ \vdash\ (A \to (B \to A)),\\ H_2: &\ \vdash\ (A \to (B \to C)) \to ((A \to B) \to (A \to C)),\\ H_3: &\ \vdash\ (\neg B \to \neg A) \to (A \to B),\\ \text{MP}: &\ A, A \to B \vdash B. \end{aligned}$$

The axiom rule H_3 is not needed if $\to$ is the only operator in all formulas. Other logical operators can be defined using either $\to$ and $\neg$. For example, $A \wedge B \equiv \neg(A \to \neg B)$ and $A \vee B \equiv \neg A \vee B$. $\neg A$ can be replaced by $A \to \bot$ in H_3. It will turn out that all tautologies are derivable from the three formulas H_1, H_2, and H_3.

Example 3.2.4. Assume $A = \emptyset$, to show that $A \vdash_{\mathcal{H}} (p \to p)$, we have the following

proof in $\mathcal{H}$:

1.	$p \to ((p \to p) \to p)$	$H_1, A = p, B = (p \to p)$
2.	$(p \to ((p \to p) \to p)) \to ((p \to (p \to p)) \to (p \to p))$	$H_2, A = C = p, B = p \to p$
3.	$(p \to (p \to p)) \to (p \to p)$	MP from $1, 2$
4.	$(p \to (p \to p))$	$H_1, A = p, B = p$
5.	$p \to p$	MP on $3, 4, A = p, B = p$.

□

Proposition 3.2.5. *$\mathcal{H}$ is sound: If $A \vdash_{\mathcal{H}} B$, then $A \models B$.*

Proof. It is easy to check that every inference rule of $\mathcal{H}$ is sound. That is, for every axiom rule H_i, $i = 1, 2, 3$, the result of the rule is a tautology. For MP, $(p \wedge (p \to q)) \to q$ is a tautology. By Theorem 3.2.2, the proposition holds. □

Hilbert systems originally assumed that $A = \emptyset$ (or $A = \top$), thus every derived formula is a tautology in a proof. This restriction is relaxed today so that we may take a set of formulas as axioms. The truth of the formulas becomes conditional on the truth of the axioms.

Example 3.2.6. Assume $A = \{p \to q, q \to r\}$, to show that $A \vdash_{\mathcal{H}} (p \to r)$, we have the following proof in $\mathcal{H}$:

1.	$p \to q$	assumed
2.	$q \to r$	assumed
3.	$((q \to r) \to (p \to (q \to r))$	H_1
4.	$p \to (q \to r)$	MP from $2, 3$
5.	$(p \to (q \to r)) \to ((p \to q) \to (p \to r))$	H_2
6.	$(p \to q) \to (p \to r)$	MP from $4, 5$
7.	$p \to r$	MP from $1, 6$

□

Since $\mathcal{H}$ is sound, from $A \vdash_{\mathcal{H}} (p \to r)$, we have $(p \to q) \wedge (q \to r) \models_{\mathcal{H}} (p \to r)$, or $((p \to q) \wedge (q \to r)) \to (p \to r)$ is valid.

Hilbert system is one of the earliest inference systems and has great impact to the creation of many inference systems. However, theorem proving in this system has always been considered a challenge because of its high complexity. Since Hilbert systems can hardly be a practical tool for problem solving, we state without proof the completeness result of Hilbert system.

Theorem 3.2.7. *The Hilbert system is sound and complete, that is, $A \models B$ iff $A \vdash_{\mathcal{H}} B$.*

3.2.3 Natural Deduction

Both of semantic tableaux and Hilbert systems for constructing proofs have their disadvantages. Hilbert's systems are difficult to construct proofs in; their main uses are metalogical, the small number of rules making it easier to prove results about logic. The tableau method on the other hand is easy to use mechanically but, because of the form of the connective rules and the fact that a tableau starts from the negation of the formula to be proved, the proofs that result are not a natural sequence of easily justifiable steps. Likewise, very few proofs in mathematics are from axioms directly. Mathematicians in practice usually reason in a more flexible way.

Natural deduction is a deductive system in which logical reasoning is expressed by inference rules closely related to the "natural" way of reasoning. This contrasts with Hilbert systems, which instead use axiom rules as much as possible to express the logical laws of deductive reasoning. Natural deduction in its modern form was independently proposed by the German mathematician Gerhard Gentzen in 1934. In Gentzen's natural deductive system, a formula is represented by a set A of formulas, $A = \{A_1, A_2, ..., A_n\}$, where the comma in A are understood as $\vee$. To avoid the confusion, we will write A as

$$(A_1 \mid A_2 \mid \cdots \mid A_n).$$

A is called a *sequent* by Gentzen. If $n = 0$, i.e., $A = (\,)$, which is equivalent to $\bot$ or $(\bot)$. Since we assume that A is a set, duplicated formulas are removed and the order of formulas in A is irrelevant.

Definition 3.2.8. *A natural deductive system $\mathcal{G}$ consists of the following three inference rules, plus the following logical rules which guide the introduction and elimination of logical operators in the sequents.*

name	**inference rule**
axiom	$\vdash \quad (A \mid \neg A \mid \alpha)$
cut	$(A \mid \alpha), \quad (\neg A \mid \beta) \quad \vdash \quad (\alpha \mid \beta)$
thinning	$(\alpha) \quad \vdash \quad (A \mid \alpha)$

op	**introduction**	**elimination**
$\neg$	$(A \mid \alpha) \vdash (\neg\neg A \mid \alpha)$	$(\neg\neg A \mid \alpha) \vdash (A \mid \alpha)$
$\vee$	$(A \mid B \mid \alpha) \vdash (A \vee B \mid \alpha)$	$(A \vee B \mid \alpha) \vdash (A \mid B \mid \alpha)$
$\wedge$	$(A \mid \alpha),\ (B \mid \alpha) \vdash (A \wedge B \mid \alpha)$	$(a)\ (A \wedge B \mid \alpha) \vdash (A \mid \alpha)$ $(b)\ (A \wedge B \mid \alpha) \vdash (B \mid \alpha)$
$\rightarrow$	$(\neg A \mid B \mid \alpha) \vdash (A \rightarrow B \mid \alpha)$	$(A \rightarrow B \mid \alpha) \vdash (\neg A \mid B \mid \alpha)$
$\neg\vee$	$(\neg A \mid \alpha),\ (\neg B \mid \alpha) \vdash (\neg(A \vee B) \mid \alpha)$	$(a)\ (\neg(A \vee B) \mid \alpha) \vdash (\neg A \mid \alpha)$ $(b)\ (\neg(A \vee B) \mid \alpha) \vdash (\neg B \mid \alpha)$
$\neg\wedge$	$(\neg A \mid \neg B \mid \alpha) \vdash (\neg(A \wedge B) \mid \alpha)$	$(\neg(A \wedge B) \mid \alpha) \vdash (\neg A \mid \neg B \mid \alpha)$
$\neg\rightarrow$	$(A \mid \alpha),\ (\neg B \mid \alpha) \vdash (\neg(A \rightarrow B) \mid \alpha)$	$(a)\ (\neg(A \rightarrow B) \mid \alpha) \vdash (A \mid \alpha)$ $(b)\ (\neg(A \rightarrow B) \mid \alpha) \vdash (\neg B \mid \alpha)$

where α and β, possibly empty, denote the rest formulas in a sequent.

In natural deduction, a sequent is deduced from a set of axioms by applying inference rules repeatedly. Each sequent is inferred from other sequents on earlier lines in a proof according to inference rules, giving a better approximation to the style of proofs used by mathematicians than Hilbert systems.

Example 3.2.9. The following is a proof of $\neg(p \wedge q) \rightarrow (\neg p \vee \neg q)$ in $\mathcal{G}$:

1. $(p \mid \neg p \mid \neg q)$ — axiom, $A = p, \alpha = \neg q$
2. $(q \mid \neg q \mid \neg p)$ — axiom, $A = q, \alpha = \neg p$
3. $(p \wedge q \mid \neg p \mid \neg q)$ — $\wedge_I$ from $1, 2, A = p, B = q, \alpha = \neg p \mid \neg q$
4. $(\neg\neg(p \wedge q) \mid \neg p \mid \neg q)$ — $\neg_I$ from $4, A = (p \wedge q), \alpha = (\neg p \vee \neg q)$
5. $(\neg\neg(p \wedge q) \mid (\neg p \vee \neg q))$ — $\vee_I$ from $3, A = \neg p, B = \neg q, \alpha = \neg\neg(p \wedge q)$
6. $(\neg(p \wedge q) \rightarrow (\neg p \vee \neg q))$ — $\rightarrow_I$ from $5, A = \neg(p \wedge q), B = (\neg p \vee \neg q), \alpha = \bot$

□

Proposition 3.2.10. *The deductive system $\mathcal{G}$ is sound, i.e., if $A \vdash_{\mathcal{G}} B$, then $A \models B$.*

Proof. It is easy to check that every inference rule of $\mathcal{G}$ is sound. In particular, the rules of operator introduction preserve the logical equivalence. In the rules of operator elimination, some sequents generate two results. For example, $(A \wedge B \mid \alpha)$ infers both (a) $(A \mid \alpha)$ and (b) $(B \mid \alpha)$. If we bind the two results together by $\wedge$, these rules preserve the logical equivalence, too. On the other hand, the cut rule and the thinning rule do not preserve the logical equivalence; only the entailment relation holds. That is, for the cut rule, $(A \mid \alpha), (\neg A \mid \beta) \vdash (\alpha \mid \beta)$, we have

$$(A \vee \alpha) \wedge (\neg A \vee \beta) \models \alpha \vee \beta,$$

and for the thinning rule, $(\alpha) \vdash (A \mid \alpha)$, we have $\alpha \models A \vee \alpha$. □

Note that the proof in the example above uses only the axiom rule (twice) and the introduction rules for logical operators. This is the case when we are looking for a proof of a tautology. Recall how we would prove that $\neg(p \wedge q) \rightarrow (\neg p \vee \neg q)$ is a tautology by semantic tableau. Since semantic tableau is a refutation prover, we need to show that the negation of a tautology has a closed tableau.

$a.$	:	$\neg(\neg(p \wedge q) \rightarrow (\neg p \vee \neg q))$	$\alpha\ \neg\rightarrow$
$b.$	:	$\neg(p \wedge q), \neg(\neg p \vee \neg q)$	$\alpha\ \neg\vee$
$c.$	:	$\neg(p \wedge q), \neg\neg p, \neg\neg q$	$\alpha\ \neg$
$d.$	:	$\neg(p \wedge q), p, \neg\neg q$	$\alpha\ \neg$
$e.$	:	$\neg(p \wedge q), p, q$	$\beta\ \neg\wedge$
$f.$	1 :	$\neg p, p, q$	$\times$
$g.$	2 :	$\neg q, p, q$	$\times$

Comparing the proof of $\mathcal{G}$ and the tableau above, we can see a clear correspondence: Closed nodes (f) and (g) correspond to (1) and (2), which are axioms; (e) to (3), (c) and (d) to (4), (b) to (5), and finally (a) to (6), respectively. That is, the formula represented by each sequent in a proof of $\mathcal{G}$ can always find its negation in the corresponding node of the semantic tableau.

We show below that $\mathcal{G}$ can be used as a refutation prover by the same example.

Example 3.2.11. Let $A = \{\neg(\neg(p \wedge q) \rightarrow (\neg p \vee \neg q))\}$. A proof of $A \models \bot$ in $\mathcal{G}$ is given below.

1.	$(\neg(\neg(p \wedge q) \rightarrow (\neg p \vee \neg q)))$	assumed
2.	$(\neg(p \wedge q))$	$\neg\rightarrow_E (a)$ from $1, A = \neg(p \wedge q)$
3.	$(\neg(\neg p \vee \neg q))$	$\neg\rightarrow_E (b)$ from $1, B = (\neg p \vee \neg q)$
4.	$(\neg p \mid \neg q)$	$\neg\wedge_E$ from $2, A = p, B = q$
5.	$(\neg\neg p)$	$\neg\vee_E (a)$ from $3, A = \neg p$
6.	$(\neg\neg q)$	$\neg\vee_E (b)$ from $3, B = \neg q$
7.	(p)	$\neg_E$ from $5, A = p$
8.	(q)	$\neg_E$ from $6, A = q$
9.	$(\neg q)$	cut from $4, 7, A = p, \alpha = \bot, \beta = \neg q$
10.	$(\,)$	cut from $8, 9, A = q, \alpha = \bot, \beta = \bot$

□

The above is indeed a proof of $A \vdash_{\mathcal{G}} \bot$, where $\bot$ is used for the empty sequent. In this proof, the used inference rules are the cut rule and the rules for eliminating logical operators. If we apply repeatedly the rules of operator elimination to a formula, we will obtain a list of sequents, each of which represents a clause. In

other words, these rules transform the formula into CNF, then the cut rule works on them. The cut rule for clauses has another name, i.e., *resolution*, which is the topic of the next section.

Indeed, $\mathcal{G}$ contain more rules than necessary: Using only the axiom rule and the rules for operator introduction, it can imitate the dual of a semantic tableau upside down. Using only the cut rule and the rules for operator elimination, it can do what a resolution prover can do. Both semantic tableau and resolution prover are decision procedures for propositional logic. The implication is that natural deduction $\mathcal{G}$ is a decision procedure, too. We give the main result without a proof.

Theorem 3.2.12. *The natural deduction $\mathcal{G}$ is sound and complete, that is, $A \models B$ iff $A \vdash_{\mathcal{G}} B$.*

3.2.4 Inference Graphs

We have seen two examples of deductive systems, Hilbert system $\mathcal{H}$ and Gentzen's natural deduction $\mathcal{G}$. To show the completeness of these deductive systems, we need strategies on how to use the inference rules in these systems and the completeness depends on these strategies. To facilitate the discussion, let's introduce a few concepts.

Definition 3.2.13. *Given an inference system $\mathcal{S}$, the* inference graph *of $\mathcal{S}$ over the axiom set A is defined as a directed graph $G = (V, E)$, where each vertex of V is a set $\mathbf{x}$ of formulas, where $A \in V$, $\mathbf{x} \subseteq \mathcal{T}(A)$, and $(\mathbf{x}, \mathbf{y}) \in E$ iff $\mathbf{y} = \mathbf{x} \cup \{C\}$, where C is the conclusion of an inference rule $\mathbf{r} \in \mathcal{S}$ using some formulas in $\mathbf{x}$ as the premises of $\mathbf{r}$.*

The graph $G = (V, E)$ defines the search space for $A \vdash_{\mathcal{S}} B$, which becomes a search problem: To find a proof of $A \vdash_{\mathcal{S}} B$ is the same as to find a path from A to a node of V containing B. That is, if all the axioms of A in a proof appear in the beginning of the proof sequence, then this proof presents a path in the inference graph. Suppose the proof is a sequence of formulas $F_1, F_2, ..., F_n$ such that the first k formulas are from A, i.e., $F_i \in A$ for $1 \leq i \leq k$ and $F_i \notin A$ for $i > k$. Then this proof represents a directed path in $\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-k}}$ in the inference graph such that $\mathbf{x_0} = A$, $F_i \in A$ for $1 \leq i \leq k$, and $F_j \in \mathbf{x_{j-k}}$ for $k < j \leq n$. That is, a proof is a succinct way of presenting a directed path in the inference graph G.

Example 3.2.14. In Example 3.2.1, the second proof is copied here:

1.	$p \to (q \to r)$	assumed
2.	p	assumed
3.	$\neg r$	assumed
4.	$q \to r$	MP, 1, 2
5.	$\neg r \to \neg q$	CT, 4
6.	$\neg q$	MP, 3, 5

This proof corresponds to the path $\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \mathbf{x_4}$ in the inference graph, where $\mathbf{x_0} = \{1., 2., 3.\}$, $\mathbf{x_1} = \mathbf{x_0} \cup \{4.\}$, $\mathbf{x_2} = \mathbf{x_1} \cup \{5.\}$, $\mathbf{x_3} = \mathbf{x_2} \cup \{5.\}$, and $\mathbf{x_4} = \mathbf{x_3} \cup \{6.\}$. □

Given a set of inference rules, a proof procedure can be easily constructed if we use a *fair strategy* to search the inference graph. For example, starting from A in the inference graph, a breadth-first search is a fair strategy, but a depth-first search is not if the graph is infinite. Fair strategy requires that if an inference rule is applicable at some point, this application will happen over time. Any fair search strategies, heuristic search, best-first search, A*, etc., can be used for a proof.

If every inference rule is sound, the corresponding proof procedure will be sound. Extra requirement like fair strategy is needed in general to show that the proof procedure is complete or halting. If the inference graph is finite, the termination of the proof procedure is guaranteed, but it does not imply the completeness of the procedure.

3.3 Resolution

In the previous section, we introduced two deductive systems, Hilbert systems and natural deduction. These two systems are exemplary for the development of many deductive systems, but are not practical for theorem proving in propositional logic. One of the reasons is the use of axiom rules which provide infinite possibilities because one rule may have infinite instances. Another reason is that, unlike reduction rules, the inference rules may be applied without termination. For automated theorem proving, we need a deductive system which contains very few rules as computers can do simple things fast. Resolution $\mathcal{R}$ is such a system, which contains a single inference rule, called *resolution*, which is a special case of the cut rule when all sequents are clauses.

3.3.1 Resolution Rule

To use the resolution rule, the formula A is first transformed into CNF, which is a conjunction of clauses (disjunction of literals). We will use $A = \{C_1, C_2, ..., C_m\}$

to denote the conjunction of the clauses; each clause C_i is denoted by

$$(l_1 \mid l_2 \mid \cdots \mid l_k),$$

where each l_j is a literal (propositional variable or its negation). If $k = 0$, C_i is the *empty clause*, denoted by $\bot$; if $k = 1$, C_i is called a *unit clause*; if $k = 2$, C_i is called a *binary clause*. A clause is said to be *positive* if it is not empty and contains only positive literals; it is *negative* if it is not empty and contains only negative literals. A clause is a tautology if it contains a pair of complementary literals, i.e., it contains both p and $\neg p$ for some variable p.

The total number of clauses on n propositional variables is bound by 3^n, because each variable has one of three cases in a non-tautology clause: missing, as a positive literal, or as a negative literal.

Suppose clause C_1 is $p \mid \alpha$ and C_2 is $\neg p \mid \beta$, where α and β are the rest literals in C_1 and C_2, respectively, resolution may produce a new clause from C_1 and C_2. Formally, the resolution rule is defined by the following schema, where A is a propositional variable.

$$\frac{(A \mid \alpha) \qquad (\neg A \mid \beta)}{(\alpha \mid \beta)}$$

Alternatively, we may write the resolution rule as

$$(A \mid \alpha),\ (\neg A \mid \beta) \quad \vdash \quad (\alpha \mid \beta).$$

The clause $(\alpha \mid \beta)$ produced by the resolution rule is called *resolvent* by resolving off A from C_1 and C_2. C_1 and C_2 are the *parents* of the resolvent. This resolution rule is also called *binary resolution*, as it involves two clauses as premises.

The resolution rule for propositional logic can be traced back to Davis and Putnam (1960). Resolution is extended to first-order logic in 1965 by John Alan Robinson, using the unification algorithm. If we use $\rightarrow$ instead of $\mid$ in the resolution rule, we obtain

$$\frac{(\neg\alpha \rightarrow A) \qquad (A \rightarrow \beta)}{(\neg\alpha \rightarrow \beta)}$$

which states the transitivity of $\rightarrow$. If α is empty, then we obtain MP:

$$\frac{A \qquad A \rightarrow \beta}{\beta}$$

Example 3.3.1. Let $C = \{(p \mid q),\ (p \mid \neg q),\ (\neg p \mid q \mid r),\ (\neg p \mid \neg q),\ (\neg r)\}$. A proof of $A \vdash_{\mathcal{R}} \bot$ is given below:

1.	$(p \mid q)$	assumed
2.	$(p \mid \neg q)$	assumed
3.	$(\neg p \mid q \mid r)$	assumed
4.	$(\neg p \mid \neg q)$	assumed
5.	$(\neg r)$	assumed
6.	$(q \mid r)$	$\mathcal{R}$ from $1, 3, A = p, \alpha = q, \beta = (q \mid r)$
7.	$(\neg q)$	$\mathcal{R}$ from $2, 4, A = p, \alpha = \neg q, \beta = \neg q$
8.	(r)	$\mathcal{R}$ from $6, 7, A = q, \alpha = r, \beta = \bot$
9.	$(\,)$	$\mathcal{R}$ from $8, 5, A = r, \alpha = \bot, \beta = \bot$

□

Note that we assume that duplicated literals are removed and the order of literals is irrelevant in a clause. Thus, the resolvent generated from clauses 1 and 3 is $(q \mid q \mid r)$, but we keep it as $(q \mid r)$ by removing a copy of q. From clauses 1 and 4, we may obtain a resolvent $(q \mid \neg q)$ on p, or another resolvent on $(p \mid \neg p)$ on q. These two resolvents are tautologies and are not useful in the search for $\bot$. You cannot resolve on two variables at the same time to get $(\,)$ directly from clauses 1 and 4.

Definition 3.3.2. *A* resolution proof *P from the set of input clauses C is a refutation proof of $C \vdash_{\mathcal{R}} \bot$ and the length of P is the number of resolvents in P, denoted by $|P|$. The set of the input clauses appearing in P is called the* core *of P, denoted by* $\text{core}(P)$.

The length of the resolution proof in Example 3.3.1 is 4, which is the length of the list minus the number of assumed clauses in the proof. The core of this proof is the entire input C.

Proposition 3.3.3. *The resolution $\mathcal{R}$ is sound, i.e., if $C \vdash_{\mathcal{R}} B$, then $C \models B$.*

Proof. Using truth table, it is easy to check that $((A \vee \alpha) \wedge (\neg A \vee \beta)) \to (\alpha \vee \beta)$ is a tautology. Thus, the resolution rule is sound. The proposition holds following Theorem 3.2.2. □

Corollary 3.3.4. *If $C \vdash_{\mathcal{R}} \bot$, then C is unsatisfiable.*

Proof. Since $C \vdash_{\mathcal{R}} \bot$ means $C \models \bot$, so $\mathcal{M}(C) = \mathcal{M}(C \wedge \bot) = \mathcal{M}(\bot) = \emptyset$ (see Corollary 2.2.26). □

This corollary allows us to design a refutation prover: To show A is valid, we convert $\neg A$ into an equivalent set C of clauses and show that $C \vdash_{\mathcal{R}} \perp$, where C is called the *input clauses*. Since the total number of clauses is finite, the inference graph of $\mathcal{R}$ is finite. It means that the proof procedure will terminate but it does not mean the graph is small. Before we establish the completeness of $\mathcal{R}$, let us consider how to build an effective resolution prover.

The successful implementation of a resolution prover needs the integration of search strategies that reduce the search space by pruning unnecessary deduction paths. Some strategies remove redundant clauses as soon as they appear in a derivation. Some strategies avoid generating redundant clauses in the first place. Some strategies sacrifice the completeness for efficiency.

3.3.2 Resolution Strategies

Here are some well-known resolution strategies which add restriction to the use of the resolution rule.

- **unit resolution**: One of the two parents is a unit clause.
- **input resolution**: One of the two parents is an input clause (given as the axioms).
- **ordered resolution**: Given an order on propositional variables, the resolved variable must be maximal in both parent clauses.
- **positive resolution**: One of the two parents is positive.
- **negative resolution**: One of the two parents is negative.
- **set-of-support**: The input clauses are partitioned into two sets, S and T, where S is called the set of support, and a resolution is not allowed if both parents are in T.
- **linear resolution**: The latest resolvent is used as a parent for the next resolution (no restrictions on the first resolution).

A resolution proof is called X *resolution proof* if every resolution in the proof is an X resolution, where X is "unit", "input", "ordered", "positive", "negative", "set-of-support", or "linear".

Example 3.3.5. Let $C = \{1.\ (p \mid q),\ 2.\ (\neg p \mid r),\ 3.\ (p \mid \neg q \mid r),\ 4.\ (\neg r)\}$. In the following proofs, the input clauses are omitted and the parents of each resolvent are shown as a pair of numbers following the resolvent.

	(a)			(b)			(c)	
5.	$(\neg p)$	$(2,4)$	5.	$(p \mid r)$	$(1,3)$	5.	$(q \mid r)$	$(1,2)$
6.	$(p \mid \neg q)$	$(3,4)$	6.	(r)	$(2,5)$	6.	$(\neg q \mid r)$	$(2,3)$
7.	$(\neg q)$	$(5,6)$	7.	$(\,)$	$(4,6)$	7.	(r)	$(5,6)$
8.	(q)	$(1,5)$				8.	$(\,)$	$(4,7)$
9.	$(\,)$	$(7,8)$						

Note that (a) is a unit and negative resolution proof; (b) is an input, positive and linear resolution proof; (c) is an ordered resolution proof, assuming $p > q > r$.

The length of (a) is 5; the length of (b) is 3; and the length of (c) is 4. It is easy to check that (a) is not an input resolution proof, because clause 7 is not obtained by input resolution. (b) is not a unit resolution proof. (c) is neither a unit nor an input resolution; it is neither positive nor negative resolution proof. □

Efficiency is an important consideration in automated reasoning and one may sometimes be willing to trade completeness for speed. In unit resolution, one of the parent clauses is always a literal; in input resolution, one of the parent clauses is always selected from the original set. Albeit efficient, neither strategy is complete. For example, $C = \{(p \mid q),\ \ (p \mid \neg q),\ \ (\neg p \mid q),\ \ (\neg p \mid \neg q)\}$ is unsatisfiable, though no unit resolutions are available. We may use input resolutions to obtain p and $\neg p$, but cannot obtain $(\,)$ because p and $\neg p$ are not input clauses.

Ordered resolution impose a total ordering on the propositional variables and treats clauses not as sets of literals but a sorted list of literals. Ordered resolution is extremely efficient and complete.

Set-of-support resolution is one of the most powerful strategies employed by (Wos, Carson, Robinson 1965), and its completeness depends on the choice of set of support. To prove $A \models B$ by resolution, we convert $A \wedge \neg B$ into clauses. The clauses obtained from $\neg B$ serve in general as a set of support S and the clauses derived from A are in T. Set-of-support resolution dictates that the resolved clauses are not both from T. The motivation behind set-of-support is that since A is usually satisfiable it might be wise not to resolve two clauses from against each other.

Linear resolution always resolves a clause against the most recently derived resolvent. This gives the deduction a simple "linear" structure and it appears that we can have a straightforward implementation. Because linear resolution imposes the restriction on time, we need to add a time stamp to every clause, assuming that all the time stamps of input clauses are 0. Now a clause is represented by $\langle C, t\rangle$, where t is the time stamp of C, the resolution rule needs to be modified as follows:

$$\frac{\langle (A \mid \alpha), i\rangle \qquad \langle (\neg A \mid \beta), j\rangle}{\langle (\alpha \mid \beta), max(i,j)+1\rangle} \text{ if } i \text{ or } j \text{ is maximal.}$$

Now consider the inference graph $G = (V, E)$ for linear resolution: $(\mathbf{x}, \mathbf{y}) \in E$ if $\langle (A \mid \alpha), i\rangle, \langle (\neg A \mid \beta), j\rangle \in \mathbf{x}$, i is maximal in $\mathbf{x}$, and $\mathbf{y} = \mathbf{x} \cup \langle (\alpha \mid \beta), i+1\rangle$. If a clause with the same time stamp appears in two different nodes of V, it means that we have different sequences (of the same length) to derive this clause by linear resolution. Since the sequences are different, one cannot be replaced by the other. For example, in $\mathbf{x}$, we may obtain $\langle C_3, i+1\rangle$ from $\langle C_1, i\rangle$ and $\langle C_2, j\rangle$, but cannot do the same in $\mathbf{x}'$, even if $\langle C_1, i\rangle \in \mathbf{x}'$, because $\langle C_2, j\rangle$ may not be in $\mathbf{x}'$ for any j. That means that the inference graph has to be a tree structure. The completeness of linear resolution ensures that there exists a path from the root (the input clauses) to a solution node (containing the empty clause) in $G = (V, E)$. If we do not use the depth-first search in G, multiple copies of the same clause may have to be kept in the search. This leaves us the depth-first search as our best option and we have to inherit all advantages and disadvantages of the depth-first search, such as generating the same resolvent many times, or wasting many futile steps before finding a solution node.

It must be noted that some strategies improve certain aspects of the deduction process at the expense of others. For instance, a strategy may reduce the size of the proof search space at the expense of increasing, say, the length of the shortest proofs.

3.3.3 Preserving Satisfiability

The resolution rule does not preserve the logical equivalence, that is, if D is a resolvent of C_1 and C_2, then $C_1 \wedge C_2 \models D$, or $C_1 \wedge C_2 \equiv C_1 \wedge C_2 \wedge D$, but not $C_1 \wedge C_2 \equiv D$. For example, q is a resolvent of p and $\neg p \mid q$, $\mathcal{M}(q)$ has two models, and $\mathcal{M}(p \wedge (\neg p \mid q))$ has only one model. Thus, $q \not\equiv p \wedge (\neg p \mid q))$. However, both of them are satisfiable.

Definition 3.3.6. *Given two formulas A and B, A and B are said to be* equally satisfiable*, if whenever A is satisfiable, so is B, and vice versa. We denote this relation by $A \approx B$.*

Obviously, $A \approx B$ means $\mathcal{M}(A) = \emptyset$ iff $\mathcal{M}(B) = \emptyset$ and is weaker than the logical equivalence, which requires $\mathcal{M}(A) = \mathcal{M}(B)$.

Proposition 3.3.7. *Let $S = C \cup \{(\alpha \mid \beta)\}$ and $S' = C \cup \{(\alpha \mid x), (\neg x \mid \beta)\}$ be two sets of clauses, where x is a variable not appearing in S. Then $S \approx S'$.*

Proof. If S is satisfiable and σ is a model of S, then $\sigma(C) = 1$ and $\sigma(\alpha \mid \beta) = 1$. If $\sigma(\alpha) = \bot$, we define $\sigma(x) = 1$; if $\sigma(\beta) = 0$, we define $\sigma(x) = 0$. Thus both $\sigma(\alpha \mid x) = 1$ and $\sigma(\neg x \mid \beta) = 1$. So σ is a model of S'.

On the other hand, if σ is a model of S', then σ is also a model of S without modification, because $(\alpha \mid \beta)$ is a resolvent of $(\alpha \mid x)$ and $(\neg x \mid \beta)$. □

In the above proposition, the condition that x does not appear in C is necessary. Without it, for example, if $C = \{(\neg p)\}$, α is empty, $\beta = q$ and $x = p$, then $S = \{(\neg p), (q)\}$ and $S' = \{(\neg p), (p), (\neg p \mid q)\}$. Then S is satisfiable but S' is not.

Using the above proposition, we may break a long clause into a set of shorter clauses by introducing some new variables. For example, the clause $(l_1 \mid l_2 \mid l_3 \mid l_4 \mid l_5)$ can be transformed into three clauses: $(l_1 \mid l_2 \mid x)$, $(\neg x \mid l_3 \mid y)$, and $(\neg y \mid l_4 \mid l_5)$. This transformation does not preserve the logical equivalence, but preserves the satisfiability.

Another usage of the above proposition is to use resolution to remove x from a clause set. If we exhaust all resolutions on x, keep all the resolvents and remove all the clauses containing x, then the resulting set of clauses will preserve the satisfiability, as stated in Lemma 3.3.11.

Proposition 3.3.8. *Let B be a subformula of A and x is a new variable. Then $A \approx (A[B \leftarrow x] \wedge (x \leftrightarrow B))$.*

Proof. If A is satisfiable and σ is a model of A, then we define $\sigma(x) = \sigma(B)$. When applying σ to the formula trees of A and $A[B \leftarrow x]$, all the common nodes have the same truth value, so $\sigma(A) = \sigma(A[B \leftarrow x])$. Since $\sigma(x) = \sigma(B)$, so $\sigma(x \leftrightarrow B) = 1$, thus σ is a model of $A[B \leftarrow x] \wedge (x \leftrightarrow B)$.

On the other hand, if σ is $A[B \leftarrow x] \wedge (x \leftrightarrow B)$, σ is also a model of A. □

The above proposition plays an important role when we convert a formula into CNF and keep the size of CNF as a linear function of the size of the original formula. When we use the distribution law

$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$$

to transform a formula into CNF, C becomes duplicated in the result. Because of this, there exists examples that the resulting CNF has an exponential size of the original size. Using the above proposition and introducing new variables, we may keep the size of CNF as four times of the original size, and every clause contains at most three literals. The CNF obtained this way is not logically equivalent to the original formula, but it preserves the satisfiability of the original formula: The CNF is satisfiable iff the original formula is satisfiable. This way of transforming a formula into CNF is called *Tseitin transformation*.

Example 3.3.9. Let A be $(p \wedge (q \vee \neg p)) \vee \neg(\neg q \wedge (p \vee q))$, we introduce a new variable for every proper subformula whose node is a binary operator, and convert each of them into at most three clauses.

$$\begin{array}{cc} x_1 \leftrightarrow (q \vee \neg p) & (\overline{x_1} \mid q \mid \overline{p}), (\overline{q} \mid x_1), (p \mid x_1), \\ x_2 \leftrightarrow (p \wedge x_1) & (\overline{x_2} \mid p), (\overline{x_2} \mid x_1), (\overline{p} \mid \overline{x_1} \mid x_2), \\ x_3 \leftrightarrow (p \vee q) & (\overline{x_3} \mid p \mid q), (\overline{p} \mid x_3), (\overline{q} \mid x_3), \\ x_4 \leftrightarrow (\neg q \wedge x_3) & (\overline{x_4} \mid \overline{q}), (\overline{x_4} \mid q), (q \mid \overline{x_3} \mid x_4), \\ x_2 \vee \neg x_4 & (x_2 \mid \overline{x_4}). \end{array}$$

□

Example 3.3.10. In Example 2.3.9, we have converted $p_1 \leftrightarrow (p_2 \leftrightarrow p_3)$ into four clauses:

$$(\overline{p_1} \mid \overline{p_2} \mid p_3), \quad (\overline{p_1} \mid p_2 \mid \overline{p_3}), \quad (p_1 \mid p_2 \mid p_3), \quad (p_1 \mid \overline{p_2} \mid \overline{p_3}).$$

If we like to convert $p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow p_4))$ into clauses, we may replace p_3 by $(p_3 \leftrightarrow p_4)$ in the above clauses and then covert each of them into clauses. That is, from $(\overline{p_1} \mid \overline{p_2} \mid (p_3 \leftrightarrow p_4))$, $(\overline{p_1} \mid p_2 \mid (\overline{p_3 \leftrightarrow p_4}))$, $(p_1 \mid p_2 \mid (p_3 \leftrightarrow p_4))$, and $(p_1 \mid \overline{p_2} \mid (\overline{p_3 \leftrightarrow p_4}))$, we obtain eight clauses, twice as before:

$$\begin{array}{cccc} (\overline{p_1} \mid \overline{p_2} \mid \overline{p_3} \mid p_4), & (\overline{p_1} \mid p_2 \mid p_3 \mid p_4), & (\overline{p_1} \mid \overline{p_2} \mid p_3 \mid \overline{p_4}), & (\overline{p_1} \mid p_2 \mid \overline{p_3} \mid \overline{p_4}), \\ (p_1 \mid p_2 \mid \overline{p_3} \mid p_4), & (p_1 \mid \overline{p_2} \mid p_3 \mid p_4), & (p_1 \mid p_2 \mid p_3 \mid \overline{p_4}), & (p_1 \mid \overline{p_2} \mid \overline{p_3} \mid \overline{p_4}). \end{array}$$

□

If we replace p_4 by $(p_4 \leftrightarrow p_5)$, we will get a set of 16 clauses of length 5. In general, $p_1 \leftrightarrow (p_2 \leftrightarrow (p_3 \leftrightarrow (...(p_{n-1} \leftrightarrow p_n)...)))$ will produce 2^{n-1} clauses of length n. Instead, if we introduce $n-3$ new variables for this formula of n variables, we obtain only $4(n-1)$ clauses of length 3. For example, for $n = 5$, we introduce two new variables, q_1 and q_2:

$$(p_1 \leftrightarrow (p_2 \leftrightarrow q_1)) \wedge (q_1 \leftrightarrow (p_3 \leftrightarrow q_2)) \wedge (q_2 \leftrightarrow (p_4 \leftrightarrow q_5)),$$

which will generate $3 \times 4 = 12$ clauses of length 3.

Lemma 3.3.11. *Let C be a set of clauses and x be a variable in C. Let $D \subset C$ such that each clause of D contains neither x nor $\overline{x}$. Let $X = \{(\alpha \mid \beta) \mid (x \mid \alpha), (\overline{x} \mid \beta) \in C\}$, then $C \approx D \cup X$.*

Proof. If C is satisfiable and σ is a model of C, then $\sigma(C) = 1$ and $\sigma(D) = 1$. If $\sigma(X) = 0$, then there exists a clause $\alpha \mid \beta \in X$ such that $\sigma(\alpha) = \sigma(\beta) = 0$. It means either $\sigma(x \mid \alpha) = 0$ or $\sigma(\overline{x} \mid \beta) = 0$, that is a contradiction to the fact that

$\sigma(C) = 1$, because both $(x \mid \alpha)$ and $(\overline{x} \mid \beta)$ are in C. So $\sigma(X) = 1$. It means σ is a model of $D \cup X$.

On the other hand, if $D \cup X$ is satisfiable and σ is its model, we may define $\sigma(x) = 1$ if $\sigma(\alpha) = 0$ for some $x \mid \alpha \in C$; in this case, we must have $\sigma(\beta) = 1$ for all $(\overline{x} \mid \beta) \in C$; otherwise, if $\sigma(\beta) = 0$ for some $(\overline{x} \mid \beta) \in C$, then $\sigma(\alpha \mid \beta) = 0$, that is a contradiction to the fact that $\sigma(X) = 1$. Similarly, if $\sigma(\alpha) = 1$ for all $x \mid \alpha \in C$, we define $\sigma(x) = 0$. Then σ becomes a model of C. □

This lemma plays an important role in the completeness proof of resolution: The set X can be obtained from C by resolution. From C to $D \cup X$, we have eliminated one variable from C. Replace C by $D \cup X$ and we repeat the same process and remove another variable, until no more variable in C: either $C = \top$ or $C = \bot$. From the final result, we can tell if the original set of clauses is satisfiable or not.

3.3.4 Completeness of Resolution

Theorem 3.3.12. *Ordered resolution is complete with any total ordering on the variables.*

Proof. Suppose there are n variables in the input clauses S_n with the order $x_n > x_{n-1} > \cdots > x_2 > x_1$. For i from n to 1, we apply Lemma 3.3.6 with $C = S_i$ and $x = x_i$, and obtain $S_{i-1} = D \cup X$. Every resolvent in X is obtained by ordered resolution on x_i, which is the maximal variable in S_i. Finally, we obtain S_0 which contains no variables. Since $S_n \approx S_{n-1} \approx \cdots \approx S_1 \approx S_0$, if S_n is unsatisfiable, then S_0 must contain $\bot$; if S_n is satisfiable, we arrive at $S_0 = \emptyset$, i.e., $S_0 \equiv \top$. In this case, we may use the idea in the proof of Lemma 3.3.6 to assign a truth value for each variable, from x_1 to x_n, to obtain a model of S_n. □

Example 3.3.13. Let $S = \{c_1.\ (\overline{a}|\overline{b}|\overline{c}), c_2.\ (a|\overline{e}), c_3.\ (b|\overline{c}), c_4.\ (\overline{b}|d),\ c_5.\ (c|\overline{d}|\overline{e}),$ $c_6.\ (\overline{d}|e)\ \}$ and $a > b > c > d > e$. By ordered resolution, we get $c_7.\ (\overline{b}|\overline{c}|\overline{e})$ from c_1 and c_2; $c_8.\ (\overline{c}|d)$ from c_3 and c_4; $c_9.\ (\overline{c}|\overline{e})$ from c_3 and c_7; and $c_{10}.\ (\overline{d}|\overline{e})$ from c_5 and c_9. No other clauses can be generated by ordered resolution, that is, c_1–c_{10} are saturated by ordered resolution. Since the empty clause is not generated, we may construct a model from c_1–c_{10} by assigning truth values to the variables from the least to the greatest. Let S_x be the set of clauses from c_1–c_{10} such that x is the maximal variable.

- $S_e = \emptyset$ and we can assign either 0 or 1 to e, say $\sigma_e = \{e\}$.
- $S_d = \{c_6, c_{10}\}$. From c_{10}, $\overline{d}$ has to be 1, so $\sigma_d = \sigma_e \cup \{d = 0\} = \{e, \overline{d}\}$.

- $S_c = \{c_5, c_8, c_9\}$. From c_8 or c_9, $\overline{c}$ has to be 1, so $\sigma_c = \sigma_d \cup \{c = 0\} = \{e, \overline{d}, \overline{c}\}$.
- $S_b = \{c_3, c_7\}$. Both c_3 and c_7 are satisfied by σ_c, we may assign 0 or 1 to b, say $b = 1$, so $\sigma_b = \sigma_c \cup \{b\} = \{e, \overline{d}, \overline{c}, b\}$.
- $S_a = \{c_1, c_2\}$. From c_2, a has to be 1, so $\sigma_a = \sigma_b \cup \{a\} = \{e, \overline{d}, \overline{c}, b, a\}$.

□

The above example illustrates that if a set of clauses is saturated by ordered resolution and the empty clause is not there, we may construct a model from these clauses. In other words, we may use ordered resolution to construct a decision procedure for deciding if a formula A is satisfiable or not.

Algorithm 3.3.14. The algorithm *orderedResolution* will take a set C of clauses and an order V of variables, $V = (p_1 < p_2 < ... < p_n)$. It returns true iff C is satisfiable. It uses the procedure $sort(A)$, which places the maximal literal of clause A as the first one in the list.

```
proc orderedResolution(C, V)
    C := {sort(A) | A ∈ C};
    for i := n downto 1 do
        X := ∅;
        for (p_i | α) ∈ C do // p_i is maximal in (p_i | α)
            for (p̄_i | β) ∈ C do // p_i is maximal in (p̄_i | β)
                A := (α | β);
                if A = ( ) return false;
                X := X ∪ {sort(A)};
        C := C ∪ X;
    return true;
```

Before returning true at the last line, this algorithm can call *createModel* to construct a model of C when C is *saturated* by ordered resolution. That is, C contains the results of every possible ordered resolution among the clauses of C.

```
proc createModel(C)
    σ := ∅; // the empty interpretation
    for i := 1 to n do
        v := 0;
        for (p_i | α) ∈ C do // p_i is maximal
            if eval(α, σ) = ⊥ do
```

$v := 1$;
$\sigma := \sigma \cup \{p_i \mapsto v\}$;
return σ;

The resolution algorithm is regarded as a deduction-style proof procedure when the empty clause is its goal. If finding a model is the goal of the resolution algorithm, it is a *saturation-style* proof procedure, because the set of clauses is *saturated* by ordered resolution, that is, all the possible results of the inference rules are generated, before we can build a model.

Corollary 3.3.15. *Resolution is a sound and complete decision procedure for propositional logic.*

Proof. We have seen that resolution is sound. If C is unsatisfiable, we may obtain an ordered resolution proof from C. This proof is also a resolution proof. Thus $C \models \bot$ implies $C \vdash_{\mathcal{R}} \bot$. If C is satisfiable, we may construct a model for C as we do for ordered resolution. □

Theorem 3.3.16. *Positive resolution is complete for refutation.*

Proof. Suppose C is unsatisfiable and there is a resolution proof $C \vdash_{\mathcal{R}} \bot$. We prove by induction on the length of the proof on how to convert this proof into a positive resolution proof.

Let P be a resolution proof of $C \vdash_{\mathcal{R}} \bot$ of minimal length. If $|P| = 1$, then the only resolution uses a positive unit clause to obtain $\bot$ and it is a positive resolution. If $|P| > 1$, suppose the last resolution in P resolves on variable q. Then $C \vdash_{\mathcal{R}} q$ and $C \vdash_{\mathcal{R}} \overline{q}$. Let P_1 and P_2 be the proofs of $C \vdash_{\mathcal{R}} q$ and $C \vdash_{\mathcal{R}} \overline{p}$ presented in P, respectively.

If we remove all the occurrences of q from P_1, we obtain a resolution proof P_1' and $core(P_1')$ contains clauses from $core(P)$ with q being removed. Since $|P_1'| < |P|$, by induction hypothesis, we have a positive resolution proof of $core(P_1') \vdash_R \bot$. If we add q back to $core(P_1')$, we obtain a positive resolution proof $P"_1$ of $core(P) \vdash_{\mathcal{R}} q$.

We then remove all the occurrences of $\overline{q}$ from P_2, and obtain a resolution proof P_2' and $core(P_2')$ contains clauses from $core(P)$ with $\overline{q}$ being removed. By induction hypothesis, there exists a positive resolution proof $P"_2$ of $core(P_2') \vdash_{\mathcal{R}} \bot$. For each clause $\alpha \in core(P_2')$, if α is obtained from $\alpha \mid \overline{q} \in core(P)$, we add α as the resolvent of $\alpha \mid\neq q$ and q at the beginning of $P"_2$. Finally, we append $P"_1$, which generates q by positive resolution, and $P"_2$ into one proof, the result is a positive resolution proof of $C \vdash_{\mathcal{R}} \bot$. □

The following theorem can be proved using a similar approach.

Theorem 3.3.17. *Negative resolution is complete for refutation.*

Completeness of resolution strategies are important to ensure that if the empty clause is not found, the input clauses are unsatisfiable. In this case, one strategy may generate the empty clause quicker than another strategy. When the input clauses are satisfiable, we have to compute all possible resolutions to ensure that no new resolvents before claiming the input clauses are satisfiable. In this case, we say the set of clauses, old and new, is saturated by resolution. Restricted resolutions like ordered resolution, positive resolution, and set-of-support resolution, work much better than unrestricted resolution, when the saturation is needed to show the satisfiability of a set of clauses.

3.3.5 A Resolution-based Decision Procedure

If we like to employ heuristics in a decision procedure based on resolution, the following procedure fits the purpose.

Algorithm 3.3.18. The algorithm *resolution* will take a set C of clauses and returns true iff C is satisfiable. It uses the procedure *resolvable*(A, B), which decides if clauses A and B are allowed to do restricted resolution, and if yes, *resolve*(A, B) will their resolvent. It uses *pickClause*(C) to pick out a clause according to a heuristic.

proc *resolution*(C)
    $G := C$; // G: given clauses
    $K = \emptyset$; // K: kept clauses
    while $G \neq \emptyset$ **do**
        $A :=$ *pickClause*(G);
        $G := G - \{A\}$;
        for $B \in K$ **if** *resolvable*(A, B) **do**
            $res :=$ *resolve*(A, B);
            if $res = (\,)$ **return** false;
            if $res \notin (G \cup K)$
                $G := G \cup \{res\}$;
        $K := K \cup \{A\}$;
    return true;

In the procedure, we move each clause from G to K, and compute the resolution between this clause and all the clauses in K. When G is empty, resolution between any two clauses are done.

With the exception of linear resolution, the above procedure can be used to implement all the resolution strategies introduced in this section. The restriction will be implemented in *resolvable*. For set-of-resolution, a better implementation is that G is initialized with the set of support and K is initialized with the rest of the input clauses.

The termination of this procedure is guaranteed because only a finite number of clauses exist. When the procedure returns false, the answer is correct because resolution is sound. When it returns true, the correctness of the answer depends on the completeness of the resolution strategy implemented in *resolvable*.

In *pickClause*, we implement various heuristics for selecting a clause in G to do resolutions with clauses in K. Popular heuristics include preferring shorter clauses or older clauses. Preferring shorter clauses because shorter clauses are stronger as constraints on the acceptance of interpretations as models. An empty clause removes all interpretations as models; a unit clause removes half of the interpretations as models; a binary clause removes a quarter of the interpretations as models; and so on. Preferring older clauses alone would give us the breadth-first strategy. If we wish to find a shorter proof, we may mix this preference with other heuristics.

Despite the result that complete resolution strategies can serve as a decision procedure for positional logic, and some strategies can even build a model when the input clauses are satisfiable, resolution provers are in general not efficient for propositional logic, because the number of possible clauses is exponential in terms of the number of variables. The computer may quickly run of memory before finding a solution for real application problems.

3.3.6 Clause Deletion Strategies

There are several strategies which can safely remove redundant clauses during the resolution. They are tautology deletion, subsumption deletion, and pure literal deletion.

A clause is a tautology if it contains a complementary pair of literals of the same variable. For example, $(p \mid \overline{p})$. A tautology clause is true in every interpretation and is not needed in any resolution proof.

Proposition 3.3.19. *Given any set S of clauses, let S be partitioned into $S' \cup T$, where T is the set of tautology clauses in S. Then $S \equiv S'$.*

Proof. For any interpretation σ, $\sigma(S) = \sigma(S') \wedge \sigma(T) = \sigma(S')$ as $\sigma(T) = 1$. □

Definition 3.3.20. *A clause A* subsumes *another clause B if $B = A \mid \alpha$ for some α. That is, every literal of B appears in A.*

For example, $(p \mid q)$ subsumes $(p \mid q \mid r)$. Intuitively, resolution tries to delete literals by resolution one by one to obtain an empty clause. If we use $(p \mid q)$ instead of $(p \mid q \mid r)$, we avoid the job of removing r.

Proposition 3.3.21. *If A subsumes B, then $S \cup \{A, B\} \equiv S \cup \{A\}$.*

Proof. Since A subsumes B, $\mathcal{M}(A) \subseteq \mathcal{M}(A) \cup \mathcal{M}(\alpha) = \mathcal{M}(B)$. $\mathcal{M}(S \cup \{A, B\}) = \mathcal{M}(S) \cap (\mathcal{M}(A) \cap \mathcal{M}(B)) = \mathcal{M}(S) \cap \mathcal{M}(A) = \mathcal{M}(S \cup \{A\})$. □

The above proposition shows that if A subsumes B, then B can be dropped as a constraint in the presence of A. The following proposition shows that B is not needed in a resolution proof.

Proposition 3.3.22. *If A and B are two clauses appearing in the resolution proof of $S \vdash_{\mathcal{R}} \bot$ such that A subsumes B, then B is no longer needed once A is present.*

Proof. If B is used in a resolution proof after the presence of A, we may replace B by A to obtain a proof of shorter or equal length, by checking all the resolutions which remove every literal from B: Suppose the first resolution is between B and γ and the resolvent is B'. There are two cases to consider: (1) the resolved variable is not in A. This resolution is unnecessary, because A subsumes B' and we replace B' by A. (2) the resolved variable is in A, then let the resolvent of A and γ be A'. It is easy to check that A' subsumes B', and we replace B' by A'. We replace the descendants of B by the corresponding descendants of A, until the last resolution. The modified resolution proof will have shorter or equal length. □

Proposition 3.3.23. *If there exists a resolution proof using a tautology clause, then there exists a resolution proof without the tautology clause.*

Proof. Suppose the tautology clause appearing in the proof is $(p \mid \overline{p} \mid \alpha)$. We need another clause $(\overline{p} \mid \beta)$ (or $(p \mid \beta)$) to resolve off p. The resolvent of $(p \mid \overline{p} \mid \alpha)$ and $(\overline{p} \mid \beta)$ is $(\overline{p} \mid \alpha \mid \beta)$, which is subsumed by $(\overline{p} \mid \beta)$. By Proposition 3.3.22, $(\overline{p} \mid \alpha \mid \beta)$ is not needed in the resolution proof. Since every resolvent of a tautology is not needed, so we need any tautology in a resolution proof. □

Definition 3.3.24. *A literal l is said to be* pure *in a set C of clauses if its complement does not appear in C.*

Proposition 3.3.25. *If l is pure in C, let D be the set of clauses obtained from C by removing all clauses containing l from C, then $C \approx D$.*

Proof. If σ is a model of C, then σ is also a model of D. If σ is a model of D, define $\sigma(l) = 1$, then σ is a model of C. □

For example, if $C = \{1.\ (p \mid q),\ 2.\ (\overline{p} \mid r),\ 3.\ (p \mid \overline{q} \mid r),\ 4.\ (\overline{p} \mid q)\}$, then r is pure. Remove 2. and 3. from C, q becomes pure. Remove 1. and 4., we obtain an empty set of clauses, which is equivalent to 1. By the proposition, it means C is satisfiable. A model can be obtained by setting $q = r = 1$ in the model.

The above propositions allow to use the following clause deletion strategies without worrying about missing a proof:

- **tautology deletion**: Tautology clauses are discarded.
- **subsumption deletion**: Subsumed clauses are discarded.
- **pure deletion**: Clauses containing pure literals are discarded.

These deletion strategies can be integrated into the algorithm *resolution* as follows.

Algorithm 3.3.26. The algorithm *resolution* will take a set S of clauses and returns true iff S is satisfiable. In *preprocessing*(S), we may implement pure-literal check or subsumption to simplify the input clauses as a preprocessing step. *subsumedBy*(A, S) checks if clause A is subsumed by a clause in S.

```
proc resolution(S)
1     G :=preprocessing(S); // G: given clauses
2     K := ∅; // K: kept clauses
3     while G ≠ ∅ do
4         A := pickClause(G);
5         G := G − {A};
6         N := { }; // new clauses from A and K
7         for B ∈ K if resolvable(A, B) do
8             res := resolve(A, B);
9             if res = ( ) return false;
10            if subsumedBy(res, G ∪ K) continue;
11            N := N ∪ {res};
12        K = K ∪ {A};
13        for A ∈ G if subsumedBy(A, N) do
14            G := G − {A};
15        for B ∈ K if subsumedBy(B, N) do
16            K := K − {B};
```

17 $G := G \cup N$;
18 **return** true; // K is saturated by resolution

A life cycle of a clause A goes as follows in the algorithm *resolution*.

1. If A is an input clause, it passes the preprocessing check, such as pure-literal check and subsumption check, and goes into G (line 1).

2. If A is picked out of G (lines 4,5), it will pair with every clause B in K to try resolution on A and B. Once this is done, A will go to K as a kept clause (line 12).

3. A may be subsumed by a new clause when it is in G (line 13) or in K (line 15), and will be thrown away.

4. If A is a new clause (line 8), it has to pass the subsumption check (line 10) and parks in N (line 11).

5. All new clauses in N will be used to check if they can subsume some clauses in G (line 13) and (line 15), then merge into G (line 17), and start a life cycle the same way as an input clause.

6. If the algorithm stops without finding the empty clause (line 9), the set G will be empty (line 3), and all the resolutions among the clauses in K are done. That is, K is saturated by resolution.

In this algorithm, unit clauses are given high priority to do unit resolution and subsumptions. Dealing with unit clauses can be separated from the main loop of the algorithm as we will see in the next section.

3.4 Boolean Constraint Propagation (BCP)

A unit clause (A) subsumes every clause like $(A \mid \alpha)$, where A appears as a literal. Unit resolution between (A) and $(\overline{A} \mid \alpha)$, where $\overline{A} = p$ if A is $\overline{p}$, generates the resolvent (α), which will subsume $(\overline{A} \mid \alpha)$. The rule for replacing $(\overline{A} \mid \alpha)$ by (α) at the presence of (A) is called *unit deletion*. That is, when subsumption deletion is used in resolution, a unit clause (A) allows us to remove all the occurrences of the variable p in (A), with the unit clause itself as the only exception. New unit clauses may be generated by unit resolution and we can continue to simplify the clauses by the new unit clause and so on. This process is traditionally called *Boolean constraint propagation* (BCP), or *unit propagation*.

3.4.1 BCP: a Simplification Procedure

Clause deletion strategies allow us to remove unnecessary clauses during resolution. BCP also allows us to simplify clauses by removing unnecessary clauses and shortening clauses. In the following, we will describe BCP as a procedure which takes a set C of clauses as input and returns a pair (U, S), where U is a set of unit clauses and S is a set non-unit clauses such that U and S share no variables and $C \equiv U \cup S$. Typically, S contains a subset of C or clauses from C by removing some literals.

Algorithm 3.4.1. The algorithm *BCP* will take a set C of clauses and apply unit resolution and subsumption deletion repeatedly, until no more new clauses can be generated by unit resolution. It will return $\perp$ if an empty clause is found; otherwise, it returns a simplified set of clauses equivalent to C.

proc *BCP*(C)
    $S := C$; // S: simplified clauses
    $U := \emptyset$; // U: unit clauses
    while S has a unit clause (A) **do**
        $S := S - \{(A)\}$;
        $U := U \cup \{A\}$;
        for $(A \mid \alpha) \in S$ **do**
            $S := S - \{(A \mid \alpha)\}$;
        for $(\overline{A} \mid \alpha) \in S$ **do**
            if $\alpha = (\,)$ **return** $\perp$;
            $S := S - \{(\overline{A} \mid \alpha)\} \cup \{(\alpha)\}$;
    return (U, S);

Example 3.4.2. Let $C = \{1 : (x_2 \mid x_5),\ 2 : (\overline{x_1} \mid \overline{x_4}),\ 3 : (\overline{x_2} \mid x_4),\ 4 : (x_1 \mid x_2 \mid \overline{x_3}),\ 5 : (\overline{x_5})\}$. Then *BCP*($C$) will return

$$(\{\overline{x_5}, x_2, x_4, \overline{x_1}\}, \emptyset)$$

because we first add $\overline{x_5}$ into U; clause 1. becomes (x_2), x_2 is then added into U; clause 4. is deleted because $x_2 \in U$; clause 3. becomes (x_4) and x_4 is added into U; clause 2. becomes $(\overline{x_1})$ and $\overline{x_1}$ is added into U. The set U allows us to construct two models of C:

$$\{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto v, x_4 \mapsto 1, x_5 \mapsto 0\}$$

where $v \in \{0, 1\}$ as x_3 can take either value. □

Proposition 3.4.3. *(a) If* BCP(C) *returns* $\bot$*, then* C *is unsatisfiable. (b) If* BCP(C) *returns* (U, S)*, then* $C \equiv U \cup S$*, where* U *is a set of unit clauses and* S *is a set of non-unit clauses, and* U *and* S *share no variables.*

Proof. In *BCP*(C), S is initialized with C. S is updated through three ways: (1) unit clause A is removed from S into U; (2) clause $(A \mid \alpha)$ is removed from S; and (3) clause $(\overline{A} \mid \alpha)$ is replaced by (α). (α) is the resolvent of (A) and $(\overline{A} \mid \alpha)$, so $C \models \alpha$.

(a): If $\alpha = (\,)$ in (3), $\bot$ will be returned and $C \equiv \bot$ because $C \models \bot$.

(b): Initially, $C \equiv U \cup S$. Since A subsumes $A \mid \alpha$ and α subsumes $(\overline{A} \mid \alpha)$, by Proposition 3.3.21, it justifies that both $A \mid \alpha$ and $(\overline{A} \mid \alpha)$ can be safely removed, as it maintains $C \equiv U \cup S$ for each update of U and S. When A is moved from S to U, either A or $\overline{A}$ is removed from S, so no variables of A appear in S. The procedure stops only when S does not have any unit clauses. □

The procedure *BCP* can serve as a powerful simplification procedure when unit clauses are present in the input. *BCP* plays an important role in deciding if a formula is satisfiable, which is the topic of the next chapter. We will show later that *BCP*(C) can implemented in time $O(n)$, where n is the number of literals in C.

3.4.2 BCP: a Decision Procedure for Horn Clauses

A *Horn clause* is a clause with at most one positive literal. Horn clauses are named for the logician Alfred Horn, who first pointed out their significance in 1951. A Horn clause is called *definite clause* if it has exactly one positive literal. Thus Horn clauses can be divided into definite clauses and negative clauses. Definite clauses can be further divided into *fact* (a positive unit clause) and *rule* (non-unit definite clauses). Horn clauses have important applications in logic programming, formal specification, and model theory, as they have very efficient decision procedures, using unit resolution or input resolution. That is, unit and input resolutions are incomplete in general, but they are complete for Horn clauses.

Proposition 3.4.4. BCP *is a decision procedure for Horn clauses.*

Proof. Suppose H is a set of Horn clauses and $\bot \notin H$. It suffices to show that procedure *BCP*(H) returns $\bot$ iff $H \equiv \bot$. If *BCP*(H) returns $\bot$, by Proposition 3.4.3, $H \equiv \bot$. If *BCP*(H) returns (U, S), by Proposition 3.4.3, $H \equiv U \cup S$, U is a set of unit clauses, S is a set of non-unit clauses, and U and S share no variables. In this case, we create an assignment σ in which every literal in U is true, and every variable in S is false. Note that each variable in U appears only once; U and S do

not share any variable. Thus this assignment is consistent and is a model of $U \cup S$, because every unit clause in U is true under σ; for each every clause in S, since it is a non-unit Horn clause and must have a negative literal, which is true under σ. □

Theorem 3.4.5. *Unit resolution is sound and complete for Horn clauses.*

Proof. Unit resolution is sound because resolution is sound. The only used inference rule in *BCP* is unit resolution. Every clause generated by *BCP* can be generated by unit resolution. Subsumption only reduces some unnecessary clauses. If the input clauses are unsatisfiable, *BCP* will generate the empty clause, which can also be generated by unit resolution. If the input clauses are satisfiable, unit resolution will terminate with a set of clauses saturated by unit resolution, without generating the empty clause, because unit resolution is sound. □

3.4.3 Unit Resolution versus Input Resolution

In the following, we show that input resolution and unit resolution are equivalent, in the sense that if there exists a unit resolution proof, then there exists an input resolution proof and vice versa.

Theorem 3.4.6. *For any set C of clauses, if there exists an input resolution proof from C, then there exists a unit resolution proof from C.*

Proof. We will prove this theorem by induction on the length of proofs. Let P be an input resolution proof from C of minimal length. The last resolvent is $\perp$ and both its parents are unit clauses and one of them must be an input clause, say A. For simplicity, assume all the assumed clauses appear in the beginning of P and A is the first in P. If $|P| = 1$, then P must be a unit resolution proof and we are done. If $|P| > 1$, then remove A and the last clause from P, and remove all occurrences of $\overline{A}$ from P, where $\overline{A} = \overline{p}$ if $A = p$ and $\overline{A} = p$ if $A = \overline{p}$. The result is an input resolution proof P' from C', where C' is obtained from C by removing all occurrences of $\overline{A}$ in C. That is, $(\overline{A} \mid \alpha) \in C$ iff $(\alpha) \in C'$. Because $|P'| < |P|$, by induction hypothesis, there exists a unit resolution Q' proof from C'. If $(\alpha) \in C'$ comes from $(\overline{A} \mid \alpha) \in C$, then change the reason of $(\alpha) \in Q'$ from "assumed" to "resolution from A and $(\overline{A} \mid \alpha)$". Let the modified Q' be Q, then Q is a unit resolution from C. □

Example 3.4.7. We illustrate the proof of this theorem by Example 3.3.5, where

$$C = \{1.\ (p \mid q),\ 2.\ (\overline{p} \mid r),\ 3.\ (p \mid \overline{q} \mid r),\ 4.\ (\overline{r})\}.$$

Let P be (b) of Example 3.3.5. Then $A = \overline{r}$, $C' = \{1.\ (p \mid q),\ 2'.\ (\overline{p}),\ 3'.\ (p \mid \overline{q})\}$ and two input resolutions in P' are $5.:\ p\ (1, 3')$ and $6.\ \bot\ (2', 5)$. A unit resolution Q' from C' will contain three unit resolutions:

$$5.\ (q)\ (1, 2') \quad 6.\ (\overline{q})\ (2', 3') \quad 7.\ (\,)\ (5, 6).$$

Adding two unit resolutions before Q', we obtain Q:

$$2':\ (\overline{p})\ (2, 4), \quad 3':\ (p \mid \overline{q})\ (3, 4) \quad 5.\ (q)\ (1, 2') \quad 6.\ (\overline{q})\ (2', 3') \quad 7.\ (\,)\ (5, 6),$$

which is a unit resolution proof from C. □

Theorem 3.4.8. *For any set C of clauses, if there exists a unit resolution proof from C, then there exists an input resolution proof from C.*

The proof of this theorem is left as exercise.

The above two theorems establish the equivalence of input resolution and unit resolution: A unit resolution proof exists iff an input resolution proof exists. Since unit resolution is complete for Horn clauses, we have the following result.

Corollary 3.4.9. *Input resolution is sound and complete for Horn clauses.*

Theorem 3.4.10. *For any set H of Horn clauses, let $H = S \cup T$, where S is a set of negative clauses and $T = H - S$. Then input resolution is sound and complete for H, when S is used as the set of support.*

Proof. Resolution is not possible among the clauses in S as resolution needs a pair of complementary literals from the two parent clauses. Set-of-support resolution does not allow resolution among the clauses in T. Thus resolution is possible only when one clause is from S and one from T, and the resolvent is a negative clause, because the only positive literal from T is resolved off. For each resolution between S and T, we add the resolvent into S, until either the empty clause $\bot$ is found or no new resolvents can be generated. If $\bot$ is generated, then $H \equiv \bot$ because resolution is sound. If no new resolvents can be generated, we create an assignment σ as follows: call BCP on T, since T is satisfiable, let $(U, S) = BCP(T)$. For any unit clause p in U, let $\sigma(p) = 1$; for any other unassigned variable q in $S \cup S$, let $\sigma(q) = \bot$. We claim that σ is a model of H. At first, $\sigma(U \cup S) = 1$, because U and S do not share any variables, and the assignment will make every clause in $U \cup S$ to be true. Let S be the set of negative clauses from H plus all the resolvents between S and T.

If $\sigma(S) = \bot$, we look for a clause $C \in S$, such that $\sigma(C) = \bot$ and every variable of C appears as a unit clause in T. Since C is a negative clause, let $C = (\overline{p_1} \mid \overline{p_2} \mid \cdots \mid \overline{p_k})$. Since $\sigma(C) = \bot$, $p_i \in U$ and $\sigma(p_i) = 1$ for $1 \leq j \leq k$. If

$p_k \notin T$, p_k is obtained by unit resolution in $BCP(T)$, say from $(\overline{q_1} \mid \overline{q_2} \mid p_k) \in T$ and $q_1, q_2 \in U$, then this unit resolution can be avoided by an input resolution between $(\overline{q_1} \mid \overline{q_2} \mid p_k) \in T$ and $C \in S$ to generate

$$C' = (\overline{p_1} \mid \overline{p_2} \mid \cdots \mid \overline{p_{k-1}} \mid \overline{q_1} \mid \overline{q_2}) \in S.$$

Obviously, $\sigma(C') = 0$. Let C be C' and we check again if every variable of C appears as a unit clause in T. Finally, we will find a clause $C \in S$, such that every variable of C appears as a unit clause in T. In this case, input resolution between C and $\{(p_1), (p_2), ..., (p_k)\}$ will generate the empty clause ( ), a contradiction coming from the assumption that $\sigma(S) = 0$. So $\sigma(S) = 1$. □

Note that input resolution with negative clauses as the set of support on Horn clauses is also negative resolution. If the set of support initially contains a single clause, this input resolution is also a linear resolution. If the positive literal is maximal in each clause, then this resolution is also ordered resolution.

Corollary 3.4.11. *If H is a set of Horn clauses which contains a single negative clause, then the negative, input and linear resolution is complete for H.*

Proof. If C is the only negative clause, let $S = \{C\}$ and $T = H - S$. Using S as the set of support, we will get an input resolution proof. Since all negative clauses between S and T are from C, the proof can be arranged as a linear resolution proof. □

3.4.4 Head/Tail Literals for *BCP*

For a unit clause (A) to be true, the truth value of literal A needs to be 1. We can use a partial interpretation σ to remember which literals appear in unit clauses.

Definition 3.4.12. *A* partial interpretation *is a set σ of literals, where no variables appear in σ more than once, with the understanding that if $A \in \sigma$, then A is true and $\overline{A}$ is false in σ. When σ contains every variable exactly once, it is called a* full interpretation

It is easy to see that a full interpretation σ is equivalent to the interpretation σ': for every variable p, $\sigma'(p) = 1$ if $p \in \sigma$ and $\sigma'(p) = 0$ if $\overline{p} \in \sigma$. From the above definition, it is convenient to consider a set of unit clauses as a partial interpretation. For example, if $U = \{p, \neg q, r\}$, it represents the interpretation $\sigma = \{p \mapsto 1, q \mapsto 0, r \mapsto 1\}$. From now on, we do not distinguish a full interpretation from the old definition of interpretation.

We can evaluate the truth value of a clause or formula in a partial interpretation as we do in a full interpretation.

Definition 3.4.13. *Given a partial interpretation σ and a clause c,*

- *c is* satisfied *if one or more of its literals is true in σ.*
- *c is* conflicting *if all the literals of c are false in σ.*
- *c is* unit *if all but one of its literals are false and c is satisfied.*
- *c is* unresolved *if c is not one of the above three cases.*

When c becomes unit, we have to assign 1 to the unassigned literal, say A, in c, so that c becomes true. That is, c is the *reason* for literal A being true, and we record this as $reason(A) = c$. We also say the literal A is *implied* by c.

Example 3.4.14. Given the clauses $\{c_1 : (\overline{x_1}), \quad c_2 : (x_3), \quad c_3 : (x_1 \mid x_2 \mid \overline{x_3})\}$, $reason(\overline{x_1}) = c_1$, $reason(x_3) = c_2$, and $reason(x_2) = c_3$. x_2 is implied to be true by c_3 because c_3 is equivalent to $(\overline{x_1} \wedge x_3) \rightarrow x_2$. □

In *BCP*, we start with an empty partial interpretation σ. If we have found a unit clause (A) , we extend σ by adding A into σ so that the unit clause becomes satisfied. To save time, we do not remove clauses containing A from the clause set. We do not remove $\overline{A}$ from any clause to create a new clause. In stead, we just remember $\overline{A}$ is now false in σ.

We need to find out which clauses become unit clauses when some of literals become false. A clause c of k literals becomes unit when $k-1$ literals of c are false. If we know two literals of c are not false, we are certain that c is not unit. Let us designate these two literals as *head* and *tail* literals of c, which are two distinct literals of c, we have the following property.

Proposition 3.4.15. *A clause c of length longer than one is neither unit nor conflicting in a partial interpretation σ iff either one of the head/tail literals of c is true or neither of them is false in σ.*

Proof. Suppose c contains k literals and $k > 1$. If c is satisfied, then it contains a true literal and we may use it as one of the head/tail literals. If c is unit or conflicting, there are $k-1$ or all literals of c are false, and no literal in c is true and there do not exist two literals which are not false for the head/tail literals. If c is unsolved, then there must exist two unassigned literals for the head/tail literals. □

Initially, let the first and last literals be the head/tail literals. When a head literal becomes false, we scan right to find in c the next literal which is not false and

make it the new head literal; similarly, when a tail literal becomes false, we scan left to find the next tail literal. If we cannot find new head or tail literal, we must have found a unit clause or a conflicting clause.

To implement the above idea, we use the following data structures for clauses: For each clause c containing more than one literal:

- $lits(c)$: the array of literals storing the literals of c;
- $head(c)$: the index of the first non-false literal of c in $lits(c)$;
- $tail(c)$: the index of the last non-false literal of c in $lits(c)$;

Let $|c|$ denote the number of literals in c, then the valid indices for $lits(c)$ are $\{0, 1, ..., |c| - 1\}$, $head(c)$ is initialized with 0, and $tail(c)$ is with $|c| - 1$.

The data structures associated with each literal A are:

- $val(A)$: the truth value of A under partial interpretation, $val(A) \in \{1, 0, \times\}$, $\times$ ("unassigned") is the initial value;
- $cls(A)$: list of clauses c such that A is in c and pointed by either $head(c)$ or $tail(c)$ in $lits(c)$. We use insert$(c, cls(A))$ and remove$(c, cls(A))$ to insert and remove clause c into/from $cls(A)$, respectively.

Assuming global variables are used for clauses and literals, the procedure BCP can be implemented as follows:

```
proc BCP(C)
      // initialization
1:    U := ∅; // U: stack of unit clauses
2:    for c ∈ C do
3:        if c = ( ) return ⊥;
4:        if (|c| > 1)
5:            lits(c) := makeArray(c); // create an array of literals
6:            head(c) := 0; insert(c, cls(lits(c)[0]));
7:            tail(c) := |c| − 1; insert(c, cls(lits(c)[|c| − 1]));
8:        else push(c, U);
9:    for each literal A, val(A) := ×; // × = "unassigned"
      // major work
10:   res := BCPht(U);
      // finishing
11:   if (res ≠ "SAT") return ⊥;
```

12: $U := \{A \mid val(A) = 1\}$;
13: $S := \{clean(c) \mid val(lits(c)[head(c)]) = \times, val(lits(c)[tail(c)]) = \times\}$;
14: **return** (U, S);

Note that the procedure $clean(c)$ at line 13 will remove from c those literals which are false (under val) and return $\top$ if one of the literals in c is true. The procedure *BCPht* will do the major work of *BCP*.

Algorithm 3.4.16. *BCPht*(U) assumes the input clauses $C = U \cup S$, where U is a stack of unit clauses and S is a set of non-unit clauses stored using the head/tail data structure, and can be accessed through $cls(A)$. *BCPht* returns a conflict clause if an empty clause is found during unit propagation; returns "SAT" if no empty clause is found.

proc *BCPht*(U)
1 **while** $U \neq \emptyset$ **do**
2 $A :=$ pop(U);
3 **if** $val(A) = 0$ **return** $reason(\overline{A})$; // an empty clause is found;
4 **else if** $(val(A) = \times)$ // $\times$ is "unassigned
5 $val(A) := 1; val(\overline{A}) := 0$;
6 **for** $c \in cls(\overline{A})$ **do** // $\overline{A}$ is either head or tail of c
7 **if** $(\overline{A} = lits(c)[head(c)])$
8 $e_1 := head(c); e_2 := tail(c); step := 1$; // scan from left to right
9 **else**
10 $e_1 := tail(c); e_2 := head(c); step := -1$; // scan from right to left
11 **while** true **do**
12 $x := e_1 + step$; // x takes values from $e_1 + step$ to e_2
13 **if** $x = e_2$ **break**; // exits the inner while loop
14 $B := lits(c)[x]$;
15 **if** $(val(B) \neq 0)$
16 remove$(c, cls(\overline{A}))$; insert$(c, cls(B))$;
17 **if** $(step = 1)$ $head(c) := x$; **else** $tail(c) := x$;
18 **break**; // exit the inner while loop
19 **if** $(x = e_2)$ // no new head or tail
20 $A := lits(c)[e_2]$; // check if c is unit or conflicting
21 **if** $val(A) = 0$ **return** c; // c is conflicting;
22 **else if** $(val(A) = \times)$ // c is unit
23 push$(A, U); reason(A) := c; val(A) := 1; val(\overline{A}) := 0$;
24 **return** "SAT"; // no empty clauses are found;

Example 3.4.17. For the clauses in Example 3.4.2, we have the following data structures after the initialization (the head and tail indices are given after the literal list in each clause):

$C = \{c_1 : \langle[x_2, x_5], 0, 1\rangle, c_2 : \langle[\overline{x_1}, \overline{x_4}], 0, 1\rangle,\ c_3 :\ \langle[\overline{x_2}, x_4], 0, 1\rangle,\ c_4 :\ \langle[x_1, x_2, \overline{x_3}], 0, 2\rangle\}$
$U = \{\overline{x_5}\}$; the rest clauses are stored as follows :
$cls(x_1) = \{c_4\}, cls(\overline{x_1}) = \{c_2\}, cls(x_2) = \{c_1\}, cls(\overline{x_2}) = \{c_3\},$
$cls(x_3) = \emptyset, cls(\overline{x_3}) = \{c_4\}, cls(x_4) = \{c_3\}, cls(\overline{x_4}) = \{c_2\},$
$cls(x_5) = \{c_1\}, cls(\overline{x_5}) = \emptyset,$

and $val(A) = \times$ (unassigned) for all literal A. Note that each clause appears twice in the collection of $cls(A)$. □

Inside of $BCPht(U)$, when $\overline{x_5}$ is popped off U, we assign $val(\overline{x_5}) = 1$, $val(x_5) = 0$ (line 5) and check $c_1 \in cls(x_5)$ (line 6). c_1 generates unit clause x_2 and is added into U (line 23) with the assignments $val(x_2) = 1$, $val(\overline{x_2}) = 0$. When x_2 is popped off U, we check $c_3 \in cls(\overline{x_2})$. c_3 generates unit clause x_4, we push x_4 into U, and assign $val(x_4) = 1$, $val(\overline{x_4}) = 0$. When x_4 is popped from U, we check $c_2 \in cls(\overline{x_4})$. c_2 generates unit clause $\overline{x_1}$, we push $\overline{x_1}$ into U and assign $val(\overline{x_1}) = 1$, $val(x_1) = 0$. When $\overline{x_1}$ is popped off U, we check $c_4 \in cls(x_1)$. c_4 is true because $val(x_2) = 1$. Now the head index of c_4 is updated to 1, and we add $c_4 = \langle[x_1, x_2, \overline{x_3}], 1, 2\rangle$ into $cls(x_2)$. Since no empty clause is found, *BCPht* returns "SAT". Finally, *BCP* returns (U, S):

$$(U, S) = (\{\overline{x_5}, x_2, x_4, \overline{x_1}\}, \emptyset).$$

Note that the two assignments $val(a) := 1; val(\overline{a}) := 0$ at line 23 of *BCPht* can be removed as the same work will be done at line 5. However, having them here often improves the performance of *BCPht*. *BCPht* can be improved slightly by checking $(val(lits(c)[e_2]) = 1)$ after line 10: if it is true, then skip clause c as c is satisfied.

In this implementation of *BCP*, once all the clauses C are read in, each occurrence of literals of non-unit clauses will be visited at most once in *BCPht*.

Proposition 3.4.18. BCP(C) *runs in* $O(n)$ *time, where* n *is the sum of the lengths of all clauses in* C.

Proof. Obviously, the initialization of *BCP* takes $O(n)$. For *BCPht*, we access a non-unit clause c when its head (or tail) literal becomes false (line 6). We skip the head (tail) literal (line 11) and check the next literal b (line 13), if it is false, we continue (line 14); otherwise, we remove c from $cls(\overline{A})$ into $cls(B)$ (line 16) and update the head (tail) index (line 17). If we cannot find a new head (tail) literal

(line 19), we have found a new unit clause (line 20). None of the literals of c is visited more than once by *BCPht*. In particular, we never access the clauses from $cls(A)$ when literal A is true. □

Combining the above proposition with Theorem 3.4.4, we have the following result:

Theorem 3.4.19. BCP *is a linear time decision procedure for Horn clauses.*

3.5 Exercise Problems

1. Prove formally that the following statements are equivalent: For any two formulas A and B, (a) $A \models B$; (b) $A \rightarrow B$ is valid; (c) $A \wedge \neg B$ is unsatisfiable.

2. Use the semantic tableaux method to decide if the following formulas are valid or not.
$$\begin{array}{ll} (a) & (A \leftrightarrow (A \wedge B)) \rightarrow (A \rightarrow B) \\ (b) & (B \leftrightarrow (A \vee B)) \rightarrow (A \rightarrow B) \\ (c) & (A \oplus B) \rightarrow ((A \vee B) \wedge (\neg A \vee \neg B)) \\ (d) & (A \leftrightarrow B) \rightarrow ((A \vee \neg B) \wedge (\neg A \vee B)) \end{array}$$

3. Use the semantic tableaux method to decide if the following entailment relations are true or not.
$$\begin{array}{ll} (a) & (A \wedge B) \rightarrow C \models (A \rightarrow C) \vee (B \rightarrow C) \\ (b) & (A \wedge B) \rightarrow C \models A \rightarrow C \\ (c) & A \rightarrow (B \rightarrow C) \models (A \rightarrow B) \rightarrow (A \rightarrow C) \\ (d) & (A \vee B) \wedge (\neg A \vee C) \models B \vee C \end{array}$$

4. Prove by the semantic tableau method that the following statement, "either the debt ceiling isn't raised or expenditures don't rise" $(\neg d \vee \neg e)$, follows logically from the statements in Problem 31, Chapter 2.

5. Prove that if all the axioms are tautologies and all the inference rules are sound, then all the derived formulas from the axioms and the inference rules are tautologies.

6. Specify the introduction and elimination rules for the operators $\oplus$ and $\leftrightarrow$ in natural deductive system.

7. Estimate the upper bound of the size of the inference graph for resolution, if the input clauses contain n variables.

8. Use the Tseitin encoding to convert the following formulas into 3CNF (CNF in which every clause has at most three literals):

   (a) $(A \leftrightarrow (A \wedge B)) \rightarrow (A \rightarrow B)$
   (b) $(A \oplus B) \rightarrow (A \vee B)$
   (c) $((A \wedge B) \rightarrow C) \rightarrow (A \rightarrow C)$
   (d) $((A \vee B) \wedge (\neg A \vee C)) \rightarrow (B \vee C)$

9. Use the resolution method to decide if the following formulas are valid or not. If the formula is valid, provide a resolution proof; if it is not valid, provide an interpretation which falsifies the formula.

   (a) $(A \leftrightarrow (A \wedge B)) \rightarrow (A \rightarrow B)$
   (b) $(B \leftrightarrow (A \vee B)) \rightarrow (A \rightarrow B)$
   (c) $(A \oplus B) \rightarrow ((A \vee B) \wedge (\neg A \vee \neg B))$
   (d) $((\overline{A} \downarrow B) \uparrow \overline{C}) \rightarrow (A \uparrow (B \downarrow C))$

10. Use the resolution method to decide if the following entailment relations are true or not. If the relation holds, provide a resolution proof; if it doesn't, provide an interpretation such that the premises are true, but the consequence of the entailment is false.

    (a) $(A \rightarrow C) \wedge (B \rightarrow D) \models (A \vee B) \rightarrow (C \vee D)$
    (b) $(A \wedge B) \rightarrow C \models (A \rightarrow C)$
    (c) $A \rightarrow (B \rightarrow C) \models (A \rightarrow B) \rightarrow (A \rightarrow C)$
    (d) $(A \vee B) \wedge (\neg A \vee C) \models B \vee C$

11. Apply clause deletion strategies to each of the following sets of clauses, and obtain an equally satisfiable simpler set.

    $S_1 = \{(p \mid q \mid r), (\overline{p} \mid q \mid \overline{r} \mid s), (p \mid \overline{q} \mid s), (\overline{q} \mid r), (q \mid \overline{r})\}$
    $S_2 = \{(p \mid q \mid r \mid s), (\overline{p} \mid q \mid \overline{r} \mid \overline{s}), (p \mid q \mid s), (q \mid \overline{r}), (q \mid s \mid \overline{r})\}$
    $S_3 = \{(p \mid q \mid s), (\overline{p} \mid q \mid \overline{r} \mid \overline{s}), (p \mid r \mid s), (\overline{p} \mid \overline{r}), (\overline{p} \mid q \mid \overline{r})\}$

12. Given the clause set S, where $S = \{1.(t \mid \overline{e} \mid d)$, $2.(\overline{t} \mid c)$, $3.(e \mid \overline{d} \mid i)$, $4.(\overline{g} \mid \overline{d} \mid i)$, $5.(t \mid c \mid \overline{d} \mid g)$, $6.(\overline{c})$, $7.(\overline{i} \mid \overline{g})$, $8.(c \mid d)$, $9.(c \mid e)$ $\}$. Find the following resolution proofs for S: (a) unit resolution; (b) input resolution; (c) positive resolution; and (d) ordered resolution using the order $t > i > g > e > d > c$.

13. Apply various resolution strategies to prove that the following clause set S entails $\neg s$: unit resolution, input resolution, negative resolution, positive resolution, and ordered resolution using the order of $p > q > r > s$.

    $S = \{1.(p \mid q \mid \overline{r}), 2.(p \mid \overline{q} \mid s), 3.(\overline{p} \mid q \mid \overline{r}), 4.(\overline{p} \mid \overline{q} \mid r), 5.(\overline{p} \mid q \mid r), 6.(\overline{q} \mid \overline{r}), 7.(r \mid \overline{s})\}$

    Please show the proofs of these resolution strategies.

14. Decide by ordered resolution if the clause set S is satisfiable or not, where $S = \{1.\ (a \mid c \mid \overline{d} \mid f),\ 2.\ (a \mid d \mid \overline{e}),\ 3.\ (\overline{a} \mid c),\ 4.\ (b \mid \overline{d} \mid e),\ 5.\ (b \mid d \mid \overline{f}),\ 6.\ (\overline{b} \mid \overline{f}),\ 7.\ (c \mid \overline{e}),\ 8.\ (c \mid e),\ 9.\ (\overline{c} \mid \overline{d})\ \}$, and the order is $a > b > c > d > e > f$. If the clause set is satisfiable, please display all the resolvents from ordered resolution and construct a model from these clauses. If the clause is unsatisfiable, please display an ordered resolution tree of the proof.

15. Prove by resolution that $S \models \neg f$, where S is the set of clauses given in the previous problem. Please display the resolution tree of this proof.

16. If head/tail literals are used for implementing BCP, what literals of the clause set S in the previous problem will be assigned a truth value during unit resolution before an empty clause is found? in what order?

17. A set C of clauses is said to be a *2CNF* if every clause contains at most two literals. The *2SAT* problem is to decidable if a 2CNF formula is satisfiable or not. Show that *orderedResolution* takes $O(n^3)$ time and $O(n^2)$ space to solve 2SAT, where n is the number of propositional variables in 2CNF.

18. Prove Theorem 3.4.8: For any set C of clauses, if there exists a unit resolution proof from C, then there exists an input resolution proof from C.

CHAPTER 4
PROPOSITIONAL SATISFIABILITY

Propositional satisfiability (SAT) is the problem of deciding if a propositional formula is satisfiable, i.e., the formula has a model. Software programs for solving SAT are called SAT solvers. SAT has long enjoyed a special status in computer science. On the theoretical side, it is the first NP-complete problem ever discovered. NP-complete problems are notorious for being hard to solve; in particular, in the worst case, the computation time of any known solution for a problem in this class increases exponentially with the input size. On the practical side, since every NP problem can transform into SAT, an efficient SAT solver can be used to solve many practical problems. SAT found several important applications in the design and verification of hardware and software systems,and in many areas of artificial intelligence and operation research. Thus, there is strong motivation to develop practically useful SAT solvers. However, the NP-completeness is cause for pessimism, since it is unlikely that we will be able to scale the solutions to large practical instances.

In theory, the decision procedures presented in the previous chapter, such as semantic tableau or resolution, can be used as SAT solvers. However, they served largely as academic exercises with little hope of seeing practical use. Resolution as a refutation prover searches for an empty clause. Semantic tableau imitates the transformation of a formula into DNF. None of these decision procedures aims at searching for a model.

Fortunately, in the last two decades or so, several research developments have enabled us to tackle instances with thousands of variables and millions of clauses. SAT researchers introduced techniques such as conflict-driven clause learning, novel branching heuristics, and efficient unit propagation. These techniques form the basis of many modern SAT solvers. Using these ideas, contemporary SAT solvers can often handle practical very large SAT instances.

Annual competitions of SAT solvers promote the continuing advances of SAT solvers. SAT solvers are the engines of modern model checking tools. Contemporary automated verification techniques such as bounded model checking, proof-based abstraction, interpolation-based model checking, are all based on SAT solvers and

their extensions. These tools are used to check the correctness of hardware designs.

4.1 The DPLL Algorithm

In 1960 David and Putnam proposed resolution for propositional logic. For realistic problems, the number of clauses generated by resolution grows quickly. To avoid this explosion, Davis, Logemann, and Loveland suggested to replace the resolution rule with a case split: Pick a variable p and consider the two cases $p \mapsto 1$ and $p \mapsto 0$. This modified algorithm is commonly referred to as the *DPLL* algorithm, where DPLL stands for Davis, Putnam, Logemann, and Loveland.

DPLL can be regarded as a search procedure: The search space is all the interpretations, partial or full. A partial interpretation can be represented by a set of literals, where each literal in the set is assigned to be true and each literal outside of the set is assumed unassigned. *DPLL* starts with an empty interpretation and tries to extend the interpretation by adding either p or $\overline{p}$ through the case split. *DPLL* stops when every variable is assigned without contradiction, and the partial interpretation it maintains becomes a model.

4.1.1 Recursive Version of *DPLL*

In the following, we give a recursive version of *DPLL*. The first call to *DPLL* with the input clauses C and $\sigma = \{\}$. It is one of the oldest backtracking procedures. *DPLL* uses the algorithm *BCP* heavily for unit propagation. From the previous chapter, we know that *BCP* implements unit resolution and subsumption deletion, and can be done in linear time. Given a set C of clauses, *BCP*(C) returns $\bot$ if an empty clause is found by unit resolution; otherwise, *BCP*(C) returns (U, S), such that $C \equiv (U \cup X)$, U is a set of unit clauses, S does not contain any unit clauses, U and S share no variables.

Algorithm 4.1.1. *DPLL*(C, σ) takes as input a set C of clauses and a partial interpretation σ, C and σ do not share any variables. It returns a model of C if C is satisfiable; return $\bot$ if C is unsatisfiable. Procedure *pickLiteral*(S) will pick a literal appearing in S.

proc *DPLL*(C, σ)
1 $res :=$ *BCP*(C);
2 **if** $(res = \bot)$ **return** $\bot$; // no models for C
3 // It must be the case $res = (U, S)$
4 $\sigma := \sigma \cup U$;
5 **if** $(S = \emptyset)$ **return** σ; // a model is found
6 $A :=$ *pickLiteral*(S);

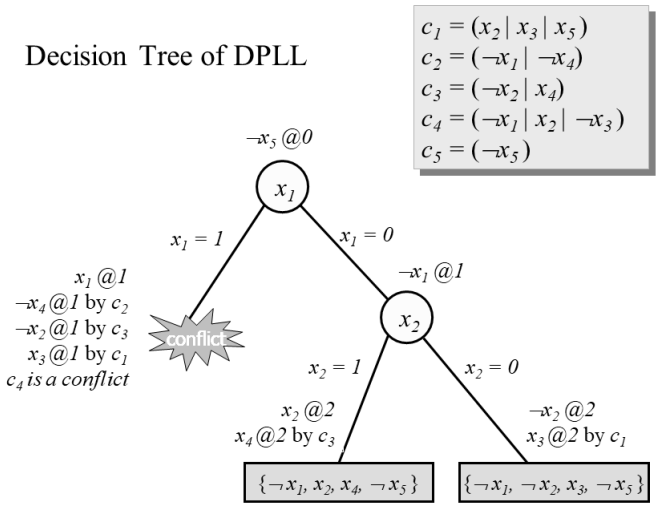


Figure 4.1.1: The result of BCP is shown in the box near each node.

7 $res := DPLL(S \cup \{(A)\}, \sigma)$;
8 **if** $(res \neq \bot)$ **return** res; // a model is found
9 **return** $DPLL(S \cup \{(\overline{A})\}, \sigma)$; // return either $\bot$ or a model

In the above procedure, we assume again $\overline{A} = p$ if A is the literal $\overline{p}$. If $BCP(C)$ does not return $\bot$, it simplifies C to $U \cup S$, where U is a set of literals, S is a set of non-unit clauses, and U and S do not share any variables. By Proposition 3.4.3, $C \equiv U \cup S$. We identify two types of literals in U.

Definition 4.1.2. *In* DPLL*, a literal $A \in U$ is said to be a* decision literal *if A is the one returned by* pickLiteral(S) *at line 6 of* DPLL*; otherwise, A is said to be an* implied literal.

Since U is a set of literals without complementary literals, U can be regarded as a partial interpretation σ such that $\sigma(U) = 1$. Since U is satisfiable and $C \equiv U \cup S$, we conclude that $C \approx S$. Furthermore, checking the satisfiability of S is reduced recursively to testing $S \wedge A$ and $S \wedge \neg A$ separately for a literal A appearing in S. This transformation, applied recursively, yields a complete decision procedure.

Example 4.1.3. Let $C = \{1 : (x_2 \mid x_3 \mid x_5),\ \ 2 : (\overline{x_1} \mid \overline{x_4}),\ 3 : (\overline{x_2} \mid x_4),\ \ 4 : (\overline{x_1} \mid x_2 \mid \overline{x_3}),\ \ 5 : (\overline{x_5})\}$. If *pickLiteral* picks a variable in the order of x_1, x_2, ..., x_5, then

$DPLL(C, \{\})$ calls
$BCP(C)$, which returns $(\{\overline{x_5}\}, C_1 = \{1' : (x_2 \mid x_3), 2., 3., 4.\})$;
x_1 is picked as the decision literal of level 1;

$DPLL(C_1 \cup \{(x_1)\}, \{\overline{x_5}\})$, which calls
    $BCP(C_1 \cup \{(x_1)\})$, which returns $\bot$;
    and returns $\bot$
$DPLL(C_1 \cup \{(\overline{x_1})\}, \{\overline{x_5}\})$, which calls
    $BCP(C_1 \cup \{(\overline{x_1})\})$, which returns $(\{\overline{x_1}\}, C_2 = \{1'., 3.\})$;
    x_2 is picked as the decision literal of level 2;
    $DPLL(C_2 \cup \{(x_2)\}, \{\overline{x_1}, \overline{x_5}\})$, which calls
        $BCP(C_2 \cup \{(x_2)\})$, which returns $(\emptyset, \{\overline{x_1}, x_2, x_4, \overline{x_5}\})$
        and returns $\{\overline{x_1}, x_2, x_4, \overline{x_5}\}$
    and returns $\{\overline{x_1}, x_2, x_4, \overline{x_5}\}$
and returns $\{\overline{x_1}, x_2, x_4, \overline{x_5}\}$

Thus a model of C can be constructed from $\{\overline{x_1}, x_2, x_4, \overline{x_5}\}$ by adding $x_3 \mapsto v$ for any value v. □

In the above example, the height of the recursion tree (also called *decision tree*) is two; x_1 and x_2 are decision literals; x_4 and $\overline{x_5}$ are implied literals. We have called *BCP* four times: one call at the root (level 0); two calls at the nodes of level one; and one call at the node of level two. If we want to search for all models, we may choose to try the branch with $x_2 \mapsto 0$, as shown in Figure 4.1.1.

Definition 4.1.4. *The* level *of an assigned variable is the depth of a node of the recursion tree of* $\mathrm{DPLL}(C, \{\})$ *at which the variable is assigned a truth value. We write* $x@l$ *if* $x \mapsto 1$ *at level* l*, and* $\overline{x}@l$ *if* $x \mapsto 0$ *at level* l*.*

In the above example, for the model returned by *DPLL*, we may write it as

$$\{\overline{x_1}@1, x_2@2, x_4@2, \overline{x_5}@0\}$$

Theorem 4.1.5. $\mathrm{DPLL}(C, \{\})$ *is a decision procedure for the satisfiability of* C*.*

Proof. Assuming $\bot \notin C$. To prove the theorem, we just need to show that *DPLL* is sound and terminating. $DPLL(C, \sigma)$ will terminate because for each recursive call, the number of variables in C is decreased by at least one. In $DPLL(C, \sigma)$, since σ and C share no variables, $(C \cup \sigma) \approx C$. So in the soundness proof of *DPLL*, we ignore the presence of σ.

We will prove by induction on the structure of the recursion tree of *DPLL*. As a base case, if $DPLL(C, \sigma)$ returns $\bot$, it means that $\bot$ is generated by unit resolution inside $BCP(C)$. By Proposition 3.4.3, $C \equiv \bot$. If $BCP(C, \sigma)$ returns (U, S), by Proposition 3.4.3, $C \equiv U \cup S$. As another base case, if $S = \emptyset$, then $C \equiv U$ and U

is the model for the unit clause set U if U is regarded as an interpretation, so C is satisfiable. If $S \neq \emptyset$, then since

$$S \equiv (S \wedge A) \vee (S \wedge \neg A)$$

for any formula A, if either $DPLL(S \cup \{p\}, \sigma)$ or $DPLL(S \cup \{\overline{p}\}, \sigma)$ returns a model, by induction hypotheses, it means $(S \wedge p) \vee (S \wedge \neg p)$ is satisfiable, so is S. If both $DPLL(S \cup \{p\}, \sigma)$ and $DPLL(S \cup \{\overline{p}\}, \sigma)$ return $\perp$, by induction hypotheses, it means $(S \wedge p) \vee (S \wedge \neg p)$ is unsatisfiable, so is S. □

4.1.2 All-SAT and Incremental SAT Solvers

Given a satisfiable set of clauses, *DPLL* returns a single satisfying assignment. Some applications, however, require us to enumerate all satisfiable assignments of a formula and this problem is called *All-SAT*. It is easy to modify *DPLL* so that all the models can be found. That is, we replace line 5 of *DPLL* as follows:

5: **if** $(S = \emptyset)$ save(σ); **return** $\perp$; // save model and continue

When the modified *DPLL* finishes, all the models are saved. Note that these models are partial models. One partial model represents 2^k full models, where k is the number of unassigned variables in the partial model.

During the execution of *DPLL*, the list of decision literals (labeled with 1) and their complements (labeled with 2) decides the current position in the recursion tree of *DPLL*. This list of literals, called *guiding path*, can be very useful for designing an incremental SAT solver: The solver takes the guiding path as input, uses the literals in the guiding path to skip the branches already explored and explores the new branches in the recursion tree for a new model.

Example 4.1.6. In Example 4.1.3, a model is found with the decision literals $\overline{x_1}$ and x_2. The guiding path for this model is $[\overline{x_1} : 2, x_2 : 1]$, where the label 2 after $\overline{x_1}$ means this is the second branch of x_1. The label 1 after x_2 means this is the first branch of x_1. If the next model is needed, it must be in the subtree with the initial guiding path $[\overline{x_1} : 2, x_2 : 2]$. Using $[\overline{x_1} : 2, x_2 : 2]$, the SAT solver avoids the search space it has traveled before. □

Since modern SAT solvers are often designed as a black box, it is cumbersome to use them as an incremental solver for enumerating each model. We can force the SAT solver to find a new model by subsequently blocking all models previously found. A model can be viewed as a conjunction of true literals, one for every

variable. The negation of this conjunction can be represented as a clause, called *blocking clause*. If this clause is added into the set of input clauses, the model will be blocked by this clause and cannot be generated by the SAT solver. Since a blocking clause contains every variable, it is unlikely to become unit and it is expensive to keep all the blocking clauses. Therefore, it is desirable to reduce the size of blocking clauses, i.e., to construct a smaller clause which still blocks the model. If we know the guiding path of this model, the negation of the conjunction of all the decision literals in the guiding path will block the model.

Many applications of SAT solvers require solving a sequence of instances by adding more clauses. Incremental SAT solvers with the guiding path can support this application without any modification, because the search space skipped by the guiding path of the first model will contain no models.

In many application scenarios, it is beneficial to be able to make several SAT checks on the same input clauses under different forced partial interpretation, called "*assumption*". For instance, people may be interested in questions like "Is the formula F satisfiable under the assumption $x \mapsto 1$ and $y \mapsto 0$?" In such applications, the input formula is read in only once, the user implements an iterative loop that calls the same solver instantiation under different sets of assumptions. The calls can be adaptive, i.e., assumptions of future SAT solver calls can depend on the results of the previous solver calls. The SAT solver can keep its internal state from the previous call to the next call. *DPLL* with a guiding path can be easily modified to support such applications, treating the assumption as a special set of unit clauses which can be added into or removed from the solver. The guiding path allows us to backtrack to the new search space but never revisit the search space visited before.

To obtain a high-performance SAT solver, we may use the deletion techniques presented in section 3.3.6 as pre-processing techniques. They enable us to reduce the size of the formula before passing it to *DPLL*.

Inside the body of *DPLL*, it calls itself twice, once with $S \cup \{(A)\}$ and once with $S \cup \{(\overline{A})\}$. A persistent data structure for S would quickly run out of memory as S is very large and the recursion tree is huge for practical SAT instances. A better approach is to use a destructive data structure for S: We remember all the operations performed onto S inside the first call, and then undo them if necessary when we enter the second call.

4.1.3 *BCPw*: Implementation of Watch Literals

In Algorithm 3.4.16, we presented an efficient implementation of *BCP* as a decision procedure for Horn clauses. This version of *BCP*, called the procedure *BCPht*, uses the head/tail data structure for each non-unit clause. Naturally, *BCPht*

can be used to support *DPLL*. Better yet, with minor modification, *BCPht* can implement the idea of *watch literals*, which is based on head/tail literals and used in solver `zchaff` by Moskewicz and Zhang.

The function of watch literals is the same as head/tail literals: Use two literals to decide if a clause becomes unit or conflicting, as stated in Proposition 3.4.15. Head/tail literals are watch literals; when head/tail literals are allowed to appear anywhere in a clause, head/tail literals are identical to watch literals.

Since the idea of watch literals is based on that of head/tail literals, the implementation of watch literals is easy if we have an implementation of head/tail literals. In *BCPht*, when a head or tail literal of c becomes false, we look for its replacement at index x, where $head(c) < x < tail(c)$. There are two alternative ways to modify *BCPht*. One solution is to allow $x \in \{0, 1, ..., |c|-1\} - \{head(c), tail(c)\}$; we will introduce an implementation of this solution shortly. The other solution is to fix $head(c) = 0$ and $tail(c) = 1$ or $|c| - 1$, and swap the elements of $lits(c)$ if necessary.

The second solution saves the space for $head(c)$ and $tail(c)$, but may cost more for visiting literals of $lits(c)$. For example, if a clause has k literals which become false in the order from left to right during *BCP*. The second solution may take $O(k^2)$ time to visit the literals of the clause, while the first solution takes $O(k)$ time. As a result, *BCP* cannot be a linear time decision procedure for Horn clauses if the second solution is used. Of course, we may introduce a pointer in the clause to remember the position of the last visited literal, so that the second solution still gives us a linear time algorithm.

Algorithm 4.1.7. *BCPw*(U) is an implementation of *BCP*(C) based on the head/tail data structure. *BCPw*(U) works the same way as *BCPht*(U), where U is a stack of unit clauses and non-unit clauses are stored using the head/tail data structure. Like *BCPht*, *BCPw* returns a conflicting clause if an empty clause is generated; returns "SAT" otherwise. Unlike *BCPht*, values of $head(c)$ and $tail(c)$ can be any valid indices of $lits(c)$.

proc *BCPw*(U)
1 **while** $U \neq \{\}$ **do**
2 $A := \text{pop}(U)$;
3 **if** $val(A) = 0$ **return** $reason(\overline{A})$; // an empty clause is found;
4 **else if** $(val(A) \neq 1)$
5 $val(A) := 1; val(\overline{A}) := 0$;
6 **for** $c \in cls(\overline{A})$ **do** // $\overline{A}$ is either head or tail of c
7 **if** $(\overline{A} = lits(c)[head(c)])$

```
8                 e_1 := head(c); e_2 := tail(c); step := 1; // scan from left to right
9               else
10                e_1 := tail(c); e_2 := head(c); step := −1; // scan from right to left
11              while true do
12                x := (e_1 + step) mod |c|; // x takes any valid index of lits(c)
13                if x = e_1 break; // exits the inner while loop
13'               if x = e_2 continue; // go to line 12
14                B := lits(c)[x];
15                if (val(B) ≠ 0)
16                  remove(c, cls(¬A)); insert(c, cls(B));
17                  if (step = 1) head(c) := x; else tail(c) := x;
18                  break; // exit the inner while loop
19              if (x = e_1) // no new head or tail found
20                A := lits(c)[e_2]; // c is unit or conflicting.
21                if val(A) = 0 return c; // c is conflicting.
22                else if (val(A) ≠ 1) // c is unit.
23                  push(A, U); reason(A) := c; val(A) := 1; val(¬A) := 0;
24    return "SAT"; // no empty clauses are found;
```

The above algorithm comes from *BCPht* (Algorithm 3.4.16) by the following modifications, which allow the values of $head(c)$ and $tail(c)$ to be any valid indices of $lits(c)$.

- At line 12, $x := x + step$ is replaced by $x := (x + step)\ mod\ |c|$.
- At line 13, the condition $x = e_2$ is replaced by $x = e_1$.
- Line 13' is added to skip the literal watched by e_2; this is the literal to be in the unit clause if the clause becomes unit.
- At line 19, the condition $x = e_2$ is replaced by $x = e_1$; when this is true, every literal of the clause is false, with the exception of the literal watched by e_2.
- At line 23, we record the reason for the assignment of a literal; this information is used at line 3 when a clause becomes conflicting.

In *DPLL*, the book-keeping required to detect when a clause becomes unit can involve a high computational overhead if implemented naively. Since it is sufficient to watch in each clause two literals that have not been assigned yet, assignments to the non-watched literals can be safely ignored. When a variable p is assigned 1, the SAT solver only needs to visit clauses watched by $\overline{p}$. Each time when one of the

watched literals becomes false, the solver chooses one of the remaining unassigned literals to watch. If this is not possible, the clause is necessarily unit, or becomes a conflicting clause, or is already satisfied under the current partial assignment. Any sequence of assignments that makes a clause unit will include an assignment of one of the watched literals. The computational overhead of this strategy is relatively low: In a formula with m clauses and n variables, $2m$ literals need to be watched, and m/n clauses are visited per assignment on average. This advantage is inherited from the idea of head/tail literals.

The key advantage of the watch literals over the head/tail literals is that the watched literals do not need to be updated upon backtracking in DPLL. That is, the key advantage of *BCPw* over *BCPht* is its use inside *DPLL*: When you backtrack from a recursive call of *DPLL*, you do not need to undo the operations on $head(c)$ and $tail(c)$, because any two positions of c will be sufficient for checking if c becomes unit.

4.1.4 Iterative Implementation of *DPLL*

Since the compiler will convert a recursion procedure into an iterative one using a stack, to avoid the expense of this conversion, in practice, the procedure *DPLL* is not implemented by means of recursion but in an iterative manner using a stack for the partial interpretation σ. If the head/tail data structure is used, σ can be obtained from $val(a)$, but we still need a stack of literals to record the time when a literal is assigned. This stack is called *trail*, which can also serve as a partial interpretation. We keep track of the recursive case-splits and their implications using the trail. Every time a literal is assigned a value, either by a decision, or an implication inside *BCP*, the literal is pushed into the trail and its level in the recursion tree is remembered.

A trail may lead to a dead-end, i.e., result in a conflicting clause, in which case we have to explore the alternative branch of one of the case splits previously made. This corresponds to backtracking which reverts one of the decisions. When we backtrack from a recursive call, we pop those literals off the trail, and make them as "unassigned" (denoted by $\times$). When backtracking enough times, the search algorithm always yields a conflicting clause or a satisfying assignment and eventually exhausts all branches.

To implement the above idea, we use the trail σ which is a stack of literals and serves as a partial interpretation. σ will keep all the assigned literals. For each literal A, we use $lvl(A)$ to record the level at which A is assigned a truth value.

This is actually done at line 5 of *BCPw*. The Line 5 of *BCPw* now is read as follows:

$$5: \quad val(A) := 1; val(\overline{A}) := 0; lvl(A) := lvl(\overline{A}) := level; \text{push}(A, \sigma);$$

where *val*, *lvl*, σ, and *level* are global variables. From now on, we call the modified *BCPw* simply as *BCP*.

In practice, it is more convenient to set the level of a right child the same as the level of its parent in the recursion tree, to indicate that the right child has no alternatives to consider. In the following implementation of *DPLL*, we have used this idea.

Algorithm 4.1.8. Non-recursive *DPLL*(C) takes as input a set C of clauses and returns a model of C if C is satisfiable; returns 0 if C is unsatisfiable. Procedure *initialize*(C) will return the unit clauses of C and store the rest clauses in the head/tail data structure.

```
proc DPLL(C)
1     U:= initialize(C); // initialize the head/tail data structure
2     σ := {}; level := 0; // initialize σ and level
3     while (true) do
4       if (BCP(U) = ⊥)
5         if (level = 0) return ⊥; // C is unsatisfiable
6         level := level − 1; // level of the second child
7         A := undo(); // A is the last literal picked at line 10
8         U := {Ā};
9       else
10        A := pickLiteral(); // pick an unassigned literal
11        if (A = nil) return σ; // all literals assigned, a model is found
12        level := level + 1; // level of the first child
13        U := {A};

proc undo()
      // Backtrack to level, undo all assignments to literals of higher levels.
      // Return the last literal of level + 1 in the trail, or nil.
1     A := nil;
2     while (σ ≠ {}) ∧ (lvl(top(σ)) > level) do
3       A := pop(σ);
4       val(A) := val(Ā) := ×; // × means "unassigned"
5     return A;
```

The procedure *undo*() will undo all the assignments (made at line 5 of *BCP*) to the literals of levels higher than *level* (including their complements), and return the last literal of $level + 1$ in σ, which must be the last decision literal A picked at line 10 of *DPLL*.

The procedure *pickLiteral*(L) (line 12) is the place for implementing various heuristics of best search strategies. It leaves the question of which literals to assign open. As we know from Theorem 4.1.5, this choice has no impact on the completeness of the search algorithm. It has, however, a significant impact on the performance of the solver, since this choice is instrumental in pruning the search space. We will address this issue later.

4.2 Conflict-Driven Clause Learning (CDCL)

This section introduces conflicting clauses as a source information to prevent the repeated exploration of futile search space. We start with a motivating example.

Example 4.2.1. Suppose C is a set of clauses over the variable set $\{x_1, x_2, ..., x_n\}$ and C has more than a million models. Let $C' = C \cup S$, where $S = \{(y_1 \mid y_2), (y_1 \mid \overline{y_2}), (\overline{y_1} \mid y_2), (\overline{y_1} \mid \overline{y_2})\}$. Then C' has no models, because the added four clauses are unsatisfiable. Now we feed C' to *DPLL* and assume that pickLiteral will prefer x_i over y_1 and y_2. Every time when *DPLL* picks y_1 for case-split, a model of C was found at that point. If y_1 is true, then one of $(\overline{y_1} \mid y_2)$ and $(\overline{y_1} \mid \overline{y_2})$ will be conflicting. If y_1 is false, then one of $(y_1 \mid y_2)$ and $(y_1 \mid \overline{y_2})$ will be conflicting. Either value of y_1 will lead to a conflicting clause in S, so *DPLL* will backtrack. Since C has more than a million models, it means *DPLL* will try to show that S is unsatisfiable more than a million times. □

If we can learn from the conflicting clauses of S, we may avoid repeatedly doing the same work again and again. This is the motivation for the technique of conflict-driven clause learning (CDCL): Learn new information from conflicting clauses and use the new information to avoid futile search effort. Using CDCL, new information is obtained in the form of clauses which are then added back into the search procedure. For the above example, when y_1 is true by case-split, $(\overline{y_1} \mid y_2)$ becomes unit, so y_2 must be set to 1. Now $(\overline{y_1} \mid \overline{y_2})$ becomes conflicting. A resolution between these two clauses on p_2 will generate a new clause, i.e., $(\overline{y_1})$. The new clause has nothing to do with the assignments of x_i, so we can jump back to the level 0 of *DPLL* and the subsequent *BCP* will find the empty clause from $S \wedge \overline{y_1}$ and *DPLL* returns $\perp$, thus avoiding finding models in C a million times.

4.2.1 Generating Clauses from Conflicting Clauses

Recall that in the last *DPLL* algorithm, when a non-unit clause c becomes unit, where c contains an unassigned literal A, we do $reason(A) := c$, i.e., c is the reason for the unassigned literal A to be true. When a conflicting clause is found in *DPLL*, all of its literals are false, and we pick the literal A, which is assigned last in c, in the clause and do resolution on A between this clause and $reason(\overline{A})$, hoping the resolvent will be useful to cut the search space. Since resolution is sound, new clauses generated this way are logical consequences of the input and it is safe to keep them. For practical reason, we must selectively do resolution and keep at most one clause per conflicting clause.

Algorithm 4.2.2. *conflictAnalysis*(c) takes as input a conflicting clause c in the current partial interpretation at level *level*, generates a new clause by resolution, and returns it. Procedure *latestAssignedLiteral*(c) will return a literal A of c such that A is the last literal assigning a truth value in c. Procedure *countLevel*(c, lvl) will return the number of literals in c which are assigned at level lvl. Procedure *resolve*(c_1, c_2) returns the resolvent of c_1 and c_2.

proc *conflictAnalysis*(c)
    // c is conflicting in the partial interpretation $val()$
1   $\alpha := c$;
2   **while** ($|\alpha| > 1$) **do** // as α has more than one literal
3     $A :=$ *latestAssignedLiteral*(α); // A is assigned last in α
4     $\alpha :=$ *resolve*($\alpha, reason(\overline{A})$); // resolution on A
5   **if** (*countLevel*($\alpha, level$) ≤ 1) **return** α;
6   **return** α;

Example 4.2.3. Let the input clauses $C = \{c_1 : (x_2 \mid x_3),\ c_2 : (\overline{x_1} \mid \overline{x_4}),\ \ c_3 : (\overline{x_2} \mid x_4),\ \ c_4 : (\overline{x_1} \mid x_2 \mid \overline{x_3})\}$. If we pick x_1 as the first decision literal in *DPLL*, it will result in the following assignments:

assignment	reason
x_1@1	(x_1)
$\overline{x_4}$@1	$c_2 : (\overline{x_1} \mid \overline{x_4})$
$\overline{x_2}$@1	$c_3 : (\overline{x_2} \mid x_4)$
x_3@1	$c_1 : (x_2 \mid x_3)$

□

Clause c_4 becomes conflicting, which contains three literals of level 1. Calling *conflictAnalysis*(c_4), we will have the following resolutions:

1. α_0 is c_4;
2. $\alpha_1 : (\overline{x_1} \mid x_2) = \mathit{resolve}(\alpha_0, c_1)$.
3. $\alpha_2 : (\overline{x_1} \mid x_4) = \mathit{resolve}(\alpha_1, c_3)$.
4. $\alpha_3 : (\overline{x_1}) = \mathit{resolve}(\alpha_2, c_2)$ and is returned.

Proposition 4.2.4. *Assuming c is conflicting in the current interpretation.*

- (a) *The procedure* conflictAnalysis(c) *will terminate and return a clause which is an entailment of the input clauses.*
- (b) conflictAnalysis(c) *will return a new conflicting clause in the current interpretation.*
- (c) *If we backtrack off the current level, the new clause returned by* conflictAnalysis(c) *will have at most one literal at level level $-$ 1.*

Proof. (a): We assigned only a finite number of literals a truth value at the current level. The literals we chose to resolve off are in the backward order when they are assigned. Once resolved off, these literals can never come back into the new clause, thus there is no chance for a loop. Since the new clause is generated by resolution from the input clauses, it must be an entailment of the input clauses.

(b): In fact, any resolvent in *conflictAnalysis*(c) is a conflicting clause in the current interpretation. Since c is a conflicting clause, every literal is false in the current interpretation. c is resolved with a reason clause which has only one true literal in the interpretation and this true literal was resolved off during the resolution. The resulting resolvent will contain only false literals in the current interpretation.

(c): The procedure *conflictAnalysis*(c) will terminate when at most one literal, say B, of the current level remains in the new clause. If we backtrack off the current level, B will be unassigned and the rest literals are false by (b). □

4.2.2 DPLL with CDCL

The new *DPLL* procedure with CDCL (conflict-driven clause learning) can be described as follows:

Algorithm 4.2.5. *DPLL*(C) takes as input a set C of clauses and returns a model of C if C is satisfiable; return $\perp$ if C is unsatisfiable. Procedure *initialize*(C) will return the unit clauses of C and store the rest clauses in the head/tail data structure.

```
proc DPLL(C)
1     U:= initialize(C); // initialize the head/tail data structure
2     σ := {}; level := 0; // initialize σ and level
3     while (true) do
4       res := BCP(U);
5       if (res ≠ "SAT") // res contains a conflicting clause
6         if (level = 0) return ⊥; // C is unsatisfiable
7         U := insertNewClause(conflictAnalysis(res)); // new level is set
8         undo(); // undo to the new level
9       else
10        A := pickLiteral(); // pick an unassigned literal
11        if (A = nil) return σ; // all literals assigned, a model is found
12        level := level + 1; // level of the first child
13        U := {A};

proc insertNewClause(α)
1     if (|α| = 1)
2       level := 0;
3       A := literal(α); reason(A) := α; // assume α = (A)
4       return {A};
5     else // insert α into the clause set
6       lits(α) := makeArray(α); // create an array of literals
7       secondHigh := 0; // look for the second highest level in α
8       for x := 0 to |α| − 1 do
9         if (lvl(lits(α)[x]) = level) head(α) := x; H := lits(α)[x]; // lvl(H) = level
10        else if (lvl(lits(α)[x]) > secondHigh) secondHigh := lvl(lits(α)[x]); t := x;
11      tail(α) := t;
12      level := secondHigh;
13      reason(H) := α;
14      return {H}; // the head literal of α
```

The procedure *insertNewClause* will return a singleton set of literals served as new unit clauses and reset the *level* variable of *DPLL*. If the new clause α generated by *conflictAnalysis* is a unit clause, this clause will be returned and the level is set to 0. If the new clause is not unit, by Proposition 4.2.4, there is only one literal of the current level in α and this literal will be the head literal of α when the level is decreased. We then pick one literal of the highest level other than the head literal as the tail literal. The level will be reset to that of the tail literal. When we backtrack to this level, by Proposition 4.2.4, α becomes unit and is the reason for the head

literal, unassigned at this point, to be assigned true.

Example 4.2.6. Consider the clause set $C = \{c_1 = (x_1 \mid x_2),\ c_2 = (x_1 \mid x_3 \mid \overline{x_7}),\ c_3 = (\overline{x_2} \mid \overline{x_3} \mid x_4),\ c_4 = (\overline{x_4} \mid x_5 \mid x_6),\ c_5 = (\overline{x_3} \mid \overline{x_4}),\ c_6 = (x_5 \mid \overline{x_6}),\ c_7 = (\overline{x_2} \mid x_3),\ c_8 = (\overline{x_1} \mid x_3),\ c_9 = (\overline{x_1} \mid x_2 \mid \overline{x_3})\}$. If the first three literals chosen by *pickLiteral* in *DPLL* are $\overline{x_7}, \overline{x_6}$, and $\overline{x_5}$, in that order, then the only implied literal is $\overline{x_4}@3$ by c_4. The next decision literal is $\overline{x_1}$, which implies the following assignments: $x_2@4$ by c_1 and $\overline{x_3}@4$ by c_3. c_7 becomes conflicting at level 4.

Calling *conflictAnalysis*(c_7), we obtain $(\overline{x_2} \mid x_4)$ by resolution between c_7 and c_3, which contains $\overline{x_2}$ as the only literal at level 4. We add $(\overline{x_2} \mid x_4)$ into *DPLL* as c_{10}.

The *level* of *DPLL* is set to 3 as $lvl(x_4) = 3$. So we backtrack to level 3, and $c_{10} = (\overline{x_2} \mid x_4)$ becomes unit. The next call to *BCP* will have make the following assignments at level 3: $\overline{x_2}@3$ by c_{10}, $x_1@3$ by c_1, $x_3@3$ by c_8, and c_9 becomes conflicting.

Calling *conflictAnalysis*(c_9), we obtain $c_{11} = (x_2)$ by two resolutions between c_8, c_9, and c_1. The new clause is unit and we set $level = 0$. Back jumping to level 0 by *undo*, the next call to *BCP* will imply the following assignments: $x_2@0$ by c_{11}, $x_3@0$ by c_7, $x_4@0$ by c_{10}, and c_5 becomes conflicting.

Calling *conflictAnalysis*(c_5), we obtain $c_{12} = (\overline{x_2})$ by two resolutions between c_5, c_{10}, and c_7. Clauses c_{11} and c_{12} will generate the empty clause. Thus, the input clauses are unsatisfiable. □

Clause learning with conflict analysis does not impair the completeness of the search algorithm: even if the learned clauses are dropped at a later point during the search, the trail guarantees that the solver never repeatedly enters a decision level with the same partial assignment. We have shown the correctness of clause learning by demonstrating that each conflicting clause is implied by the original formula.

The idea of CDCL was used first in solver `grasp` of Marques-Silva and Sakallah, and solver `relsat` of Bayardo and Schrag. In their solvers, they introduced independently a novel mechanism to analyze the conflicts encountered during the search for a satisfying assignment. There are many ways to generate new clauses from conflicting clauses and most of them are based on *implication graphs*. The method of learning new clauses through resolution is the easiest one to present.

CDCL brings a revolutionary change to DPLL so that DPLL is no longer a simple backtrack search procedure, because the level reset by *insertNewClause* may not be $level - 1$ and DPLL will backjumping to a higher level, thus avoiding unnecessary search space. If the new clause learned contains a single literal, the new level is set to 0. DPLL with CDCL can learn from failure and back jumping

to the level where the source of failure is originated. Several new techniques are proposed based on CDCL, including generating a resolution proof when the input clauses are unsatisfiable, or randomly restarting the search.

4.2.3 Unsatisfiable Cores

The existence of a resolution proof for a set of unsatisfiable clauses is guaranteed by the completeness of resolution. How to find a resolution proof is a difficult job. It has been shown that CDCL as practiced in today's SAT solvers corresponds to a proof system exponentially more powerful than resolution. Specifically, each learning step is in fact a sequence of resolution steps, of which the learned clause is the final resolvent; conversely, a resolution proof can be obtained by regarding the learning procedure as a guided resolution procedure. In Example 4.2.6, we have seen how the empty clause is generated. The resolution proof leading to the empty clause can be displayed as follows:

c_1	$(x_1 \mid x_2)$	assumed
c_3	$(\overline{x_2} \mid \overline{x_3} \mid x_4)$	assumed
c_5	$(\overline{x_3} \mid \overline{x_4})$	assumed
c_7	$(\overline{x_2} \mid x_3)$	assumed
c_8	$(\overline{x_1} \mid x_3)$	assumed
c_9	$(\overline{x_1} \mid x_2 \mid \overline{x_3})$	assumed
c_{10}	$(\overline{x_2} \mid x_4)$	resolvent of c_7, c_3
c_{11}	$(\overline{x_1} \mid x_2)$	resolvent of c_8, c_9
c_{12}	(x_2)	resolvent of c_{11}, c_1
c_{13}	$(\overline{x_2} \mid \overline{x_3})$	resolvent of c_5, c_{10}
c_{14}	$(\overline{x_2})$	resolvent of c_{13}, c_7
c_{15}	$(\ )$	resolvent of c_{12}, c_{14}

This example shows the general idea of obtaining a resolution proof when *DPLL* finds a set of clauses unsatisfiable. During the search inside *DPLL*, conflicting clauses are generated due to decision literals. The procedure *conflictAnalysis* produces a new clause to show why the previous decisions were wrong and we go up in the decision tree according to this new clause. *DPLL* returns $\perp$ when it finally finds a conflicting clause at level 0. In this case, a resolution proof is generated by *conflictAnalysis* and this proof uses either the input clauses or the clauses generated previously by *conflictAnalysis*.

Given an unsatisfiable set C of clauses, we can use all the resolution steps inside the procedure *conflictAnalysis* to construct a resolution proof. Obviously, such a proof provides an evidence for the unsatisfiability of C. The clauses used as the premises of the proof are a subset of the clauses of C and are called the

unsatisfiable *core* of the proof. Note that a formula typically does not have a unique unsatisfiable core. Any unsatisfiable subset of C is an unsatisfiable core. Resolution proofs and unsatisfiable cores have applications in hardware verification.

An unsatisfiable core is *minimal* if removing any clause from the core makes the remaining clauses in the core satisfiable. We may use this definition to check if every clause in the core is necessary, and this is obviously a difficult job when the core is huge. The core of the above example contains every input clauses except c_2, c_4, and c_6; it is a minimal core.

4.2.4 Random Restart

Suppose we are looking for a model of C by $DPLL(C)$ and the first decision literal is A. Unfortunately, $C \wedge A$ is a hard unsatisfiable instance and your *DPLL* stuck there without success. Randomly restarting *DPLL* may be your only option. By "restart", we mean that *DPLL* throws away all the previous decision literals (this can be easily done by $level := 0$; $undo()$ in *DPLL*). By "random", we mean that when you restart the search, the first decision literal will most likely be a different literal from A. Intuitively, randomly restarting means there is a chance of avoiding bad luck and getting luckier with guessing the right literal assignments that would lead to a quick solution.

Without CDCL, random restart makes *DPLL* incomplete. With CDCL, the input and generated clauses keep *DPLL* from choosing the same sequence of decision literals, and cannot generate the same clause again. Since the number of possible clauses is finite, *DPLL* cannot run forever.

The same intuition suggests random restart should be much more effective when the problem instance is in fact satisfiable. Experimental results showed that random restart also helps when the input clauses are unsatisfiable. If restart is frequent, this assumes a deviation from standard practice, CDCL can be as powerful as general resolution, while DPLL without restart has been known to correspond to the weaker tree-like resolution. The theoretical justification for the speedup is that a restart allows the search to benefit from the knowledge gained about persistently troublesome conflicting variables sooner than backjumping would otherwise allow the partial assignment to be similarly reset. In effect, restarts may allow the discovery of shorter proofs of unsatisfiability.

To implement the restart strategy, when *DPLL* has found a certain number of failures (empty clauses) without success (a model), it is time for a restart. Today's SAT solvers often use the following restart policy: Let $r_i = r_0 \gamma^{i-1}$ be the number of failures for the i^{th} restart, where r_0 is a given integer, and $1 \leq \gamma < 2$. If $r_0 = 300$ and $\gamma = 1.2$, it means the first restart happens when *DPLL* has 300 failures, the

second restart happens after another 360 failures, and so on. If $\gamma = 1$, it means *DPLL* restarts after a fixed number of failures.

Given the common use of restarts in today's clause learning SAT solvers, the task of choosing a good restart policy appears appealing. While experimental results showed that no restart policy is better than others for a wide range of problems, a clause learning SAT solver could benefit substantially from a carefully designed restart policy. Specifically, experiments show that nontrivial restart policies did significantly better than if restarts were disabled, and exhibited considerably different performance among themselves. This provides motivation for the design of better restart policies, particularly dynamic ones based on problem types and search statistics.

4.2.5 Branching Heuristics for DPLL

Using CDCL or not, every implementation of DPLL needs a function for selecting literals for case-split. We assume that when a literal is chosen for case-split, the literal will be assigned true in the first branch of DPLL. Prior to the development of the CDCL techniques, branching heuristics were the primary method used to reduce the size of the search space. It seems likely, therefore, that the role of branching heuristics is likely to change significantly for algorithms that prune the search space using clause learning.

It is conventional wisdom that it is advantageous to assign the most tightly constrained variables, i.e., variables that occur in a large number of clauses. One representative of such a selection strategy is known as the MOMS rule, which branches on a literal which has the *Maximum Occurrences in clauses of Minimum Size*. If the clauses contain any binary clauses, then the MOMS rule will choose a literal in a binary clause. By assigning true to this literal, it is likely that a maximal number of binary clauses will become satisfied. On the other hand, if the primary criterion for the selection of a branch literal is to pick one that would enable a cascade of unit propagation (the result of such a cascade is a smaller subproblem), we would assign false to the literal chosen by the MOMS rule, in the presence of binary clauses, assigning this literal false will likely create a maximal number of unit clauses from all the binary clauses. MOMS provides a rough but easily computed approximation to the number of unit propagation that a particular variable assignment might cause.

Alternatively, one can call BCP multiple times on a set of promising variables in turn, and compute the exact number of unit clauses that would be caused by a branching choice. Each chosen variable is separately fixed to true and to false and BCP is executed for each choice. The precise number of unit propagation caused is then used to evaluate possible branching choices. Unlike the MOMS heuristic, this

rule is obviously exact in its attempt to judge the number of unit clauses caused by a potential variable assignment. Unfortunately, it is also considerably more expensive to compute because of the multiple executions of BCP and undoing them. It has shown that, using MOMS to choose a small number of promising candidates, each of which is then evaluated exactly using BCP, it outperforms other heuristics on randomly generated problems.

Another strategy is to branch on variables that are likely to be *backbone literals*. A *backbone literal* is one that must be true in all models of the input clauses. The likelihood that any particular literal is a backbone literal is approximated by counting the appearances of that literal in the satisfied clauses during the execution of DPLL. This heuristic outperforms those discussed in the previous paragraphs on many examples.

The development of CDCL techniques enabled solvers to attack more structured, realistic problems. There are no formal studies comparing the previously discussed heuristics on structured problems when CDCL techniques are used. CDCL techniques create many new clauses and make the occurrence counts of each literal difficult or less significant. Branching techniques and learning are deeply related, and the addition of learning to a DPLL implementation will have a significant effect on the effectiveness of any of these branching strategies. As new clauses are learned, the number of unit clauses an assignment will cause can be expected to vary; the reverse is also true in that the choice of branch literal can affect which clauses the algorithm learns.

Branching heuristics that are designed to function well in the context of clause learning generally try to branch on literals among the new clauses which have been learned recently. This tends to allow the execution of DPLL to keep "making progress" on a single section of the search space as opposed to moving from one area to another; an additional benefit is that existing learned clauses tend to remain relevant, avoiding the inefficiencies associated with losing the information present in learned clauses that become irrelevant and are discarded.

One of the popular branch heuristics for DPLL with clause learning is called "dynamic largest individual sum (DLIS) heuristic" and it behaves similarly as the MOMS rule. At each decision point, it chooses the assignment that satisfies the most unsatisfied clauses. Formally, let p_x be the number of unresolved clauses containing x and n_x be the number of unresolved clauses containing $\overline{x}$. Moreover, let x be the variable for which p_x is maximal, and let y be variable for which n_y is maximal. If $p_x > n_y$, choose x as the next branching literal; otherwise, choose $\overline{y}$. The disadvantage of this strategy is that the computational overhead is high: the algorithm needs to visit all clauses that contain a literal that has been set to true

in order to update the values p_x and n_x for all variables contained in these clauses. Moreover, the process needs to be reversed upon backtracking.

A heuristic commonly used in contemporary SAT solvers favors literals in recently added conflict clauses. Each literal is associated with a count, which is initialized with the number of times the literal occurs in the clause set. When a learned clause is added, the count associated with each literal in the clause is incremented. Periodically, all counters divided by a constant greater than 1, resulting in a decay causing a bias towards branching on variables that appear in recently learned clauses. At each decision point, the solver then chooses the unassigned literal with the highest count (where ties are broken randomly by default). This approach, known as the variable state independent decaying sum (VSIDS) heuristics, was first implemented in `zchaff`. `Zchaff` maintains a list of unassigned literals sorted by count. This list is only updated when learned clauses are added, resulting in a very low overhead. Decisions can be made in constant time. The heuristic used in `berkmin` builds on this idea but responds more dynamically to recently learned clauses. The berkmin heuristic prefers to branch on literals that are unassigned in the most recently learned clause that is not yet satisfied.

The emphasis on variables that are involved in recent conflicting clauses leads to a locality-based search, effectively focusing on a subspace. The subspaces induced by this decision strategy tend to coalesce, resulting in more opportunities for resolution of conflicting clauses, since most of the variables are common. Representing count using integer variables leads to a large number of ties. Solver `minisat` avoids this problem by using a floating-point number to represent the weight. Another possible (but significantly more complex) strategy is to concentrate only on unresolved conflicting clauses by maintaining a stack of conflicting clauses.

When restart is supported, the branching heuristic should be a combination of random selection and other heuristics, so that DPLL will select a different branch literal after each restart. All told, there are many competing branching heuristics for satisfiability solvers, and there is still much to be done in evaluating their relative effectiveness with clause learning on realistic, structured problems.

4.3 Use of SAT Solvers

There exist excellent implementations of *DPLL* and you can find these SAT solvers at the International SAT competition web page (`satcompetition.org`). Most of them are available, free of charge, such as `minisat, glucose, lingeling, maple_LCM`, to name a few. These SAT solvers are also called general-purpose model generators, because they accept CNF as input and many problems can be specified in CNF as SAT is NP-complete.

If you have a hard search problem to solve, either solving a puzzle or finding a solution under certain constraints, you may either write a special-purpose program or describe your problem in CNF and use one of these general-purpose model generators. While both special-purpose tools and general-purpose model generators rely on exhaustive search, there are fundamental differences. For the latter, every problem has a uniform internal representation (i.e., clauses for SAT solvers). This uniform representation may introduce redundancies and inefficiencies. However, since a single search engine is used for all the problems, any improvement to the search engine is significant. Moreover, we have an accumulated knowledge of three decades on how to make such a uniform search engine efficient. Using general-purpose model generators to solve hard combinatorial problems deserves the attention for at least two reasons.

- It is much easier to specify a problem in CNF than to write a special program to solve it. Similarly, fine-tuning a specification is much easier than fine-tuning a special program.
- General-purpose model generators can provide competitive and complementary results for certain combinatorial problems. Several examples will be provided in the following.

4.3.1 Specify SAT Instances in DIMACS Format

Today SAT solvers uses the format suggested by DIMACS (the Center for Discrete Mathematics and Theoretical Computer Science) many years ago for CNF formulas. The DIMACS format uses a text file: If the first character of a line is "c", it means a comment. The CNF starts with a line "p cnf n m", where n is the number of propositional variables and m is the number of clauses. Most SAT solvers only require m be an upper bound for the number of clauses. Each literal is represented by an integer: positive integers are positive literals and negative literals are negative literals. A clause is a sequence of integers ending with 0.

Example 4.3.1. To represent $C = \{(x_1 \mid x_2 \mid \overline{x_3}), (\overline{x_1} \mid x_2), (\overline{x_2} \mid \overline{x_3}), (x_2 \mid x_3)\}$ in the DIMACS format, the text file will look as follows: □

```
c a tiny example
c
p cnf 3 4
1 2 -3 0
```

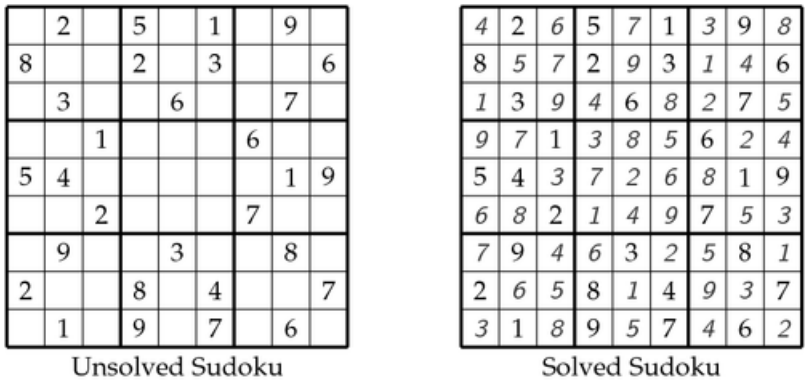


Figure 4.3.1: A typical example of Sudoku puzzle and its solution.

```
-1 2 0
-2 -3 0
2 3 0
```

To use today's SAT solvers, you need to prepare your SAT instances in the DIMACS format. This is done typically by writing a small program called *encoder*. Once a model is found by a SAT solver, you may need another program, called *decoder*, which translates the model into desired format. For example, the encoder for a Sudoku puzzle will generate a formula A in CNF from the puzzle and the decoder will take the model of A and generate the solution of the Sudoku.

4.3.2 Sudoku Puzzle

The standard Sudoku puzzle is to fill a 9×9 board with the digits 1 through 9, such that each row, each column, and each of nine 3×3 blocks contain all the digits from 1 to 9. A typical example of Sudoku puzzle and its solution is shown in Figure 4.3.1.

To solve this problem by a SAT solver, we need to specify the puzzle in CNF. At first, we need to define the propositional variables. Let $p_{i,j,k}$ be the propositional variable such that $p_{i,j,k} = 1$ iff digit k is at row i and column j of the Sudoku board. Since $1 \leq i, j, k \leq 9$, there are $9^3 = 729$ variables.

We may encode $p_{i,j,k}$ as $81(i-1)+9(j-1)+k$, then $p_{1,1,1} = 1$ and $p_{9,9,9} = 729$. Another encoding is $p_{i,j,k} = 100i + 10j + k$, then $p_{1,1,1} = 111$ and $p_{9,9,9} = 999$. The latter wasted some variables, but it allows us to see clearly the values of i, j and k from the integer. We will use the latter, since this is a very easy SAT problem, a tiny waste does not hurt. The first three lines of the CNF file may read as

```
c Sudoku puzzle
```

```
p cnf 999 1000000
```

where 1000000 is an estimate for the number of clauses from the constraints on the puzzle.

The general constraints of the puzzle can be stated as follows.

- Each cell contains exactly one digit of any value.
- Each row contains every digit exactly once, i.e., there are no duplicate copies of a digit in a row.
- Each column contains every digit exactly once.
- Each 3×3 grid contains every digit exactly once.

A solution to a given board will say which digit should be placed in which cell.

The first constraint above can be divided into two parts: (1): Each cell contains at least one digit; and (2): Each cell contains at most one digit. If every cell in a row contributes one digit and each digit appears at most once, then it implies that every digit appears exactly once in this row. The same reasoning applies to columns and grids, so the next three constraints can be simplified by dropping the constraints like "every row contains every digit at least once".

1. Each cell contains at least one digit.
2. Each cell contains at most one digit.
3. Each row contains every digit at most once.
4. Each column contains every digit at most once.
5. Each 3×3 grid contains every digit at most once.

Now it is easy to convert the above constraints into clauses:

1. For $1 \leq i, j \leq 9, (p_{i,j,1} \mid p_{i,j,2} \mid \cdots \mid p_{i,j,9})$.
2. For $1 \leq i, j \leq 9, 1 \leq k < c \leq 9, (\overline{p_{i,j,k}} \mid \overline{p_{i,j,c}})$.
3. For $1 \leq i, k \leq 9, 1 \leq j < b \leq 9, (\overline{p_{i,j,k}} \mid \overline{p_{i,b,k}})$.
4. For $1 \leq j, k \leq 9, 1 \leq i < a \leq 9, (\overline{p_{i,j,k}} \mid \overline{p_{a,j,k}})$.
5. For $0 \leq a, b \leq 2, 1 \leq i < i' \leq 3, 1 \leq j < j' \leq 3, 1 \leq k \leq 9, (\overline{p_{3a+i,3b+j,k}} \mid \overline{p_{3a+i',3b+j',k}})$.

Finally, we need to add the initial configuration of the board as clauses. If digit c is placed at row a and column b, we create a unit clause $(p_{a,b,c})$ for it. Each Sudoku puzzle should have at least 17 digits placed initially on the board.

You may write a small program to convert the above clauses in the DIMACS format and use one of the SAT solvers available on the internet to solve the Sudoku puzzle. A SAT solver takes no time to find the solution. If you like to design a Sudoku puzzle for your friend, you could use a SAT solver to check if your puzzle has no solutions or has more than one solution.

4.3.3 Latin Square Problems

Definition 4.3.2. *Given a set S of n elements, a* Latin square *L over S is an $n \times n$ matrix such that each row and each column of L is a permutation of S. The size of S, $|S|$, is called the order of L.*

A Sudoku puzzle is a Latin square of order 9 over $S = \{1, 2, ..., 9\}$. People usually use $S = Z_n = \{0, 1, ..., n-1\}$ for a Latin square of order n.

Every Latin square can be regarded as a definition of the function $* : S \times S \longrightarrow S$ such that $(a * b) = c$ iff the cell at row a and column b contains c. The pair $(S, *)$ is a special structure in abstract algebra, called *quasigroup*.

Latin squares of special properties have been extensively studied in design theory and SAT solvers have been used to solve many open problems there. For example, SAT solvers have been used successfully to show the non-existence of Latin squares of orders $n = \{9, 10, 12, 13, 14, 15\}$ satisfying the constraint $((y*x)*y)*y = x$ for $x, y \in Z_n$. It remains open if there exists such a Latin square of order 18.

To represent the constraint $((y * x) * y) * y = x$ in CNF, we need to "flatten" the constraint by introducing new variables: the constraint $((y * x) * y) * y = x$ is equivalent to

$$(y * x = z) \wedge (z * y = w) \rightarrow w * y = x.$$

If we replace $y * x = z$ by $p_{y,x,z}$, etc., we have the following clause:

$$(\overline{p_{y,x,z}} \mid \overline{p_{z,y,w}} \mid p_{w,y,x})$$

for $x, y, z, w \in Z_n$. Since a Latin square can be specified in CNF as we did for a Sudoku puzzle, now searching for a Latin square of order n satisfying $((y*x)*y)*y = x$ is the same as searching a model of the following clauses:

1. For $0 \leq i, j < n, (p_{i,j,0} \mid p_{i,j,2} \mid \cdots \mid p_{i,j,n-1})$.
2. For $0 \leq i, j < n, 0 \leq k < c < n, (\overline{p_{i,j,k}} \mid \overline{p_{i,j,c}})$.

3. For $0 \leq i, k < n, 0 \leq j < b < n, (\overline{p_{i,j,k}} \mid \overline{p_{i,b,k}})$.

4. For $0 \leq j, k < n, 0 \leq i < a < n, (\overline{p_{i,j,k}} \mid \overline{p_{a,j,k}})$.

5. For $0 \leq x, y, z, w < n, (\overline{p_{y,x,z}} \mid \overline{p_{z,y,w}} \mid p_{w,y,x})$.

The first four sets of clauses are copied from those for the Sudoku puzzle, and the last set is the specification of $((y * x) * y) * y = x$ in propositional logic.

4.3.4 Graph Problems

Many application problems can be modeled as a graph problem, such as representing the exam scheduling problem as graph coloring problem. In the study of NP-completeness theory, we learned to reduce SAT to some graph problems. That is, we use the algorithms for the graph problems to solve SAT. Now, since we have a SAT solver, the reduction is reversed: graph problems are reduced to SAT, i.e., we use SAT solvers to solve graph problems. The challenge is to ensure that the reduction produces a CNF of minimal size, or at least, the reduction is polynomial.

Definition 4.3.3. (graph coloring) *Given an undirected simple graph $G = (V, E)$ and a positive integer K, each number k, $1 \leq k \leq K$, represents a color. The* graph coloring problem *is to assign a color k, to each $v \in V$ such that the two ends of every edge of E receive different colors.*

Suppose $V = \{v_1, v_2, ..., v_n\}$. We introduce kn propositional variables $p_{i,k}$, $1 \leq i \leq n, 1 \leq k \leq K$, with the meaning that $p_{i,k} = 1$ iff v_i receives color k. The constraints for the graph coloring are

1. Every vertex receives exactly one color.

2. The two ends of every edge receive different color.

These constraints translate into the following clauses:

1. For $1 \leq i \leq |V|, (p_{i,1} \mid p_{i,2} \mid \cdots \mid p_{i,K})$.

2. For $1 \leq i \leq |V|, 1 \leq j < k \leq K, (\overline{p_{i,j}} \mid \overline{p_{i,k}})$.

3. For $(v_i, v_j) \in E, 1 \leq k \leq K, (\overline{p_{i,k}} \mid \overline{p_{j,k}})$.

The first two sets of clauses specify "at least one" and "at most one" for the constraint "exactly one", as we did in the Sudoku puzzle. The size of the first set of clauses is $K|V|$ (counting the number of literals); the size of the second set is

$(K-1)K/2|V|$; the size of the third set is $2K|E|$. Strictly speaking, this is not a polynomial reduction if K is not a constant. If K is large, we may assume $K = s*t$ for some integers $s, t > 1$ and introduce s new variables $q_{i,j}$, $0 \leq j < s$, such that $q_{i,j}$ is true if node i receives color c for $j*t < c \leq (j+1)*t$, and use these variables to reduce the size of the second set of clauses.

Definition 4.3.4. (clique) *Given an undirected simple graph $G = (V, E)$ and a positive integer k, the* clique problem *is to find a subset $S \subseteq V$ such that $|S| = k$ and for any two vertices x, y of S, $(x, y) \in E$.*

Suppose $V = \{v_1, v_2, ..., v_n\}$. As a first try, let us define n propositional variables, x_i, $1 \leq i \leq n$, with the understanding that $x_i = 1$ iff $v_i \in S$. It is easy to specify if $v_i, v_j \in S$, then $(v_i, v_j) \in E$. That is, if $(v_i, v_j \notin E$, then $(\overline{x_i} \mid \overline{x_j})$. However, it will be inefficient to specify that $|S| = k$. To ensure that $|S| = k$, we consider any subset $T \subset V$ with $n - k + 1$ vertices. One of them must be in S. To specify this constraint in logic, we have the following clauses:

For $T \subset V$ and $|T| = n - k + 1$, $(\vee_{v_i \in T} x_i)$.

This is a very large set of very long clauses, as evidently the reduction from the clique to CNF is not polynomial. For example, if $n = 100$ and $k = 5$, then there are $3,921,225$ $(= 100!/(4!96!))$ clauses of length 96. Obviously, any SAT solver will have difficulty to handle them.

An alternative solution is to consider that S is stored as a list L of k vertices. We introduce additionally $k * n$ propositional variables, $p_{i,j}$, $1 \leq i \leq n, 1 \leq j \leq k$, with the understanding that $p_{i,j} = 1$ iff v_i is at position j of L. We then have the following constraints:

1. There is exactly one vertex at each position of L.
2. If $v_i \in L$, then $v_i \in S$.
3. If $v_i \in S$, then $v_i \in L$ (optional).

The above constraints, plus the old constraint, can be specified in CNF as follows:

1. For $1 \leq j \leq k, (p_{1,j} \mid p_{2,j} \mid \cdots \mid p_{n,j})$.
2. For $1 \leq j \leq k, 1 \leq i < i' \leq n, (\overline{p_{i,j}} \mid \overline{p_{i',j}})$.
3. For $1 \leq j \leq k, 1 \leq i \leq n, (\overline{p_{i,j}} \mid x_i)$.
4. For $1 \leq i \leq n, (\overline{x_i} \mid p_{i,1} \mid p_{i,2} \mid \cdots \mid p_{i,k})$.

5. For $1 \leq i, j \leq n$, if $(v_i, v_j) \notin E$, $(\overline{x_i} \mid \overline{x_j})$.

Without loss of generality, we may assume that L is a sorted list by adding more clauses, to reduce some symmetric solutions.

If $n = 100, k = 5$, (1) produces 5 clauses of length 100; (2) produces $24,750$ $(=5 * 100 * 99/2)$ clauses of length 2; (3) produces 500 clauses of length 2; and (4) produces 100 clauses of length 6, for a total of $25,355$ clauses. This is a substantial reduction from $3,921,225$ at the cost of introducing 500 new variables.

4.4 Local Search Methods and MaxSAT

Given a set C of clauses and a full interpretation σ, we may count how many clauses are false under σ. Let

$$f(C, \sigma) = |\{c : c \in C, \sigma(c) = 0\}|,$$

then $0 \leq f(C, \sigma) \leq |C|$. When $f(C, \sigma) = 0$, σ is a model of C. The maximum satisfiability problem *MAX-SAT* is thus the optimization problem where f serves as the objective function: find σ such that $f(C, \sigma)$ is minimal (or $|C| - f(C, \sigma)$ is maximal).

4.4.1 Local Search Methods for SAT

Local search methods are widely used for solving optimization problems. In these methods, we first define a search space and for each point in the search space, we define the neighbors of the point. Starting with any point, we replace repeatedly the current point by one of its better neighbors under the objective function, until the point fits the search criteria or we run out of time.

For SAT or MAX-SAT, the search space of common local search methods is the set of full interpretations. If C contains n variables, then the search space size is $O(2^n)$, which is smaller than $O(3^n)$, the search space size for exhaustive search procedures like *DPLL*, where all partial interpretations are considered. For convenience, we assume that a full interpretation σ is represented by a set of n literals where each variable appears exactly once (or equivalently, a full minterm). The *neighbor* of σ is obtained by flipping the truth value of one variable in σ. Thus, every σ has n neighbors:

$$\mathit{neighbors}(\sigma) = \{\sigma - \{A\} \cup \{\neg A\} \mid A \in \sigma\}$$

Selman et al. proposed a greedy local search procedure called *GSAT* for SAT. The running time of *GSAT* is controlled by two parameters: $MaxTries$ is the

maximal number of the starting points; and $MaxFlips$ is the maximal number of flips allowed for any starting point.

Algorithm 4.4.1. $GSAT(C)$ takes as input a set C of clauses and returns a model of C if it finds one; returns "unknown" if it terminates without finding a model.

```
proc GSAT(C, σ)
1    for i := 1 to MaxTries do
2        σ := a random full interpretation of C; // a starting point
3        for j := 1 to MaxFlips do
4            if σ(C) = 1 return σ; // a model is found
5            pick A ∈ σ; // pick a literal of σ for flipping
6            σ' = σ − {A} ∪ {¬A};
7            σ = σ';
8    return "unknown";
```

Example 4.4.2. Let $C = \{c_1 : (x_1 \mid x_3 \mid x_4),\ \ c_2 : (\overline{x_1} \mid \overline{x_4}),\ c_3 : (\overline{x_2} \mid \overline{x_4}),\ \ c_4 : (\overline{x_1} \mid x_2 \mid \overline{x_3}),\ \ c_5 : (x_2 \mid \overline{x_4})\}$. If the starting interpretation is $\sigma_0 = \{x_1, x_2, x_3, x_4\}$, $f(C, \sigma_0) = 2$ as $\sigma_0(c_2) = \sigma_0(c_3) = 0$. If x_1 is chosen for flipping, we obtain $\sigma_1 = \{\overline{x_1}, x_2, x_3, x_4\}$ and $f(C, \sigma_1) = 1$ since $\sigma_1(c_3) = 0$. If x_2 is chosen next for flipping, then $\sigma_2 = \{\overline{x_1}, \overline{x_2}, x_3, x_4\}$ and $f(C, \sigma_2) = 1$ since $\sigma_2(c_5) = 0$. If x_4 is chosen next for flipping, then $\sigma_3 = \{\overline{x_1}, \overline{x_2}, x_3, \overline{x_4}\}$ and $f(C, \sigma_3) = 0$, i.e., σ_3 is a model of C. □

There exist many variations of $GSAT$, as there are many ways to pick a literal at line 5. In general, we need to pick a literal $A \in neighbors(\sigma)$ such that the resulting interpretation σ', here $\sigma' = \sigma - \{A\} \cup \{\neg A\}$, satisfies $(f(C, \sigma') \leq f(C, \sigma))$. We may randomly pick one of such literals, or pick the best one such that $f(C, \sigma')$ is minimal (the original $GSAT$ uses this strategy).

When $f(C, \sigma') = f(C, \sigma)$, $GSAT$ still chooses to move from σ to σ', and such moves are called "sideways" moves; they are moves that do not increase or decrease the total number of unsatisfied clauses. To avoid $GSAT$ going back and forth between two neighboring interpretations of the same f value, the idea of *tabu search* is needed, so that recent visited interpretations are not allowed to be the next interpretation. The search of GSAT typically begins with a rapid greedy descent towards a better truth assignment, followed by long sequences of "sideways" moves. Experiments indicate that on many formulas, $GSAT$ spends most of its time on sideways moves.

We can also allow moves that increase the value of f (called *upward moves*) occasionally, especially in the beginning of the execution. For example, using the idea of "simulated annealing", upward moves can be allowed with a probability, and

this probability decreases as the number of tries increases. To implement this idea, we replace line 7 by the following two lines:

7 $x :=$ random(0, 1); // x is a random number in [0, 1)
7' **if** $(x < p \vee (f(C, \sigma') \leq f(C, \sigma))$ $\sigma := \sigma'$;

Selman et al. proposed a variation of *GSAT*, called `walksat`, which interleaves the greedy moves of *GSAT* with random walk moves of a standard Metropolis search. `Walksat`'s strategy for picking a literal to flip is as follows:

1. Randomly pick a conflicting clause, say α;
2. If there is a literal $B \in \alpha$ such that the flipping of B does not turn any currently satisfied clauses to unsatisfied, return B as the picked literal;
3. With probability p, randomly pick $B \in \alpha$ and return B;
4. With probability $1 - p$, pick $B \in \alpha$ such that the flipping of B turns the least number of currently satisfied clauses to unsatisfied, return B.

Since the early 1990s, there has been active research on designing, understanding, and improving local search methods for SAT. In the framework of GSAT, various ideas are proposed and they include the following:

- When picking a literal, take the literal that was flipped longest ago for breaking ties.
- A weight is given to each clause, incrementing the weight of unsatisfied clauses by one for each interpretation. One of the literals occurring in more clauses of maximum weight is picked.
- A weight is given to each variable, which is increased in variables that are often flipped. The variable with minimum weight is chosen.

Selman et al. showed through experiments that *GSAT* as well as `walksat` substantially outperformed the best DPLL-based SAT solvers on some classes of formulas, including randomly generated formulas and graph coloring problems.

GSAT returns either a model or "unknown", as it can never declare that "the input clauses are unsatisfiable". This is a typical feature of local search methods as they are incomplete in the sense that they do not provide the guarantee that it will

eventually either report a satisfying assignment or declare that the given formula is unsatisfiable.

There have also been attempts at hybrid approaches that explore the combination of ideas from DPLL methods and local search techniques. There has also been work on formally analyzing local search methods, yielding some of the best $o(2^n)$ time algorithms for SAT. For instance, the expected running time, ignoring polynomial factors, of a simple local search method with restarts after every $3n$ "flips" has been shown to $(k+1)/k)^n$ for kCNF instances of n variables, where each clause contains at most k literals. When $k = 3$, the result yields a complexity of $(4/3)^n$ for 3SAT. This result has been derandomized to yield a deterministic algorithm with complexity 1.481^n (up to polynomial factors) to solve 3SAT.

4.4.2 2SAT versus Max2SAT

Let Max-kSAT denote the decision version of MaxSAT which takes a positive integer m and a set C of clauses, where each clause has no more than k literals, as input and returns true iff there exists an interpretation σ such that at least m clauses are satisfied under σ. It is well-known that 2SAT can be solved in polynomial time, while Max2SAT is NP-complete.

Theorem 4.4.3. *There is a linear time decision procedure for 2SAT.*

Proof. This sketch of the proof comes from (Aspvall, Plass, Tarjan, 1979). Let C be in 2CNF without pure literals. We construct a directed graph $G = (V, E)$ for C as follows: V contains the two literals for each variable appearing in C. For each clause $(A \mid B) \in C$, we add two edges, $(\overline{A}, B)$ and $(\overline{B}, A)$, into E. Obviously, the edges represent the implication relation among the literals, because $(A \mid B) \equiv (\overline{A} \rightarrow B) \wedge (\overline{B} \rightarrow A)$. Since the implication relation is transitive and $(A \rightarrow B) \equiv (\overline{B} \rightarrow \overline{A})$, the following three statements are equivalent for any pair of literals x and y in G:

1. $C \models (x \rightarrow y)$;
2. there is a path from x to y in G; and
3. there is a path from $\overline{y}$ to $\overline{x}$ in G.

We then run the algorithm for strongly connected components (SCC), which can be done in linear time, on G. It is easy to see that $C \models (x \leftrightarrow y)$ for any two literals x and y in the same component, because x and y share a cycle in G. If both a variable, say p, and its complement, $\overline{p}$, appear in the same component, then C is unsatisfiable, because $C \models (p \leftrightarrow \overline{p})$. If no such cases exist, then C must be

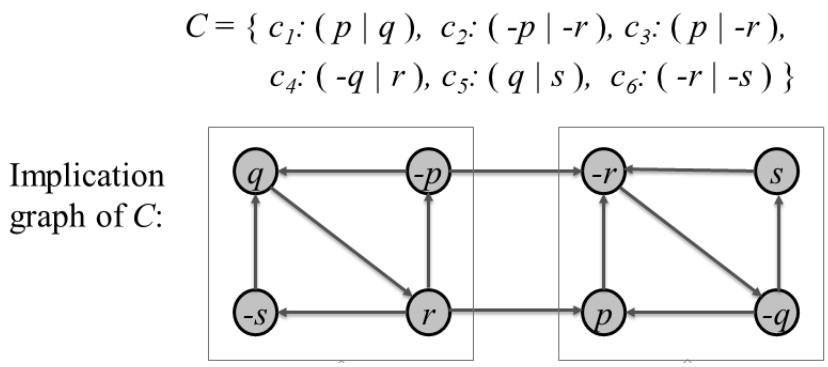


Figure 4.4.1: The implication graph for 2CNF C and the SCC.

satisfiable. A model can be obtained by assigning the same truth value to each node in the same component, following the topological order of the components induced by G: Always assign 0 to a component, unless one of the literals in the component had already received 1. □

Example 4.4.4. Given $C = \{c_1 : (p \mid q),\ c_2 : (\overline{p} \mid \overline{r}),\ c_3 : (p \mid \overline{r}),\ c_4 : (\overline{q} \mid r),\ c_5 : (q \mid s),\ c6 : (\overline{r} \mid \overline{s})\}$, its implication graph is shown in Figure 4.4.1. There are two SCCs: $S_1 = \{\overline{p}, q, r, \overline{s}\}$ and $S_2 = \{p, \overline{q}, \overline{r}, s\}$. We have to assign 0 to every node in S_1 (and 1 to every node in S_2) to obtain a model of C. □

Theorem 4.4.5. *Max2SAT is NP-complete.*

Proof. This proof comes from (Garey, Johnson, and Stockmeyer, 1976). Max2SAT is in NP because we can guess non-deterministically an interpretation σ and check if there are at least K clauses are true under σ in polynomial time.

To show Max2SAT is NP-hard, we reduce 3SAT to Max2SAT. That is, if there exists an algorithm A to solve Max2SAT, we can construct algorithm B to solve 3SAT, and the computing times of A and B differ polynomially.

Given any instance C of 3SAT, we assume that every clause of C has exactly three literals. For every clause $c_i = (l_1 \mid l_2 \mid l_3) \in C$, where l_1, l_2, and l_3 are arbitrary literals, we create 10 clauses in Max2SAT:

$$(l_1), (l_2), (l_3), (x_i), (\overline{l_1} \mid \overline{l_2}), (\overline{l_1} \mid \overline{l_3}), (\overline{l_2} \mid \overline{l_3}), (l_1 \mid \overline{x_i}), (l_2 \mid \overline{x_i}), (l_3 \mid \overline{x_i}),$$

where x_i is a new variable. Let C' be the collection of these clauses created from the clauses in C, then $|C'| = 10|C|$ and the following statements are true:

- If an interpretation satisfies c_i, then we can make seven of the ten clauses satisfied.

- If an interpretation falsifies c_i, then at most six of the ten clauses can be satisfied.

The above statements can be verified by a truth table on l_1, l_2, l_3 and x_i:

l_1	l_2	l_3	x_i	$(\overline{l_1}\|\overline{l_2})$	$(\overline{l_1}\|\overline{l_3})$	$(\overline{l_2}\|\overline{l_3})$	$(l_1\|\overline{x_i})$	$(l_2\|\overline{x_i})$	$(l_3\|\overline{x_i})$	sum
0	0	0	0	1	1	1	1	1	1	6
0	0	0	1	1	1	1	0	0	0	4
0	0	1	0	1	1	1	1	1	1	7
0	0	1	1	1	1	1	0	0	1	6
0	1	1	0	1	1	0	1	1	1	7
0	1	1	1	1	1	0	0	1	1	7
1	1	1	0	0	0	0	1	1	1	6
1	1	1	1	0	0	0	1	1	1	7

The first four columns serve both the truth values of $\{l_1, l_2, l_3, x_i\}$ and the first four clauses of the ten clauses. The first two lines of the table shows that when $(l_1 \mid l_2 \mid l_3)$ is false, the sum of the truth values of the ten clauses (the last column) is at most 6. The rest of the table shows the sums of the truth values of the ten clauses when one, or two, or three literals of $\{l_1, l_2, l_3\}$ are true (some cases are ignored due to the symmetry of l_1, l_2, and l_3). It is clear from the table that when $(l_1 \mid l_2 \mid l_3)$ is true, we may choose the value of x_i to obtain 7 true clauses out of the ten clauses. Then C is satisfiable iff C' has an interpretation which satisfies $K = 7|C|$ clauses of C'. If we have an algorithm to solve C' with $K = 7|C|$, then the output of the algorithm will tell if C is satisfiable or not. □

The above theorem shows that MaxSAT is a hard problem even for 2CNF.

4.5 Maximum Satisfiability

The algorithm *GSAT* can be used to solve the MaxSAT problem, which asks to find an interpretation for a set of clauses such that a maximal number of clauses are true. Since *GSAT* is incomplete, the answer found by *GSAT* is not guaranteed to be optimal. This is a typical feature of local search methods which always find local optimal solutions. *GSAT*can also be used to solve MaxSAT of general form: weighted MaxSAT and hybrid MaxSAT.

4.5.1 Weight MaxSAT and Hybrid MaxSAT

Definition 4.5.1. *Given a set C of clauses and a function $w : C \longrightarrow \mathcal{N}$, $w(c)$ is called the* weight (or cost) *of $c \in C$. (C, w) is called* weighted CNF (WCNF).

Given (C, w), the weighted MaxSAT *problem is to find a full interpretation σ such that $\sum_{c \in C, \sigma(c)=0} w(c)$ is minimal (or equivalently, $\sum_{c \in C, \sigma(c)=1} w(c)$ is maximal). Any clause in C is called a* weighted clause *and denoted by $(c; w(c))$.*

Example 4.5.2. Let C be a weighted MaxSAT instance, where $C = \{c_1 : (x_1; 5),\ c_2 : (\overline{x_1} \mid x_2; 4),\ c_3 : (x_1 \mid \overline{x_2} \mid x_3; 3),\ c_4 : (\overline{x_1} \mid \overline{x_2}; 2),\ c_5 : (x_1 \mid x_2 \mid \overline{x_3}; 4),\ c_6 : (\overline{x_1} \mid x_3; 1),\ c_7 : (\overline{x_1} \mid \overline{x_2} \mid \overline{x_3}; 2)\}$. Then $\sigma = \{x_1, x_2, \overline{x_3}\}$ is the optimal interpretation for C, where c_4 and c_6 are false, with the total weights of falsified clauses being 3; the sum of weights of true clauses is 18. □

Definition 4.5.3. *Given a WCNF (C, w), let (H, S) be a partition of the set C, where each clause $c \in H$ has weight $w(c) = \infty$, a* solution *of (H, S) is a full interpretation σ such that every clause of H is true in σ. The* hybrid MaxSAT *problem is to find an optimal solution such that $\sum_{c \in S, \sigma(c)=0} w(c)$ is minimal. The clauses in H are called* hard clauses *and the clauses in S are called* soft clauses.

Hybrid MaxSAT is also called *weighted partial MaxSAT*. The definition of hybrid MaxSAT assumes that hybrid MaxSAT is a special case of weighted MaxSAT. Weighted MaxSAT can be also viewed as a special case of hybrid MaxSAT when $H = \emptyset$. In other words, the two definitions are equivalent in expressive power. In practice, ∞ can be replaced by the sum of all weights in S. Separating H from S will allow us to handle H efficiently, using advanced techniques for SAT.

Since MaxSAT is a special case of weighted MaxSAT or hybrid MaxSAT, weighted MaxSAT or hybrid MaxSAT must be NP-complete, too. In fact, it is very easy to reduce NP-complete problems to weighted MaxSAT or hybrid MaxSAT.

The clique problem is a well-known NP-complete problem in graph theory. Given a graph $G = (V, E)$, let $V = \{v_1, v_2, ..., v_n\}$. We define n propositional variables x_i, $1 \leq i \leq n$, with the meaning that $x_i = 1$ iff v_i is in the solution of a maximum clique. Then we may reduce the clique problem to the weighted MaxSAT problem by converting $G = (V, E)$ into a formula of 2WCNF, which consists of the following two sets of weighted clauses.

- For $1 \leq i \leq n$, create $(x_i; 1)$.
- For $v_i, v_j \in V$, $i \neq j$ and $(v_i, v_j) \notin E$, create $(\overline{x_i} \mid \overline{x_j}; n)$.

The first set of clauses says that if a vertex is chosen in the solution, there is a reward of 1 (equivalently, if the vertex is not in the solution, there is a penalty of 1). The second set of clauses specifies the property of a clique: If two vertices are chosen in the solution and there is no edge between them, there is a penalty of $|V|$.

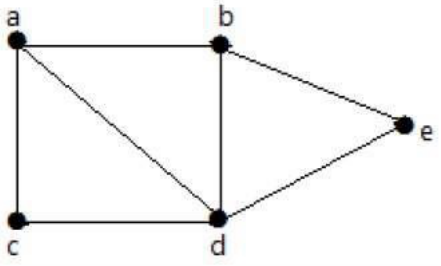


Figure 4.5.1: Finding a maximal clique in a graph.

Example 4.5.4. For the graph in Figure 4.5.1, the clauses are

$$C = \{(\overline{x_a} \mid \overline{x_e}; 5), (\overline{x_b} \mid \overline{x_c}; 5), (\overline{x_c} \mid \overline{x_e}; 5), (x_a; 1), (x_b; 1), (x_c; 1), (x_d; 1), (x_e; 1)\}.$$

One of the optimal solutions is $\sigma = \{x_a \mapsto 1, x_b \mapsto 1, x_c \mapsto 0, x_d \mapsto 1, x_e \mapsto 0\}$, under which, two clauses, i.e., (x_c) and (x_e), are false and the total penalty is 2. □

Consider an interpretation σ in which one variable x_i is true; all the other variables are false. Given σ, all the clauses in the second set will be true and all the clauses in the first set will be false, except one. That is, the sum of weights of the falsified clauses is $n-1$ under σ. Thus, the total weights of all falsified clauses in any optimal solution cannot be greater than $n-1$. In other words, all the clauses in the second set must be true in an optimal solution. This implies that there must be an edge between two vertices in the solution. The optimal solution picks a maximal number of x_i's and thus gives us a maximal clique. It is natural to specify the second set of clauses as hard clauses in the hybrid MaxSAT format.

4.5.2 The Branch-and-Bound Algorithm

To obtain an optimal solution of MaxSAT or hybrid MaxSAT, the traditional approach is to use the branch-and-bound algorithm. In these algorithms, we often cast MaxSAT as a minimization problem: find solution with minimum (optimal) cost. The branching part of the branch-and-bound algorithm is very much like *DPLL*, doing case-splits on selected literals. The bounding part of the algorithm checks if an estimation of the current solution is better or not than the solution found so far. If not better, the algorithm cuts this futile branch and backtracks. If we look for a minimal solution, the estimation produces a lower bound; for a maximal solution, the estimation produces an upper bound. For example, if we look for a minimal solution, and already have a solution of value 30, if the estimation says the lower bound is 32, then there is no point to continue, because the value of the solution from the current branch cannot be lower than 32.

Here we present the branch-and-bound algorithm for hybrid MaxSAT in the style of recursive *DPLL* (Algorithm 4.1.1).

Algorithm 4.5.5. *HyMaxSATBB*$(H, F, \sigma, soln)$ takes as input a set H of hard clauses, a set F of weighted soft clauses, a partial interpretation σ, and the best solution *soln* found so far. H and σ do not share any variables. It looks for a full interpretation σ which is a model of H and the sum of the weights of the false clauses in F is minimal under σ; this weight sum will be returned by the algorithm. Procedure *pickLiteral*(S) will pick a literal appearing in S.

proc *HyMaxSATBB*$(H, F, \sigma, soln)$
1 $res := BCP(H)$;
2 **if** $(res = \bot)$ **return** ∞; // hard causes violated
3 // Assume $res = (U, S)$
4 $\sigma := \sigma \cup U$;
5 $F :=$ *simplifySoft*(F, σ);
6 **if** $soln \leq$ *lowerBound*(F) **return** *soln*;
7 $A :=$ *pickLiteral*$(S \cup F)$;
8 **if** $(A = nil)$ **return** *countFalseClauseWeight*(F, σ);
9 $soln := min(soln,$ *HyMaxSATBB*$(S \cup \{(A)\}, F, \sigma, soln))$;
10 **return** $min(soln,$ *HyMaxSATBB*$(S \cup \{(\overline{A})\}, F, \sigma, soln))$;

We have seen $BCP(H)$ in Algorithm 4.1.1; the following four procedures are called inside *HyMaxSATBB*:

- *simplifySoft*(F, σ) at line 5: Simplify soft clauses F using the partial interpretation σ and keep the total weights of false clauses unchanged under any interpretation. Details will follow.

- *lowerBound*(F) at line 6: Estimate the minimal total weights of false clauses when σ is extended to a full interpretation in all possible ways. Details will be discussed later.

- *pickLiteral*$(S \cup F)$ at line 7: pick an unassigned literal for case-splits; returns *nil* if every literal has a truth value. Here is the place heuristics for branching are implemented.

- *countFalseClauseWeight*(F, σ) at line 8: a simple procedure for computing the sum of all the weights of false clauses in F. Here, σ is a full interpretation and can be saved for late use.

For an efficient implementation of *HyMaxSATBB*, the first solution is always obtained by a local search algorithm such as *GSAT*. We can use all techniques used for *DPLL*, including iterative procedure, destructive data structure, and conflict-driven clause learning for hard clauses.

For soft clauses, the treatment is different. For example, unit resolution is a powerful simplification rule for hard clauses, but it is unsound for soft clauses. If $C = \{(\overline{x_1}; 1), (x_1 \mid x_2; 1), (x_1 \mid \overline{x_2}; 1), (x_1 \mid x_3; 1), (x_1 \mid \overline{x_3}; 1)\}$, the optimal solution has one false clause when $x \mapsto 1$. If we apply unit resolution to C to get $C' = \{(\overline{x_1}; 1), (x_2; 1), (\overline{x_2}; 1), (x_3; 1), (\overline{x_3}; 1)\}$, an optimal solution of C' will contain two false clauses. Of course, if a unit clause is hard, we may apply unit resolution to both hard and soft clauses.

4.5.3 Simplification Rules and Lower Bound

Definition 4.5.6. *An inference rule or simplification rule applicable to a set C of weighted clauses is said to be* sound*, if the optimal solution of the resulting set C' of weighted clauses has the same value as the optimal solution of C.*

In other words, sound rules should preserve the value of optimal solutions. Most inference rules are unsound for weighted clauses because they generate extra clauses. One exception is when a hard clause can be generated from soft clauses. For example, given a soft clause $(c; w)$, if c becomes false, w will be added as a portion of the lower bound. If this addition of w causes the lower bound to exceed the value of the current solution, then we can generate the hard clause c to prevent c from being false. In the following, we focus on simplification rules, which replace some clauses by other clauses when we transform C to C'. Each simplification rule can be denoted by

$$C_1, C_2, .., C_m \Longrightarrow C'_1, C'_2, .., C'_n$$

when we want to replace $\{C_1, C_2, .., C_m\} \subseteq C$ by $\{C'_1, C'_2, .., C'_n\}$, where C_i, C'_j are weighted clauses.

In the following rules, α and β denote arbitrary, possibly empty, disjunction of literals.

1. **zero elimination**: Weighted clauses with zero weights can be discarded.

$$(\alpha; 0) \Longrightarrow 1$$

2. **tautology elimination**: Valid clauses are always true and can be discarded.

$$(p \mid \overline{p} \mid \alpha; w) \Longrightarrow 1$$

3. **identical merge**: Identical clauses are merged into one.

$$(\alpha; w_1), (\alpha; w_2) \Longrightarrow (\alpha; w_1 + w_2)$$

4. **unit clash**: Unit clauses with complement literals can be reduced to one.

$$(A; w_1), (\overline{A}; w_2) \Longrightarrow (A; w_1 - w_2), (\bot; w_2)$$

where A is a literal and $w_1 \geq w_2$. For example, given $(p; 2)$ and $(\overline{p}; 3)$, if $p \mapsto 1$, we get a cost of 3 from $(\overline{p}; 3)$; if $p \mapsto 0$, we get a cost of 2 from $(p; 2)$. The same result can be obtained from $(\overline{p}, 1)$ and $(\bot; 2)$; the latter is added into the cost function.

5. **weighted resolution**:

$$(p \mid \alpha; w_1), (\overline{p} \mid \beta; w_2) \Longrightarrow \begin{array}{l} (\alpha \mid \beta; u), (p \mid \alpha, w_1 - u), (\overline{p} \mid \beta; w_2 - u), \\ (p \mid \alpha \mid \overline{\beta}; u), (\overline{p} \mid \overline{\alpha} \mid \beta; u) \end{array}$$

where $w_1 > 0$, $w_2 > 0$, and $u = min(w_1, w_2)$. This rule is proposed by Larrosa et al. (2007). Since either $w_1 - u = 0$ or $w_2 - u = 0$, the above rule cannot be used forever. For example, from $(p \mid q; 2)$ and $(\overline{p} \mid q; 3)$, $\alpha = \beta = q$, $u = min(2, 3)$, we obtain five clauses: $(q; 2), (p \mid q; 0), (\overline{p} \mid q; 1)$, $(p \mid q \mid \overline{q}; 2)$, and $(\overline{p} \mid \overline{q} \mid q; 2)$. The second clause is removed by zero elimination, and the last two clauses by tautology elimination, we obtain two clauses: $(q; 2), (\overline{p} \mid q; 1)$. Note that $\overline{\beta}$ and $\overline{\alpha}$ in the last two clauses are not disjunction of literals in general. We need to convert $(p \mid \alpha \mid \overline{\beta}; u)$ (or $(\overline{p} \mid \overline{\alpha} \mid \beta; u)$) into a set of clauses, each having the weight u. In practice, this rule often applies to short clauses or when $\alpha = \beta$.

It is easy to check all of the above simplification rules are sound for weighted clauses, as they preserve optimal solutions.

To efficiently implement *simplifySoft*(F, σ), F will be simplified first by σ, as σ represents a set of hard unit clauses. Every true clause under σ will be removed from F and every false literal under σ will be removed from the clauses in F. After this step, F and σ do not share any variable. The second step is to apply the above simplification rules. Some rules like zero elimination and tautology elimination, are applied eagerly. Some rules like weighted resolution are applied selectively.

To implement *lowerBound*$(F, soln)$, at first, we assumed that F is simplified and we count the total weights of all false clauses. If the sum exceeds the current solution, we exit.

For non-false clauses of F, there exist various techniques to estimate the lower bound. One technique is to formulate F as an instance of Integer Linear Programming (ILP). ILP Solvers are common optimization tools for operation research. ILP solvers solve problems with linear constraints and linear objective function where some variables are integers. State-of-the-art ILP solvers are powerful and effective and can attack MaxSAT directly. In practice, ILP solvers are effective on many standard optimization problems, e.g., vertex cover, but for problems where there are many boolean constraints ILP is not as effective.

To obtain a lower bound of ILP, the constraints on integer solutions are relaxed so that ILP becomes an instance of the Linear Programming (LP), which is known to have polynomial-time solutions.

If we view $\top$ as 1 and $\bot$ as 0, it is well known that a clause can be converted into a linear inequation, such as $(\overline{x_1} \mid x_2 \mid \overline{x_3})$ becomes $(1 - x_1) + x_2 + (1 - x_3) \geq 1$, so that the clauses are satisfiable iff the set of inequations has a 0/1 solution for all variables.

For any clause c, let $P(c)$ and $N(c)$ be the sets of positive literals and negative literals in c, respectively. Let $F = \{(c_i; w_i) : 1 \leq i \leq m\}$ over n variables, say $\{x_1, x_2, ..., x_n\}$. F can be formulated as the following instance of ILP:

$$
\begin{array}{ll}
\text{minimize } \sum_{1 \leq i \leq m} w_i(1 - y_i) & \text{// minimize the weights of falsified clauses} \\
\text{subject to } \sum_{x \in P(c_i)} x + \sum_{\overline{x} \in N(c_i)} (1 - x) \geq y_i,\ 1 \leq i \leq m, & \text{// } c_i \text{ is true iff } y_i = 1 \\
\quad y_i \in \{0, 1\},\ 1 \leq i \leq m, & \text{// every clause is falsified or satisfied} \\
\quad x_j \in \{0, 1\},\ 1 \leq j \leq n. & \text{// every variable is false or true} \\
\quad \text{where } y_i \text{ is a new variable for each clause } c_i \text{ in } F. &
\end{array}
$$

Intuitively, the more y_i's with $y_i = 1$, the less the value of $\sum_{1 \leq i \leq m} w_i(1 - y_i)$. And $y_i = 1$ implies that $\sum_{x \in P(c_i)} x + \sum_{\overline{x} \in N(c_i)} (1 - x) \geq 1$, i.e., c_i is true.

The above ILP instance can be relaxed to an instance L of LP:

$$
\begin{array}{ll}
\text{minimize } \sum_{1 \leq i \leq m} w_i(1 - y_i) & \\
\text{subject to } \sum_{x \in P(c_i)} x + \sum_{\overline{x} \in N(c_i)} (1 - x) \geq y_i,\ 1 \leq i \leq m, & \\
\quad 0 \leq y_i \leq 1,\ 1 \leq i \leq m, \text{ and} & \\
\quad 0 \leq x_j \leq 1,\ 1 \leq j \leq n. &
\end{array}
$$

If L has a solution, then $\sum_{1 \leq i \leq m} w_i(1 - y_i)$ is a lower bound for the solutions of F. The ILP can be used to obtain an approximate solution for F when we round up/down the values of x_i from reals to integers.

Other techniques for lower bounds look for inconsistencies that force some soft clause to be falsified. Some suggested learning falsified soft clauses, some people

use the concepts of *clone*, *minibcukets*, or width-restricted BDD in relaxation. For instance, we may treat the soft clauses as if they were hard and then run BCP to locate falsified clauses. This can help us to find unsatisfiable subsets of clauses quickly and use them to estimate the total weights of falsified clauses. These techniques can be effective on small combinatorial problems. Once the number of variables in F gets to $1,000$ or more, these lower bound techniques become weak or too expensive. Strategies for turning on/off these techniques in *HyMaxSATBB* should be considered.

Besides the algorithm based on branch-and-bound, some MaxSAT solvers convert a MaxSAT instance into a sequence of SAT instances where each instance encodes a decision problem of the form, for different values of k, "is there an interpretation that falsifies soft clauses of total weights at most k?"

For example, if we start with a small value of k, the SAT instance will be unsatisfiable. By increasing k gradually, the first k when the SAT instance becomes satisfiable will be the value of an optimal solution. Of course, this linear strategy on k does not provide necessarily an effective MaxSAT solver.

There are many ongoing researches in this direction. The focus of this approach is mainly doing two things:

1. Develop economic ways to encode the decision problem at each stage of the sequence.

2. Exploit information obtained from the SAT solver at each stage in the next stage.

Some MaxSAT solvers use innovation techniques to obtain more efficient ways to encode and solve the individual SAT decision problems. Some solvers use unsatisfiable cores to improve SAT solving efficiency. These solvers appear promising as they are effective on some large MaxSAT problems, especially those with many hard clauses.

4.5.4 Use of Hybrid MaxSAT Solvers

Most real-world problems involve an optimization component and there are high demand for automated approaches to finding good solutions to computationally hard optimization problems. We have seen how to encode the clique problem easily and compactly into a MaxSAT instance. MaxSAT allows for compactly encoding various types of high-level finite-domain soft constraints due to Cook-Levin Theorem: Any NP constraint can be polynomially represented as a weighted SAT instance.

There exists an extension of DIMACS format for WCNF. To specify an instance of hybrid MaxSAT, the line starting with `p` will contain the keyword `wcnf`, followed by the number of variables, the number of clauses, and the label for hard clauses. That is,

```
p wcnf <# of variables> <# of clauses> <hard clause label>
```

The hard clause label is an integer larger than the maximal weight of soft clauses. Each clause starts with the weight value, followed by literals and ending with 0. For the small clique example of Example 4.5.4, the input file to a MaxSAT solver will look like:

```
c small clique example
c xa = 1, xb = 2, xc = 3, xd = 4, xe = 5
p wcnf 5 8 9
9 -1 -5  0
9 -2 -3  0
9 -3 -5  0
1 1 0
1 2 0
1 3 0
1 4 0
1 5 0
```

A clause is hard if the weight of the clause is the label for hard clauses.

The standard format for MaxSAT instances makes the annual evaluation of MaxSAT solvers possible. The latest website for MaxSAT evaluation can be found on the internet and the site for the year 2020 is given below:

```
https://maxsat-evaluations.github.io/2020/
```

Researchers on MaxSAT can assess the state-of-the-art MaxSAT solvers and create a collection of publicly available MaxSAT benchmark instances.

There MaxSAT solvers are built on the successful techniques for SAT, evolve constantly in practical solver technology, and offer an alternative to traditional approaches, e.g., integer programming. MaxSAT solvers use propositional logic as the underlying declarative language and are especially suited for inherently "very Boolean" optimization problems. The performance of MaxSAT solvers have surpassed that of specialized algorithms on several problem domains:

- correlation clustering

- probabilistic inference
- maximum quartet consistency
- software package management
- fault localization
- reasoning over bionetwork
- optimal covering arrays
- treewidth computation
- Bayesian network structure learning
- causal discovery
- cutting planes for IP (integer programming)
- argumentation dynamics

The advances of contemporary SAT and MaxSAT solvers have transformed the way we think about NP complete problems. They have shown that, while these problems are still unmanageable in the worst case, many instances can be successfully tackled.

4.6 Exercise Problems

1. Draw the four complete decision trees (trees of recursive calls) of *DPLL* on the following set of clauses,

$$\{(p \mid q), (p \mid \overline{r}), (\overline{p} \mid r), (\overline{p} \mid q \mid \overline{r}), (\overline{q} \mid \overline{r})\}$$

   using the orders of p, q, r and r, q, p, respectively, for splitting (either value can be chosen first) and with or without BCP.

2. Draw the complete decision tree of *DPLL* on the following set of clauses,

$$\{(p \mid q \mid s), (p \mid \overline{r} \mid \overline{s}), (q \mid \overline{r} \mid s), (\overline{p} \mid q \mid \overline{r}), (\overline{q} \mid \overline{r} \mid \overline{s})\}$$

   using the orders of p, q, r, s for splitting (the value 1 is used first for each splitting variable) and report the truth values of variables at various levels. How many models are found by *DPLL*? What guiding paths are found when each model is found?

3. If we feed the clause set in the previous problem to a SAT solver which works as a black box and can produce only one model at a time, what clause should be added into the clause set so that the SAT solver will produce a new model?

4. Estimate the upper bound of the size of the inference graph for resolution, if the input clauses contain n variables.

5. Provide the pseudo-code for implementing *BCPw* based on *BCPht* such that the space for $head(c)$ and $tail(c)$ are freed by assuming $head(c) = 0$ and $tail(c) = |c| - 1$ for any non-unit clause c, and no additional space can be used for c. In your code, you may call $swap(lits(c), i, j)$, which swaps the two elements at indices i and j in the array $lits(c)$.

6. Given a set H of $m + 1$ Horn clauses over m variables:

$$H = \{c_1 = (\overline{x_1} \mid \overline{x_2} \mid \cdots \mid \overline{x_{m-1}} \mid \overline{x_m}), c_2 = (\overline{x_1} \mid x_2), c_3 = (\overline{x_2} \mid x_3), ..., c_m = (\overline{x_{m-1}} \mid x_m), c_{m+1} = (x_1)\}$$

   Show that the size of H is $O(m)$, and the algorithm *BCPw* in the previous problem will take $\Omega(m^2)$ time on H, if all the clauses in $cls(A)$ must be processed before considering $cls(B)$, where $\neg A$ is assigned true before $\neg B$.

7. We apply *DPLL* (without CDCL) to the following set S of clauses: $S = \{1.\ (a \mid c \mid \overline{d} \mid f),\ 2.\ (a \mid d \mid \overline{e}),\ 3.\ (\overline{a} \mid c),\ 4.\ (b \mid \overline{d} \mid e),\ 5.\ (b \mid d \mid \overline{f}),\ 6.\ (\overline{b} \mid \overline{f}),\ 7.\ (c \mid \overline{e}),\ 8.\ (c \mid e),\ 9.\ (\overline{c} \mid \overline{d})\ \}$, assuming that *pickLiteral* picks a literal in the following order $\overline{a}, \overline{b}, \overline{c}, \overline{d}, \overline{e}, \overline{f}$ (false is tried first for each variable). Please (a) draw the decision tree of DPLL; (b) provide the unit clauses derived by BCP at each node of the decision tree; and (c) identify the head/tail literals of each clause at the end of *DPLL*(assuming initially the first and last literals are the head/tail literals).

8. We apply *GSAT* to the set S of clauses in the previous problem. Assume that the initial node (i.e., interpretation) is $\sigma_0 = \{\overline{a}, b, \overline{c}, d, \overline{e}, f\}$, and the strategy for picking a neighbor to replace the current node is (a) the best among all the neighbors; (b) the first neighbor which is better than the current node (flipping variables in the order of a, b, ..., f). Sideways moves are allowed if no neighbors are better than the current node. If no neighbors are better than the current node, a local optimum is found and you need to start with the node $\sigma_0 = \{a, \overline{b}, c, \overline{d}, e, \overline{f}\}$.

9. We apply DPLL with CDCL to the clause set $C : \{c_1 : (x_1 \mid x_2 \mid x_5),\ c_2 : (x_1 \mid x_3 \mid \overline{x_4} \mid x_5),\ c_3 : (\overline{x_1} \mid x_2 \mid x_3 \mid x_4),\ c_4 : (\overline{x_3} \mid x_4 \mid x_5),\ c_5 : (\overline{x_2} \mid x_3 \mid x_5),\ c_6 :$

$(\overline{x_3} \mid \overline{x_4} \mid \overline{x_5})$, $c_7 : (\overline{x_3} \mid x_4 \mid \overline{x_5})\}$. If *pickLiteral* picks a literal for case-split in the following order $\overline{x_5}, \overline{x_4}, ..., \overline{x_1}$ ($x_i = 0$ is tried first), please answer the following questions:

(a) For each conflicting clause found in $DPLL(C)$, what new clause will be generated by *conflictAnalysis*?

(b) What will be the head and tail literals if the new clause generated in (a) is not unit? And what will be the value of *level* at the end of the call to *insertNewClause*?

(c) Draw the DPLL decision tree until either a model is found or $DPLL(C)$ returns *false*.

10. The pigeon hole problem is to place $n+1$ pigeons into n holes such that every pigeon must be in a hole and no holes can hold more than one pigeon. Please write an encoder in any programming language that reads in an integer n and generates the CNF formula for the pigeon hole problem with $n+1$ pigeons and n holes in the DIMACS Format. Store the output of your encoder in a file named `pigeonN.cnf` and feed `pigeonN.cnf` to a SAT solver which accepts CNF in the DIMACS format. Turn in both the encoder, the result of the SAT solver for $n = 4, 6, 8, 10$, and the CNF file for $n = 4$ (i.e., `pigeon4.cnf`), plus a summary of the outputs of SAT solvers on the sizes of the input, the computing times, numbers of conflicts. What relation between the size of the input and the computing time?

11. The n-queen problem is to place n queens on an $n \times n$ board such that no two queens are in the same row, the same column, or the same diagonal. Please write a smaller program called *encoder* in any programming language that reads in an integer n and generates the CNF formula for the n-queen problem in the DIMACS Format. Store the output of your encoder in a file named `queenN.cnf` and feed `queenN.cnf` to a SAT solver which accepts CNF in the DIMACS format. Write another smaller program called *decoder* which reads the output of the SAT solver and displays a solution of n-queen on your computer screen. Turn in both the encoder and decoder, the result of the SAT solver for $n = 5, 10, 15$, and the CNF file for $n = 5$ (i.e., `queen5.cnf`), and the output of your decoder for $n = 10$.

12. You are asked to write two small programs, one called *encoder* and the other called *decoder*. The encoder will read a sudoku puzzle from a file (some examples are provided online) and generates the CNF formula for this puzzle

in the DIMACS Format. Store the output of your encoder in a file named `sudokuN.cnf` and feed `sudokuN.cnf` to a SAT solver which accepts CNF in the DIMACS format. The decoder will read the output of the SAT solver and displays a solution of the Sudoku puzzle on your computer screen. Turn in both the encoder and decoder, the result of the SAT solver, and the result of your decoder for the provided Sudoku puzzles.

13. A Latin square over $N = \{0, 1, ..., n-1\}$ is an $n \times n$ square where each row and each column of the square is a permutation of the numbers from N. Every Latin square defines a binary function $* : N, N \longrightarrow N$ such that $(x * y) = z$ iff the entry at the x^{th} row and the y^{th} column is z. You are asked to write a small program called *encoder*. The encoder will read an integer n and generates the CNF formula for the Latin square of size n satisfying the constraints $x * x = x$ and $(x*y)*(y*x) = y$ (called *Stein's third law*) in the DIMACS Format. Store the output of your encoder in a file named `latinN.cnf` and feed `latinN.cnf` to a SAT solver which accepts CNF in the DIMACS format. Turn in the encoder, and a summary of the results of the SAT solver for $n = 4, 5, ..., 9$.

14. Given an undirected simple graph $G = (V, E)$, an *independent set* of G is a subset $X \subseteq V$ such that for any two vertices of X, $(x, y) \notin E$. The *independent set problem* is to find a maximum independent set of G. Specify the independent set problem in hybrid MaxSAT; and (b) specify the decision version of the independent set problem in SAT.

15. Given an undirected simple graph $G = (V, E)$, a *vertex cover* of G is a subset $X \subseteq V$ such that for every edge $(x, y) \in E$, $\{x, y\} \cap X \neq \emptyset$. The *vertex cover problem* is to find a minimal vertex cover of G. (a) Specify the vertex cover problem in hybrid MaxSAT; and (b) specify the decision version of the vertex cover problem in SAT.

16. Given an undirected simple graph $G = (V, E)$, the *longest path problem* is to find the longest simple path (no repeated vertices) in G. (a) Specify the longest path problem in hybrid MaxSAT; and (b) specify the decision version of the longest path problem in SAT.

17. Given a set S of binary clauses,

$$S = \{(\overline{a} \mid \overline{b}), (\overline{a} \mid \overline{c}), (a \mid \overline{d}), (b \mid \overline{c}), (b \mid d), (c \mid \overline{d})\}$$

(a) draw the implication graph $G = (V, E)$ of S; (b) find the strongly connected components of G; (c) From G, decide if S is satisfiable or not. If satisfiable, construct a model of S from G.

CHAPTER 5
FIRST ORDER LOGIC

Propositional logic provides a good start at describing the general principles of logical reasoning, but it is not possible to express general properties of many important sets, especially when a set is infinite. For example, how to specify that every natural number is either even or odd, a natural number is even iff its successor is odd, or the relation $<$ is transitive over the natural numbers?

The weak expressive power of propositional logic accounts for its relative mathematical simplicity, but it is a very severe limitation, and it is desirable to have more expressive logics. Mathematics and some other disciplines such as computer science often consider sets of elements in which certain relations and operations are singled out. When using the language of propositional logic, our ability to talk about the properties of such relations and operations is very limited. Thus, it is necessary to refine the logic language, in order to increase the expressive power of logic. This is exactly what is provided by a more expressive logical framework known as first-order logic, which will be the topic of the next three chapters. First-order logic is a considerably richer logic than propositional logic, yet enjoys many nice mathematical properties.

In first-order logic, many interesting and important properties about various sets can be expressed. Technically, this is achieved by allowing the propositional symbols to have arguments ranging over elements of sets, to express the relations and operations in question. These propositional symbols are called *predicate* symbols. This is the reason that first-order logic is often called *predicate calculus*.

5.1 Syntax of First Order Languages

A logic is a collection of closely related formal languages. There are certain languages called first-order languages, and together they form first-order logic. Our study of first-order logic will parallel the study of propositional logic: We will at first define the syntax and semantics of first-order languages as an extension of propositional languages, and then present various reasoning tools for these languages.

5.1.1 Terms and Formulas

Consider some general statements about the natural numbers:

- Every natural number is even or odd, but not both.
- A natural number x is even iff $x + 1$ is odd.
- For any natural number x, $x < x + 1$ and $\neg(x < x)$.
- For any three natural numbers x, y, and z, if $x < y$ and $y < z$, then $x < z$.
- For every natural number x, there exists a natural number y, such that $x < y$.

Before any attempt to prove these statements, let us consider how to write them in a formal language. At first, we need a way to describe the set of natural numbers. Like propositional logic, first-order logic does not assume the existence of any special sets, except some symbols, and everything needs to be constructed from scratch. Fortunately, from Example 1.3.11, we know that the set of natural numbers can be constructed recursively from two symbols: the constant 0 and the unary function symbol s: Natural number i is uniquely represented by the term $s^i(0)$, which means there are i copies of s in $s(s(...s(0)...))$. Hence, our first-order language will have 0 and s.

Next, to express "even" or "odd" properties, we may introduce the predicate symbols *even* and *odd*, which take a natural number and return a Boolean value. The relation $<$ is used in the statement and we take it in the first-order language. Since some variables like x, y, and z are already used in the statements, we borrow them, too.

Finally, to express "for every" and "there exists" in natural languages, we add two important symbols, $\forall$ and $\exists$, respectively, in a first-order language. The symbol $\forall$ is called *universal quantifier* and $\exists$ is called *existential quantifier*. These two symbols will be discussed extensively in the next subsection.

Definition 5.1.1. *A first-order language L is built up by an* alphabet *that consists of four sets of symbols:*

- *A set P of* predicate *symbols, including propositional variables, usually denoted by p, q, r, with or without subscripts, or some popular binary relation symbols, such as $<$, $\leq$, $\geq$, $>$, etc.*
- *A set F of* function *symbols, including constants, usually denoted by a, b, c (constants), f, g, h, with or without subscripts.*
- *A set X of* variables*, usually denoted by x, y, and z, with or without subscripts.*
- *A set Op of logical operators such as $\top, \bot, \neg, \wedge, \vee, \rightarrow$, and $\leftrightarrow$, plus $\forall$ and $\exists$.*

For each symbol in $P \cup F$, the number of arguments for that symbol is fixed and is dictated by the arity *function $arity : P \cup F \longrightarrow \mathcal{N}$. If $arity(p) = 0$ for $p \in P$, p is the same as a propositional variable; if $arity(f) = 0$ for $f \in F$, f is a constant.*

The triple $(P, F, arity)$ is called the signature *of the language. We denote the language L by $L = (P, F, X, Op)$.*

In general, P or F could be empty; X and Op can never be empty. In theory, P or F or X could be infinite, but in practice, they are always finite. We also assume that no symbols appear more than once in these four sets. Rather than fixing a single language once and for all, different signatures generate different first-order languages and allow us to specify the symbols we wish to use for any given domain of interest.

Example 5.1.2. To specify the properties of the natural numbers listed at the beginning of this section, we use the signature $(P, F, arity)$, where $P = \{even/1, odd/1, < /2\}$, $F = \{0/0, s/1\}$, the arity is given after the symbol / following each predicate or function symbol. □

Once a signature is given, we add some variables X and logical operators Op, to obtain the alphabet for a first-order language. First-order logic allows us to build complex expressions out of the alphabet. Starting with the variables and constants, we can use the function symbols to build up compound expressions which are called *terms*.

Definition 5.1.3. *Given a first-order language $L = (P, F, X, Op)$, the set of* terms *of L are the set of strings built up by F and X, which can be formally defined by the following BNF grammar:*

$$\begin{array}{rcl} \langle Constants \rangle & ::= & a \textbf{ if } a \in F, arity(a) = 0 \\ \langle Variables \rangle & ::= & x \textbf{ if } x \in X \\ \langle Terms \rangle & ::= & \langle Constants \rangle \mid \langle Variables \rangle \mid \\ & & f(\langle Terms \rangle_1, \langle Terms \rangle_2, ..., \langle Terms \rangle_k) \textbf{ if } f \in F, arity(a) = k \end{array}$$

The set of terms in $\langle Terms \rangle$ is often denoted by $T(F, X)$. A term is said to be ground *if it does not contain any variable. The set of all ground terms is denoted by $T(F)$. Note that $T(F) = \emptyset$ if F contains no constants.*

For $F = \{0, s\}$ in the previous example, $T(F) = \{0, s(0), s(s(0)), ...\}$ and $T(F, X) = \{0, s(0), s(x), s(y), s(s(0)), s(s(x)), s(s(y)), ...\}$ for $X = \{x, y\}$.

Intuitively, the terms name objects in the intended domain of interest. In a term, besides the symbols from F and X, the parentheses "(" and ")" and the comma "," are also used to identify its structure. Like every propositional formula

has a formula tree (Definition 2.1.1), for each term we can have a term tree in which parentheses and commas are omitted. We may assign a *position* to each node of the term tree and use this position to identify the subterm denoted by that node.

Definition 5.1.4. *Given a term t, a* position *p of t, with the* subterm *at p, denoted by t/a, is recursively defined as follows:*

- *If t is a constant or a variable, then $p = \epsilon$, the empty sequence, and $t/p = t$.*
- *If $t = f(t_1, ..., t_k)$, then either $p = \epsilon$ and $t/p = t$, or $p = i.q$, where $1 \leq i \leq k$, and $t/p = t_i/q$.*

For example, if $t = f(x, g(g(y)))$, then the legal positions of t are $\epsilon, 1, 2, 2.1$, and 2.1.1, and the corresponding subterms are t, x, $g(g(y))$, $g(y)$, and y, respectively.

Now adding the usual logical operators from propositional logic, we have pretty much all the symbols needed for a first-order language.

Definition 5.1.5. *Given a first-order language $L = (P, F, X, Op)$, the* formulas *of L are the set of strings built up by terms, predicate symbols and logical operators. Let op be the binary operators used in the current application, then the* formulas *for this application can be defined by the following BNF grammar:*

$$
\begin{array}{rcl}
\langle op \rangle & ::= & \wedge \mid \vee \mid \rightarrow \mid \oplus \mid \leftrightarrow \\
\langle Atoms \rangle & ::= & p(\langle Terms \rangle_1, \langle Terms \rangle_2, ..., \langle Terms \rangle_k) \textbf{ if } p \in P, arity(a) = k \\
\langle Literals \rangle & ::= & \langle Atoms \rangle \mid \neg \langle Atoms \rangle \\
\langle Formulas \rangle & ::= & \top \mid \bot \mid \langle Atoms \rangle \mid \neg \langle Formulas \rangle \mid \\
 & & (\langle Formulas \rangle \ \langle op \rangle \ \langle Formulas \rangle) \mid \\
 & & (\forall \langle Variables \rangle \langle Formulas \rangle) \mid (\exists \langle Variables \rangle \langle Formulas \rangle)
\end{array}
$$

A formula (clause, literal, atom) is said to be ground *if it does not contain any variable.*

Atoms are a shorthand for *atomic formulas*. We can have a formula tree for each first-order formula.

Some parentheses are unnecessary if we use a precedence relation among the logical operators. When displaying a formula, it is common to drop the outmost parentheses or those following a predicate symbol of zero arity, e.g. we write p instead of $p()$. We may also define subformulas of a formula as we did for propositional formulas and terms.

Example 5.1.6. Given a signature $(P, F, arity)$, where $P = \{child/2, love/2\}$ and $F = \{a_i/0 : 1 \leq i \leq 100\}$, then $T(F) = F$ and $T(F, X) = F \cup X$, because F does not have any non-constant function symbols. If $X = \{x, y, z\}$, the following are some examples of formulas:

- **ground atoms**: $child(a_2, a_1)$, $child(a_3, a_1)$, $love(a_1, a_2)$, ...
- **non-ground atoms**: $child(x, y)$, $loves(y, x)$, $love(z, x)$, ...
- **ground non-atom formulas**: $\neg child(a_3, a_4)$, $child(a_2, a_1) \vee child(a_3, a_1)$, $child(a_4, a_1) \to love(a_1, a_4)$, ...
- **general formulas**: $\neg child(a_1, y)$, $\neg child(x, y) \vee love(y, x)$, $love(x, y) \vee \neg love(y, x)$, ...

□

In these examples, if we replace each atom by a propositional variable, they become propositional formulas. This is true for all ground formulas. Examples of formulas with quantifiers are given in the next subsection.

5.1.2 The Quantifiers

What makes first-order logic powerful is that it allows us to make general assertions using quantifiers: Universal quantifier $\forall$ and existential quantifier $\exists$. In the definition of $\langle Formulas \rangle$, every occurrence of quantifiers is followed by a variable, say x, and then followed by a formula, say A. That is, the quantifiers appear in a formula like either $\forall x\, A$ or $\exists x\, A$; the former says A is held for every value of x and the latter says A is held for some value of x.

In middle school algebra, when we say $+$ is commutative, we use formula $x + y = y + x$, here x and y are meant for all values. When solving an equation like $x^2 - 3x + 2 = 0$, we try to find some value of x so that the equation holds. In first-order logic, the two cases are expressed as $\forall x \forall y\, (x + y = y + x)$ and $\exists x\, (x^2 - 3x + 2 = 0)$, respectively.

Example 5.1.7. Continue from Example 5.1.6, if $child(x, y)$ means x is a child of y, $parent(x, y)$ means x is a parent of y, $descen(x, y)$ means x is a descendant of y, and $love(x, y)$ means x loves y, then we can write many assertions in first-order logic. □

- x is a child of y iff y is a parent of x: $\forall x \forall y\, child(x, y) \leftrightarrow parent(y, x)$.
- If x is a child of y, then x is a descendant of y: $\forall x \forall y\, child(x, y) \to descen(x, y)$.
- Every parent loves his child: $\forall x \forall y\, (parent(x, y) \to love(x, y))$.
- Everybody is loved by somebody: $\forall x \exists y\, love(y, x)$.

- The descendant relation is transitive: $\forall x \forall y \forall z\, (descen(x, y) \wedge descen(y, z) \rightarrow descen(x, z))$.

When using $\forall$ to describe a statement under various conditions, these conditions usually should be the first argument of $\rightarrow$. When using $\exists$ to describe a statement under various conditions, we cannot use *imply*. For example, let $own(x, y)$ means "x owns y", $red(x)$ means "x is red", $toy(x)$ means "x is a toy", $car(x)$ means "x is a car". Then the statement "Tom loves every red toy he owns" can be expressed as

$$\forall x\, (own(Tom, x) \wedge red(x) \wedge toy(x) \rightarrow love(Tom, x)).$$

It is natural to read it as a restricted quantifier: "for everything x, if Tom owns x, x is red and x is a toy, then Tom loves x." Of course, if we get rid of $\rightarrow$ in the above formula, we have

$$\forall x\, \neg own(Tom, x) \vee \neg red(x) \vee \neg toy(x) \vee love(Tom, x)),$$

which can be read as ""for everything x, either Tom doesn't own x, x is not red, x is not a toy, or Tom loves x." Note that if we replace $\rightarrow$ by $\wedge$ in the above formula, then it claims that "Tom owns everything", "everything is red", "everything is a toy", and "Tom loves everything".

On the other hand, to express "Tom loves one of his red toy cars", the formula is

$$\exists x\, (own(Tom, x) \wedge toy(x) \wedge red(x) \wedge car(x) \wedge love(Tom, x)).$$

If we replace the last $\wedge$ by $\rightarrow$ in the above formula, then it claims that there exists something, denoted by "it", and one (or more) of the following is true: "Tom doesn't own it", "it is not a toy", "it is not red", "it is not a car", or "Tom loves it". Since there are many things that "Tom doesn't own it", "it is not a toy", "it is not red", or "it is not a car", the above formula does not catch the meaning that "Tom loves one of his red toy cars."

Definition 5.1.8. *In $\forall x\, A$ or $\exists x\, A$, the* scope *of the variable x is A and every occurrence of x in A is said be* bounded. *A variable in A is said to be* free *if it is not bounded. A formula A is said to be* closed *if it has no free variables. A* sentence *is a closed formula.*

In $love(x, y)$, both x and y are free; in $\exists y\, p(x, y)$, x is free, y is bounded (we cannot say "y in $p(x, y)$ is bounded" as "y is free in $p(x, y)$"); in $\forall x \exists y\, p(x, y)$, neither x nor y is free; in $(\forall x\, q(x)) \vee r(x)$, the first occurrence of x is bounded and the second occurrence of x is free.

When A represents any formula, we write $A(x_1, x_2, ..., x_n)$ to indicate that the free variables of A are among $x_1, x_2, ..., x_n$.

Like propositional formulas, we may use the precedence relation on the logical operators to avoid writing out all the parentheses in a formula. The precedence relation used in Chapter 2 is

$$\neg,\ \wedge,\ \vee,\ \ \rightarrow,\ \{\oplus, \leftrightarrow\}.$$

Where to place $\forall$ and $\exists$? According to *Wikipedia* on first-order logic, the quantifiers can be placed right before $\rightarrow$. However, as a common practice, we will place them at the end, after $\leftrightarrow$. That is,

$$\neg,\ \wedge,\ \vee,\ \ \rightarrow,\ \{\oplus, \leftrightarrow\},\ \{\forall, \exists\}.$$

For example, $\forall x\, A \vee B$ is interpreted as $\forall x\, (A \vee B)$, and we would write $(\forall x\, A) \vee$ to limit the scope of x on A only.

Now, using the predicates and relation symbols, we can create formulas about the statements mentioned in the beginning of this section:

- $\forall x\, odd(x) \leftrightarrow \neg even(x)$: Every natural number is even or odd, but not both.
- $\forall x\, even(s(x)) \leftrightarrow odd(x)$: A natural number x is odd iff $x + 1$ is even.
- $\forall x\, x < s(x) \wedge \neg(x < x)$: For any natural number x, $x < x + 1$ and $\neg(x < x)$.
- $\forall x \exists y\, x < y$: For every natural number x, there exists a natural number y, such that $x < y$.
- $\forall x \forall y \forall z\, (x < y) \wedge (y < z) \rightarrow (x < z)$: For any three natural numbers x, y, and z, if $x < y$ and $y < z$, then $x < z$, that is, $<$ is transitive.

5.1.3 Unsorted and Many-Sorted Logics

First-order languages are generally regarded as unsorted languages. If we look at its syntax carefully, two sets of objects are defined: $\langle Terms \rangle$ and $\langle Formulas \rangle$. Obviously, $\langle Formulas \rangle$ are of sort boolean, denoted by $Bool = \{0, 1\}$. $\langle Terms \rangle$ represent objects of interests, and the set of objects is commonly called the *universe* (also called the *domain* of discourse) of the language. For example, the universe could use $T(F)$, the set of ground terms, to represent the universe.

If we use U to denote the sort of $\langle Terms \rangle$, for any signature $(P, F, arity)$, predicate symbol $p \in P$ represents a function $p : U^{arity(p)} \longrightarrow Bool$ and function

symbol $f \in F$ represents a function $f : U^{arity(f)} \longrightarrow U$. The universal and existential quantifiers range over that universe U. For example, the first-order language in Example 5.1.6 talks about 100 people living in a certain town, with a relation $love(x, y)$ to express that x loves y. In such a language, we might express the statement that "everyone loves someone" by writing $\forall x \exists y\, love(x, y)$.

For many applications, $Bool$ and U are disjoint, like in Examples 5.1.2 and 5.1.6. For some applications, $Bool$ can be regarded as a subset of U. For example, to model the if-then-else command in a programming language, we may use the symbol $ite : Bool \times U \times U \longrightarrow U$. Another example is the function symbol $says : U \times Bool \longrightarrow Bool$ in the Knight and Knave puzzles (Example 2.5.4).

It is not difficult to introduce sorts in first-order logic and this is done by using unary predicate symbols ($arity = 1$). In Example 5.1.6, we use a_i to stand for person i. If we need function $age(x)$ to tell the age of person x, we need the function symbols for natural numbers. If we use those in Example 5.1.2, then $F = \{0, s, a_i \mid 1 \leq i \leq 100\}$. In this application, we do need to tell which are persons and which are numbers by using the following predicate symbols:

- $person : U \longrightarrow Bool$: $person(x)$ is true iff x is a person.
- $nat : U \longrightarrow Bool$: $nat(x)$ is true iff x is a natural number.

$person(x)$ can be easily defined by asserting $person(a_i)$ for $1 \leq i \leq 100$. $nat(x)$ can also be defined by asserting $nat(0)$ and $nat(s(x)) \equiv nat(x)$.

Now, to state general properties as in Examples 5.1.2 and 5.1.6, we have to put restrictions on the variables. For example, to express "everybody is loved by somebody", the formula now is

$$\forall x \exists y\, person(x) \wedge person(y) \rightarrow love(y, x).$$

To say "$<$" is transitive on natural numbers, we use

$$\forall x, y, z\, nat(x) \wedge nat(y) \wedge nat(z) \wedge (x < y) \wedge (y < z) \rightarrow (x < z).$$

Dealing with such unary predicates is tedious. To get rid of such clumsy, people introduce "*sorts*". Let

$$Person = \{x \mid person(x) \wedge x \in U\}$$
$$Nat = \{x \mid nat(x) \wedge x \in U\}$$

Then define $love : Person^2 \longrightarrow Bool$ and $<: Nat^2 \longrightarrow Bool$ and the formulas become neat again: We do not need $person(x)$, $nat(x)$, etc., in the formula. For

the *age* function, we have $age : Person \longrightarrow Nat$. That is the birth of many-sorted logic, which is a mild extension of first-order logic.

The idea of many-sorts is based on set theory, where each sort is a set of similar objects. Sets are usually defined by unary predicates. From a logician's viewpoint, a set $S = \{x \mid p(x)\}$ and a predicate $p(x)$ have the same meaning. In many-sorted logic, one can have different sorts of objects — such as *persons* and *natural numbers* — and a separate set of variables ranging over each. Moreover, the specification of function symbols and predicate symbols indicates what sorts of arguments they expect (the *domain*), and, in the case of function symbols, what sort of value they return (the *range*). The concept of *signature* is extended to contain the domain and range information of each function and predicate symbol.

Many-sorted logic can reflect formally our intention not to handle the universe as a collection of objects, but to partition it in a way that is similar to types in programming languages. This partition can be carried out on the syntax level: substitution of variables can be done only accordingly, respecting the "*sorts*".

5.2 Semantics

We should be aware that, at this stage, all the function and predicate symbols are just symbols. In propositional logic, a symbol p can represent "Tom runs fast", or "Mary is tall". The same holds true in first-order logic as we may give different meaning to a symbol.

In Example 5.1.7, we have seen the following two formulas:

- $\forall x \forall y \, (child(x, y) \rightarrow descen(x, y))$.
- $\forall x \forall y \, (parent(x, y) \rightarrow love(x, y))$.

The first formula says "if x is a child of y, then x is a descendant of y"; the second says "every parent loves his child." What is the difference between them? They differ only on predicate symbols. If we give different meanings to the predicate symbols, one can replace the other. That is, we have designed the language with a certain interpretation in mind, but one could also interpret the same formula differently.

In this section, we will spell out the concepts of interpretations, models, satisfiable or valid formulas, as an extension of these concepts of propositional logic.

5.2.1 Interpretation

For propositional logic, an interpretation is an assignment of propositional variables to truth values. For a first-order language $L = (P, F, X, Op)$, we need an

interpretation for every symbol of L.

Definition 5.2.1. *Given a closed formula A of $L = (P, F, X, Op)$, an interpretation of A is a triple $I = (D, R, G)$, where*

- *D is a non-empty set called the* universe *of I, often denoted by D^I.*
- *For each predicate symbol $p \in P$ of arity k, there is a k-ary relation $r_p \in R$ over D, $r_p \subseteq D^k$, denoted by $r_p = p^I$. If $k = 0$, then p is a proposiitonal variable and p^I is either 1 or 0, a truth assignment to p.*
- *For each function symbol $f \in F$ of arity k, there is a function $g : D^k \longrightarrow D \in G$, denoted by $g = f^I$. If $k = 0$, then $f^I \in D$.*

Example 5.2.2. Consider $L = (P, F, X, Op)$, where $P = \{p/2\}$, $F = \{a/0\}$, and $X = \{x\}$. Then for the formula $A = \forall x\, p(a, x)$, we may have the following interpretations: □

1. $I_1 = (\mathcal{N}, \{\leq\}, \{0\})$ and the meaning of A is "for every natural number x, $0 \leq x$."
2. $I_2 = (\mathcal{N}, \{|\}, \{1\})$, $x|y$ returns true iff x divides y, and the meaning of A is "for every natural number x, $1 \mid x$."
3. $I_3 = (\{0,1\}^3, \{\mathit{prefix}\}, \{\epsilon\})$, $\mathit{prefix}(x, y)$ returns true iff x is a prefix of y, the meaning of A is "for every binary string x of length 3, the empty string ϵ is a prefix of x."
4. $I_4 = (\mathcal{P}(\mathcal{N}), \{\subseteq\}, \{\emptyset\})$, $\mathcal{P}(\mathcal{N})$ is the power set of $\mathcal{N}$, and the meaning of A is "for every subset x of natural numbers, $\emptyset \subseteq x$."
5. $I_5 = (V, \{E\}, \{a\})$, where $V = \{a, b, c\}$, $E = \{(a, a), (a, b), (a, c), (c, c)\}$, $G = (V, E)$ is a graph, and the meaning of A is "for every vertex $x \in V$, there is an edge $(a, x) \in E$."

In propositional logic, once we have an interpretation, we may use a procedure like *eval* (Algorithm 2.2.4) to obtain the truth value of a formula. We can do the same in first-order logic, provided that we have a way of handling free variables and quantifiers in the formula.

Let $free(A)$ be the free variables of formula A. Given a universe D, an *assignment* is a mapping $\theta : free(A) \longrightarrow D$, which maps each free variable of A to an element of D. The purpose of θ is that, during the evaluation of a formula under an interpretation, if we see a free variable x, we will use the value $\theta(x)$ for x.

Procedure 5.2.3. The procedure *eval* takes a formula or a term A, with or without free variables, in $L = (P, F, X, Op)$, an interpretation I for L, and an assignment $\theta : free(A) \longrightarrow D^I$, and returns a Boolean value for a formula and an element of D^I for a term.

proc $eval(A, I, \theta)$
  if $A = \top$ **return** 1; // 1 means true
  if $A = \bot$ **return** 0; // 0 means false
  if $A \in X$ **return** $\theta(A)$; // A is a free variable
  if $A = f(t_1, t_2, ..., t_k)$ **return** $f^I(t'_1, t'_2, ..., t'_k)$, where $t'_i = eval(t_i, I, \theta)$;
  if $A = \neg B$ **return** $\neg eval(B, I, \theta)$;
  if $A = (B\ op\ C)$ **return** $eval(B, I, \theta)\ op\ eval(C, I, \theta)$, where $op \in Op$;
  if $A = (\forall x\, B)$ **return** $allInD(B, I, \theta, x, D^I)$;
  if $A = (\exists x\, B)$ **return** $someInD(B, I, \theta, x, D^I)$;
  else return "unknown";

Procedure $allInD(B, I, \theta, x, S)$ takes formula B, interpretation I, assignment θ, free variable x in B, and $S \subseteq D$, evaluates the value of B under I and $\theta \cup \{x \leftharpoonup d\}$ for every value $d \in S$. If $eval(B, I, \theta \cup \{x \leftharpoonup d\})$ returns 0 for one $d \in D^I$, return 0; otherwise, return 1.

proc $allInD(B, I, \theta, x, S)$
  if $S = \emptyset$ **return** 1;
  pick $d \in S$;
  if $(eval(B, I, \theta \cup \{x \leftharpoonup d\}) = 0)$ **return** 0;
  return $allInD(B, I, \theta, x, S - \{d\})$;

Procedure $someInD(B, I, \theta, x, S)$ works in the same way as $allInD$, except that if one of the evaluations, $eval(B, I, \theta \cup \{x \leftharpoonup d\})$, return 1, return 1; otherwise, return 0.

proc $someInD(B, I, \theta, x, S)$
  if $S = \emptyset$ **return** 0;
  pick $d \in S$;
  if $(eval(B, I, \theta \cup \{x \leftharpoonup d\}) = 1)$ **return** 1;
  return $someInD(B, I, \theta, x, S - \{d\})$;

Proposition 5.2.4. *Given a formula A of $L = (P, F, X, Op)$, an interpretation I, and an assignment $\theta : free(A) \longrightarrow D^I$,* $\text{eval}(A, I, \theta)$ *will return a boolean value; if t is a term appearing in A, then* $\text{eval}(t, I, \theta)$ *will return a value of D^I.*

Proof. (sketch) Let D be D_I. We do induction on the tree structure of A. The base cases when A or t are symbols of arity 0 are trivial. If $t = f(t_1, t_2, ..., t_k)$ is a term in A, the returned value is $f^I(t'_1, t'_2, ..., t'_k) \in D^I$ because $f^I : D^k \longrightarrow D$ is a function over D. If A is an atom $p(t_1, t_2, ..., t_k)$, where $p \in P$, then p^I is a relation and $p^I(t'_1, t'_2, ..., t'_k) = 1$ iff $(t'_1, t'_2, ..., t'_k) \in p^I$ by our assumption. That is, $p^I(t'_1, t'_2, ..., t'_k)$ is boolean. For non-atom formulas, the returned values are always boolean as shown by the next proposition. □

From now on, we will denote $eval(A, I, \theta)$ by $I(A, \theta)$ and $eval(A, I, \emptyset)$ by $I(A)$ for brevity.

Proposition 5.2.5. *For formulas A and B, interpretation I, and assignment θ,*

- $I(\neg A, \theta) = \neg I(A, \theta)$;
- $I(A \vee B, \theta) = I(A, \theta) \vee I(B, \theta)$;
- $I(A \wedge B, \theta) = I(A, \theta) \wedge I(B, \theta)$;
- $I(A \oplus B, \theta) = I(A, \theta) \oplus I(B, \theta)$;
- $I(A \leftrightarrow B, \theta) = I(A, \theta) \leftrightarrow I(B, \theta)$;
- $I(\forall x\, B(x), \theta) = allInD(B(x), I, \theta, x, D^I)$;
- $I(\exists x\, B(x), \theta) = someInD(B(x), I, \theta, x, D^I)$.

From the definition of *eval*, it is easy to check that the above proposition holds. Obviously, *eval* cannot be an algorithm because $allInD$ will not terminate when D^I is infinite. We present *eval* as a procedure because we assume that the reader is familiar with algorithms. We need other tools such as induction to evaluate a formula when the universe of the interpretation is infinite. When D^I is finite, the termination of *eval* is easy to establish.

Example 5.2.6. For I_5 in Example 5.2.2, $I_5(\forall x\, p(a, x))$ will call $allInD$ with parameters $(p(a, x), I_5, \emptyset, x, \{a, b, c\})$, which will call *eval* three times, with $\theta = \{x \leftharpoonup a\}, \{x \leftharpoonup b\}$, and $\{x \leftharpoonup c\}$, respectively. Since E contains (a, x) for $x \in \{a, b, c\}$, all the evaluations return 1. The returned result can be expressed as

$$I_5(\forall x\, p(a, x)) = \bigwedge_{d \in \{a,b,c\}} I_5(p(a, x), \{x \leftharpoonup d\})$$

□

We can do the same for all the interpretations in Example 5.2.2. In general,

$$I(\forall x\, B(x), \theta) = \bigwedge_{d \in D^I} I(B(x), \theta \cup \{x \leftharpoonup d\})$$

which catches the meaning that "$I(\forall x\, B(x), \theta)$ returns 1 iff $B(x)$ is evaluated to be 1 under I and $\theta \cup \{x \leftharpoonup d\}$ for every value $d \in D$".

Similarly, we have the following notation for $I(\exists x\, B(x), \theta)$, which is equal to $someInD(B(x), I, \theta, x, D)$:

$$I(\exists x\, B(x), \theta) = \bigvee_{d \in D^I} I(B(x), \theta \cup \{x \leftharpoonup d\})$$

which catches the meaning that "$I(\exists x\, B(x), \theta)$ returns 1 iff $B(x)$ is evaluated to be true under I and $\theta \cup \{x \leftharpoonup d\}$ for some value $d \in D^I$". These notations will help us to understand quantified formulas and establish their properties. Note that if the domain D^I is empty, then $I(\forall x\, B(x), \theta) = 1$ and $I(\exists x\, B(x), \theta) = 0$.

5.2.2 Models, Satisfiability, and Validity

Here are just straightforward extensions from propositional logic to first-order logic. Recall that a sentence is a closed formula.

Definition 5.2.7. *Given a sentence A and an interpretation I, I is said to be a* model *of A if $I(A) = 1$. If A has a model, A is said to be* satisfiable.

For a propositional formula, the number of models is finite; for a first-order formula, the number of models is infinite in general, because we have an infinite number of choices for domains, relations, and functions in an interpretation. However, we can still borrow the notation $\mathcal{M}(A)$ from propositional logic.

Definition 5.2.8. *Given a closed formula A, let $\mathcal{M}(A)$ be the set of all models of A. If $\mathcal{M}(A) = \emptyset$, A is said to be* unsatisfiable*; if $\mathcal{M}(A)$ contains every interpretation, i.e., every interpretation is a model of A, then A is said to be* valid.

With the identical definitions from propositional logic to first-order logic, the following result is expected.

Proposition 5.2.9. *Every valid propositional formula is a valid formula in first-order logic, and every unsatisfiable propositional formula is unsatisfiable in first-order logic.*

Of course, there are more valid formulas in first-order logic and our attention is on formulas with quantifiers.

Example 5.2.10. Let us check some examples: □

1. $A_1 = \forall x\,(p(a,x) \to p(a,a))$.
   Consider $I = (\{a,b\}, \{p\}, \{a\})$, where $p^I = p = \{\langle a,a\rangle\}$, $a^I = a$, $I(A_1) = 1$ because $p^I(a,a) = 1$ and

$$\begin{aligned} I(A_1) = & \bigwedge_{d\in\{a,b\}} I(p(a,x) \to p(a,a)), \{x \leftharpoonup d\}) \\ = & I(p(a,a) \to p(a,a)) \wedge I(p(a,b) \to p(a,a)) \\ = & 1 \wedge (p^I(a,b) \to p^I(a,a)) \\ = & 1 \end{aligned}$$

   Let $I' = (\{a,b\}, \{\{\langle a,b\rangle\}\}, \{a\})$, then $I'(A_1) = 0$ because $p^{I'}(a,a) = 0$ and $p^{I'}(a,b) = 1$ imply that $I'(p(a,b) \to p(a,a)) = 0$. $I'(A_1) = I'(p(a,a) \to p(a,a)) \wedge I'(p(a,b) \to p(a,a)) = 1 \wedge 0 = 0$. Thus A_1 is satisfiable but not valid.

2. $A_2 = (\forall x\, p(a,x)) \to p(a,a)$.
   A_2 is different from A_1 because the scopes of x are different in A_1 and A_2. For any interpretation I, we consider the truth value of $p^I(a^I, a^I)$. If $p^I(a^I, a^I) = 1$, then $I(A_2, \emptyset) = 1$. If $p^I(a^I, a^I) = 0$, then $I(\forall x\, p(a,x)) = 0$ because x will take on value a^I. So $I(A_2) = 1$ in any case. Since I is arbitrary, A_2 must be valid.

3. $A_3 = \exists x \exists y\,(p(x) \wedge \neg p(y))$.
   Consider $I = (\{a,b\}, \{p\}, \emptyset)$, where $p(a) = 1$ and $p(b) = 0$, then $p(a) \wedge \neg p(b) = 1$. From $I(p(x) \wedge \neg p(y), \{x \leftharpoonup a, y \leftharpoonup b\}) = 1$, we conclude that I is a model of A_3. A_3 is false in any interpretation whose domain is empty or contains a single element. Thus, A_3 is not valid.

Definition 5.2.11. *Let $A(x)$ be a formula where x is a free variable of A. Then $A(t)$ denotes the formula $A(x)[x \leftharpoonup t]$, where every occurrence of x is replaced by term t. The formula $A(t)$ is called an* instance *of $A(x)$ by substituting x for t* (ref. Definition 2.1.3).

We will use substitutions extensively in the next chapter. The substution $\theta : X \longrightarrow D$ used in the evalutation procedure *eval* coincides with the above definition if $D = T(F)$ (the set of ground terms built on F). That is, for any interpretation $I = (D, R, G)$ of $L = (P, F, X, Op)$, if $D = T(F)$, then $I(A(x), \{x \leftharpoonup d\}) = I(A(d))$, i.e., $\mathit{eval}(A(x), I, \{x \leftharpoonup d\}) = \mathit{eval}(A(d), I, \emptyset)$, where $x \in X, d \in T(F)$, and $A(d) = A(x)[x \leftharpoonup d]$.

Proposition 5.2.12. *For any formula $A(x)$, where x is a free variable of A and variable y does not appear in $A(x)$, $\forall x\, A(x) \equiv \forall y\, A(y)$ and $\exists x\, A(x) \equiv \exists y\, A(y)$.*

Proof. For any interpretation I,

$$I(\forall x\, A(x)) = \bigwedge_{d \in D} I(A(x), \{x \leftharpoonup d\})$$
$$I(\forall y\, A(y)) = \bigwedge_{d \in D} I(A(y), \{y \leftharpoonup d\})$$

The variable x serves as a placeholder in $A(x)$, just like y serves as the same placeholder in $A(y)$, to tell *eval* when to replace x or y by d. Any distinct name can be used for this placeholder, thus $I(\forall x\, A(x), \theta) = I(\forall y\, A(y), \theta)$. Since I is arbitrary, it must be the case that $\forall x\, A(x) \equiv \forall y\, A(y)$. The proof for the second part is identical. □

The above proposition allows us to change the names of bounded variables safely, so that different variables are used for each occurrence of quantifiers. For example, it is better to replace $\forall x\, p(x) \vee \exists x\, q(x, y)$ by $\forall x\, p(x) \vee \exists z\, q(z, y)$, to improve the readability.

We will keep the convention that a set S of formulas denotes the conjunction of the formulas appearing in the set. If every formula is true in the same interpretation, then the set S is said to be satisfiable and the interpretation is its model.

Example 5.2.13. Let $S = \{\forall x \exists y\, p(x, y), \forall x\, \neg p(x, x), \forall x \forall y \forall z\, p(x, y) \wedge p(y, z) \rightarrow p(x, z)\}$. The second formula states that "p is irreflexive"; the third formula says "p is transitive".

Consider the interpretation $I = (\{\mathcal{N}\}, \{<\}, \emptyset)$, it is easy to check that I is a model of S, because $<$ is transitive and irreflexive, i.e., $\neg(x < x)$, and for every $x \in \mathcal{N}$, there exists $x + 1 \in \mathcal{N}$ such that $x < x + 1$.

Does S has a finite model, i.e., a model in which the universe is finite? If S has a finite model, say $I = (D, \{r_p\}, \emptyset)$, where $|D| = n$, consider the directed graph $G = (D, r_p)$, then G cannot have any cycle, because, if there exists a cycle, then there exists a path from a vertex in the cycle to itself. If there exists a path from a to b in G, there must exist an edge from a to b, because r_p is transitive. We cannot have an edge from a vertex to itself because r_p is irreflexive. For a vertex x at the end of a path, we cannot find another vertex y satisfying $\forall x \exists y\, p(x, y)$. Hence, S cannot have any finite model. □

5.2.3 Entailment and Equivalence

The following two definitions are copied from Definitions 2.2.13 and 2.2.17.

Definition 5.2.14. *Given two formulas A and B, we say A* entails *B, or B is a* logical consequence *of A, denoted by $A \models B$, if $\mathcal{M}(A) \subseteq \mathcal{M}(B)$.*

To denote that A is valid, we can simply write $\models A$.

Definition 5.2.15. *Given two formulas A and B, A and B are said to be* logically equivalent *if $\mathcal{M}(A) = \mathcal{M}(B)$, denoted by $A \equiv B$.*

With the same definitions, we have the same results for first-order logic.

Theorem 5.2.16. *For any two formulas A and B, (a) $A \models B$ iff $A \to B$ is valid; and (b) $A \equiv B$ iff $A \leftrightarrow B$ is valid.*

The substitution theorem from propositional logic still holds in first-order logic.

Theorem 5.2.17. *For any formulas A, B, and C, where B is a subformula of A, and $B \equiv C$, then $A \equiv A[B \leftharpoonup C]$.*

Proposition 5.2.18. *Let $A(x)$, $B(x)$, and $C(x, y)$ be the formulas with free variables x and y.*

1. $\neg\forall x\, A(x) \equiv \exists x\, \neg A(x)$
2. $\neg\exists x\, A(x) \equiv \forall x\, \neg A(x)$
3. $\forall x\, A(x) \wedge B(x) \equiv (\forall x\, A(x)) \wedge (\forall x\, B(x))$
4. $\exists x\, A(x) \vee B(x) \equiv (\exists x\, A(x)) \vee (\exists x\, B(x))$
5. $(\forall x\, A(x)) \vee B \equiv \forall x\, A(x) \vee B$ if x is not free in B
6. $(\exists x\, A(x)) \wedge B \equiv \exists x\, A(x) \wedge B$ if x is not free in B
7. $\forall x \forall y\, C(x, y) \equiv \forall y \forall x\, C(x, y)$
8. $\exists x \exists y\, C(x, y) \equiv \exists y \exists x\, C(x, y)$

Proof. We give the proofs of 1, 3, 5, and 7, and leave the rest as exercises.

1. Consider any interpretation I, using de Morgan's law, $\neg(A \wedge B) \equiv \neg A \vee \neg B$,

$$\begin{array}{rl} & I(\neg\forall x\, A(x)) \\ = & \neg I(\forall x\, A(x)) \\ = & \neg \bigwedge_{d \in D^I} I(A(x), \{x \leftharpoonup d\}) \\ = & \bigvee_{d \in D^I} \neg I(A(x), \{x \leftharpoonup d\}) \\ = & \bigvee_{d \in D^I} I(\neg A(x), \{x \leftharpoonup d\}) \\ = & I(\exists x\, \neg A(x)). \end{array}$$

Because I is arbitrary, the equivalence holds.

3. Consider any interpretation $I = (D, R, G)$, using the commutativity and associativity of $\wedge$,

$$\begin{array}{rl} & I(\forall x\, A(x) \wedge B(x)) \\ = & \bigwedge_{d\in D} I(A(x) \wedge B(x), \{x \leftharpoonup d\}) \\ = & \bigwedge_{d\in D} I(A(x), \{x \leftharpoonup d\}) \wedge I(B(x), \{x \leftharpoonup d\}) \\ = & (\bigwedge_{d\in D} I(A(x), \{x \leftharpoonup d\})) \wedge (\bigwedge_{d\in D} I(B(x), \{x \leftharpoonup d\})) \\ = & I(\forall x\, A(x)) \wedge I(\forall x\, B(x)) \\ = & I((\forall x\, A(x)) \wedge (\forall x\, B(x))) \end{array}$$

Because I is arbitrary, the equivalence holds.

5. In the following proof, we assume that x does not appear in B. Consider any interpretation $I = (D, R, G)$, using the distribution law of $\vee$ over $\wedge$,

$$\begin{array}{rl} & I((\forall x\, A(x)) \vee B) \\ = & I(\forall x\, A(x)) \vee I(B) \\ = & (\bigwedge_{d\in D} I(A(x), \{x \leftharpoonup d\})) \vee I(B) \\ = & \bigwedge_{d\in D} (I(A(x), \{x \leftharpoonup d\}) \vee I(B)) \\ = & \bigwedge_{d\in D} (I(A(x), \{x \leftharpoonup d\}) \vee I(B, \{x \leftharpoonup d\})) \\ = & \bigwedge_{d\in D} I(A(x) \vee B, \{x \leftharpoonup d\}) \\ = & I(\forall x\, A(x) \vee B) \end{array}$$

Because I is arbitrary, the equivalence holds.

7. Consider any interpretation $I = (D, R, G)$, using the commutativity and associativity of $\wedge$,

$$\begin{array}{rl} & I(\forall x \forall y\, C(x, y)) \\ = & \bigwedge_{d\in D} I(\forall y\, C(x, y), \{x \leftharpoonup d\}) \\ = & \bigwedge_{d\in D} (\bigwedge_{e\in D} I(C(x, y), \{x \leftharpoonup d, y \leftharpoonup e\})) \\ = & \bigwedge_{e\in D} (\bigwedge_{d\in D} I(C(x, y), \{x \leftharpoonup d, y \leftharpoonup e\})) \\ = & \bigwedge_{e\in D} I(\forall x\, C(x, y), \{y \leftharpoonup e\}) \\ = & I(\forall y \forall x\, C(x, y)) \end{array}$$

Because I is arbitrary, the equivalence holds. □

Example 5.2.19. Consider $\exists x \forall y\, p(x, y)$ and $\forall y \exists x\, p(x, y)$. If $p(x, y)$ stands for "x loves y", then the former means "there exists somebody who loves everyone", but the latter means "everyone is loved by somebody". Obviously, the two are not equivalent. To show

$$\exists x \forall y\, p(x, y) \models \forall y \exists x\, p(x, y),$$

consider any model $I = (D, R, G)$ of $\exists x \forall y\, p(x, y)$. There must exists $d \in D$ such that $I(\forall y\, p(x, y), \{x \leftharpoonup d\}) = 1$. Then $\forall y \exists x\, p(x, y)$ must be true in I:

$$\begin{array}{ll} & I(\forall y \exists x\, p(x, y), \emptyset) \\ = & \bigwedge_{e \in D} I(\exists x\, p(x, y), \{y \leftharpoonup e\}) \\ = & \bigwedge_{e \in D} I(p(x, y), \{y \leftharpoonup e, x \leftharpoonup d\}) \\ = & I(\forall y\, p(x, y), \{x \leftharpoonup d\}) \\ = & 1 \end{array}$$

□

To formally show that the two formulas are not equivalent, we need to provide a model in which one is true and the other is false.

Example 5.2.20. Let us prove that $(\exists x\, p(x)) \to \forall y\, q(y)$ and $\forall x\, p(x) \to q(x)$ are not equivalent. Let $I = (\{a, b\}, \{r_p, r_q\}, \emptyset)$ where $r_p(a) = r_q(a) = 1$ and $r_p(b) = r_q(b) = 0$. Then $I(\exists x\, p(x)) = 1$ and $I(\forall y\, q(y)) = 0$. So $I((\exists x\, p(x)) \to \forall y\, q(y)) = 0$. On the other hand, $I(\forall x\, p(x) \to q(x)) = 1$ because $r_p = r_q$, thus $I(p(x) \to q(x), \{x \leftharpoonup d\}) = 1$. Hence, $(\exists x\, p(x)) \to \forall y\, q(y)$ and $\forall x\, (p(x) \to q(x))$ are not equivalent. □

Example 5.2.21. We may use known equivalence relations to show $\exists x\, A(x) \to B(x) \equiv (\forall x\, A(x)) \to \exists x\, B(x)$.

$$\begin{array}{lll} & \exists x\, A(x) \to B(x) & \\ \equiv & \exists x\, \neg A(x) \vee B(x) & //A \to B \equiv \neg A \vee B \\ \equiv & (\exists x\, \neg A(x)) \vee \exists x\, B(x) & //\text{Proposition 5.2.18 4.} \\ \equiv & \neg(\forall x\, A(x)) \vee \exists x\, B(x) & //\text{Proposition 5.2.18 1.} \\ \equiv & (\forall x\, A(x)) \to \exists x\, B(x) & //A \to B \equiv \neg A \vee B \end{array}$$

□

5.3 Proof Methods

In Chapter 3, we introduced various proof systems for propositional logic: semantic tableau, natural deduction, resolution, etc. All these systems can be extended to first-order logic. In this section, we briefly introduce the extensions of semantics tableau and natural deduction to first-order logic and leave resolution to the next chapter.

5.3.1 Semantic Tableau

Semantic tableau as given in Chapter 3 transforms a propositional formula into DNF by a set of α-rules (which handle conjunctions and generate one child in the tableau and β-rules (which handle disjunctions and generate two children in the tableau. These rules will be inherited for first-order logic. Keep in mind that the rules apply to the top logical operators in a formula in a leaf node of the tableaux.

α	α_1, α_2
$A \wedge B$	A, B
$\neg(A \vee B)$	$\neg A, \neg B$
$\neg(A \rightarrow B)$	$A, \neg B$
$\neg(A \oplus B)$	$(A \vee \neg B), (\neg A \vee B)$
$A \leftrightarrow B$	$(A \vee \neg B), (\neg A \vee B)$
$\neg(A \uparrow B)$	A, B
$A \downarrow B$	$\neg A, \neg B$
$\neg\neg A$	A

β	β_1	β_2
$\neg(A \wedge B)$	$\neg A$	$\neg B$
$A \vee B$	A	B
$A \rightarrow B$	$\neg A$	B
$A \oplus B$	$A, \neg B$	$\neg A, B$
$\neg(A \leftrightarrow B)$	$A, \neg B$	$\neg A, B$
$A \uparrow B$	$\neg A$	$\neg B$
$\neg(A \downarrow B)$	A	B

To handle the quantifiers, we introduce two rules, one for the universal quantifier, called the $\forall$-rule, and one for the existential quantifier, called the $\exists$-rule.

Definition 5.3.1. *The $\forall$-rule is an inference rule:*

$$(\forall) \quad \frac{\forall x\, A(x)}{A(t)} \qquad t \text{ is a ground term}$$

The $\forall$-rule works as follows: if $\forall x\, A(x)$ appears in a leaf node n, we create a child node n' of n, copy every formula of n into n', and add $A(t)$ into n', where t is a ground term (without variables) appearing in n. If n' does not have any ground term, we may introduce a new constant as t.

We emphasize that the $\forall$-rule is an inference rule, not a simplification rule, because it creates and adds something, not replacing $\forall x\, A(x)$ by something. Because of the $\forall$-rule, we may not be able to show the termination of the semantic tableau method.

Definition 5.3.2. *The $\exists$-rule is a simplification rule:*

$$(\exists) \quad \frac{\exists x\, A(x)}{A(c)} \qquad c \text{ is a new constant}$$

The $\exists$-rule works as follows: if $\exists x\, A(x)$ appears in a leaf node n, we create a child node n' of n, copy every formula of n into n', and replace $\exists x\, A(x)$ in n'

by $A(c)$. Later after you finish reading this chapter, you will know that the new constant is called *Skolem constant*, to remember Skolem's contribution to logic.

If the formulas are not in NNF (negation normal form), we also need the $\neg\exists$-rule, which is also an inference rule:

$$(\neg\exists)\quad \frac{\neg\exists x\, A(x)}{\neg A(t)} \qquad t \text{ is a ground term}$$

and the $\neg\forall$-rule, which is a simplification rule:

$$(\neg\forall)\quad \frac{\neg\forall x\, A(x)}{\neg A(c)} \qquad c \text{ is a new constant}$$

Example 5.3.3. To show $(\exists x\forall y\, p(x,y)) \to \forall y\exists x\, p(x,y)$ is valid by semantic tableaux, we show that the negation of the formula has a closed tableau.

$\epsilon:$	$\neg((\exists x\forall y\, p(x,y)) \to \forall y\exists x\, p(x,y))$	$\alpha\ \neg\ \to$
$1:$	$\exists x\forall y\, p(x,y),\ \neg\forall y\exists x\, p(x,y)$	$\exists$
$11:$	$\forall y\, p(c_1,y),\ \neg\forall y\exists x\, p(x,y)$	$\neg\forall$
$111:$	$\forall y\, p(c_1,y),\ \neg\exists x\, p(x,c_2)$	$\forall$
$1111:$	$\forall y\, p(c_1,y),\ \neg\exists x\, p(x,c_2), p(c_1,c_2)$	$\neg\exists$
$11111:$	$\forall y\, p(c_1,y),\ \neg\exists x\, p(x,c_2)),\ p(c_1,c_2),\ \neg p(c_1,c_2)$	*closed*

□

Example 5.3.4. Let A be $B\wedge C\wedge\wedge(\forall x\forall y\, \neg q(x)\vee\neg q(y))$ where $B = (\forall x\, p(x,s(x)))$ and $C = (\forall x\forall y\, q(x)\vee q(y))$. If we choose to work on q first, we will find a closed tableau for A, thus A is unsatisfiable. □

$\epsilon:$	$B\wedge C\wedge(\neg\exists x\, q(x))$	$\alpha\wedge$
$1:$	$B, C\wedge(\neg\exists x\, q(x))$	$\alpha\wedge$
$11:$	$B, C, (\neg\exists x\, q(x))$	$\forall$
$111:$	$B, C, (\neg\exists x\, q(x)), (\forall y\, q(c)\vee q(y)),$	$\forall$
$1111:$	$B, C, (\neg\exists x\, q(x)), (\forall y\, q(c)\vee q(y)), q(c)\vee q(c),$	$\neg\exists$
$11111:$	$B, C, (\neg\exists x\, q(x)), (\forall y\, q(c)\vee q(y)), q(c)\vee q(c), \neg q(c),$	$\beta\vee$
$111111:$	$B, C, (\neg\exists x\, q(x)), (\forall y\, q(c)\vee q(y)), q(c), \neg q(c),$	*closed*
$111112:$	$B, C, (\neg\exists x\, q(x)), (\forall y\, q(c)\vee q(y)), q(c), \neg q(c),$	*closed*

If we choose to work on p first, then the tableau will never terminate:

$\epsilon:$	$(\forall x\, p(x,s(x))), (\forall x\forall y\, q(x)\vee q(y)), (\neg\exists x\, q(x))$	$\forall$
$1:$	$(\forall x\, p(x,s(x))), ..., p(c,s(c)),$	$\forall$
$11:$	$(\forall x\, p(x,s(x))), ..., p(c,s(c)), p(s(c),s(s(c)))$	$\forall$
$111:$	$(\forall x\, p(x,s(x))), ..., p(c,s(c)), p(s(c),s(s(c))), p(s(s(c)),s(s(s(c))))$	$\forall$
$1...1:$	...	

This example shows clearly that the nondeterminism in the application of the rules (i.e., choosing a leaf node, choosing a rule, or choosing a term in the $\forall$ rule) does matter for obtaining a closed tableau. If we use a fair strategy for the application of the rules, then a closed tableau is guaranteed to be found when the input formula is unsatisfiable, as indicated by the following result (without a proof).

Theorem 5.3.5. *A formula A is unsatisfiable iff A has a closed tableau, which can be found by a fair strategy.*

5.3.2 Natural Deduction

In natural deduction for propositional logic, we have an introduction rule and an elimination rule for each logical operator. These rules will be inherited, plus the new rules for the quantifiers over the sequents:

op	**introduction**	**elimination**
$\forall$	$(A(x) \mid \alpha) \vdash (\forall y\, A(x) \mid \alpha)$	$(\forall x\, A(x) \mid \alpha) \vdash (A(t) \mid \alpha)$
$\exists$	$(A(t) \mid \alpha) \vdash (\exists x\, A(x) \mid \alpha)$	$(\exists x\, A(x) \mid \alpha) \vdash (A(c) \mid \alpha)$

where x and y are variables, t is a term and c is a new constant. From these rules, we can see that free variables have the same function as universally quantified variables. We use two examples to illustrate these rules.

Example 5.3.6. Let $A = \{\forall x\, p(x), \forall x\, q(x)\}$. A proof of $A \models \forall y\, p(y) \wedge q(y)$ in natural deduction is given below. □

1.	$(\forall x\, p(x))$	assumed
2.	$(\forall x\, q(x))$	assumed
3.	$(p(y))$	$\forall_E$ from 1
4.	$(q(y))$	$\forall_E$ from 2
5.	$(p(y) \wedge q(y))$	$\wedge_I$ from $3, 4$
6.	$(\forall y\, p(y) \wedge q(y))$	$\forall_I$ from 5

By Theorem 5.2.16, $(\forall x\, p(x) \wedge (\forall x\, q(x)) \rightarrow \forall y\, p(y) \wedge q(y)$ is valid.

Example 5.3.7. Show that $(\exists x\, p(x) \vee q(x)) \models (\exists x\, p(x)) \vee (\exists x\, q(x))$ in natural deduction. □

1.	$(\exists x\, p(x) \vee q(x))$	assumed
2.	$(p(c) \vee q(c))$	$\exists_E$ from 1
3.	$(p(c) \mid q(c))$	$\vee_E$ from 2
4.	$(\exists x\, p(x) \mid q(c))$	$\exists_I$ from 3
5.	$(\exists x\, p(x) \mid (\exists x\, q(x)))$	$\exists_I$ from 4
6.	$(\exists x\, p(x) \vee (\exists x\, q(x)))$	$\vee_I$ from 5

By Theorem 5.2.16, $(\exists x\, p(x) \vee q(x)) \rightarrow ((\exists x\, p(x)) \vee (\exists x\, q(x)))$ is valid.

5.4 Conjunctive Normal Form

We just introduced an extension of semantic tableau and natural deduction to first-order logic and illustrated the use of these two methods for proving theorems in first-order logic. In practice, these two methods are not as effective as the resolution method, which is the focus of the next chapter.

Recall that in propositional logic, the resolution is a refutational method and the input formula must be in clausal form. This is still true for first-order logic. In this section, we discuss how to convert first-order formulas into clausal form.

Definition 5.4.1. *In first-order logic, a* literal *is either an atom (positive literal) or the negation of an atom (negative literal); a* clause *is a disjunction of literals; a formula is in CNF if it is a conjunction of clauses.*

From the definition, if we replace each atom in a clause by a propositional variable, we obtain a clause in propositional logic. Since clauses use only three logical operators, i.e., $\neg$, $\vee$ and $\wedge$, we need to get rid of $\rightarrow$, $\leftrightarrow$, $\oplus$, etc., as we did in propositional logic. We also need to get rid of quantifiers and we will talk about it now.

5.4.1 Prenex Normal Form

Definition 5.4.2. *A formula A is in* prenex normal form *(PNF) if*

$$A = Q_1 x_1 Q_2 x_2 \cdots Q_k x_k B(x_1, x_2, ..., x_k),$$

where $Q_i \in \{\forall, \exists\}$, and B does not contain any quantifiers.

The idea of prenex normal form is to move all the quantifiers to the top of the formula. For example, $\forall x \forall y\, p(x, y) \vee q(x)$ is a PNF but $\forall x(\forall y\, p(x, y)) \vee q(x)$ is not.

To make the discussion easier, we assume that the formulas contain only $\neg$, $\wedge$, $\vee$, plus the two quantifiers, as operators. A quantifier which is not at the top must have another operator immediately above it: in this case the operator will be one of the three operators: $\neg$, $\wedge$ or $\vee$. Since the quantifier can be in either parameter position of $\wedge$ or $\vee$, and there are two quantifiers, there are 10 cases to consider. If we consider $\wedge$ and $\vee$ are commutative, we need six rules, which are provided in Proposition 5.2.18:

Equivalence Rules for PNF

1. $\neg\forall x\, A(x) \equiv \exists x\, \neg A(x)$
2. $\neg\exists x\, A(x) \equiv \forall x\, \neg A(x)$
3. $(\forall x\, A(x)) \wedge B \equiv (\forall x\, A(x) \wedge B)$ if x is not free in B
4. $(\forall x\, A(x)) \vee B \equiv (\forall x\, A(x) \vee B)$ if x is not free in B
5. $(\exists x\, A(x)) \wedge B \equiv (\exists x\, A(x) \wedge B)$ if x is not free in B
6. $(\exists x\, A(x)) \vee B \equiv (\exists x\, A(x) \vee B)$ if x is not free in B

where the arguments of $\vee$ and $\wedge$ can be switched.

To meet the condition that "x is not free in B", it is necessary first to rename all the quantified variables so that no variable appears more than once following a quantifier.

In Chapter 2, we introduced the concept of "negation normal form", where the negation appears only in literals. Using the first two of the above rules plus de Morgan's laws and the double negation law, we may push the negation down, until all negations appear in literals. The resulting formulas are also said to be in *negation normal form* (NNF), a straightforward extension from propositional logic.

Proposition 5.4.3. *Every first-order formula can be transformed into an equivalent formula in NNF.*

Example 5.4.4. Given the following formula

$$(\forall x \exists y\, p(x, y)) \wedge \neg(\forall x \forall y\, q(x, y) \wedge q(y, x)),$$

we may choose to rename the variable names first so that every quantified variable has a different name:

$$(\forall x \exists y\, p(x, y)) \wedge \neg(\forall z \forall w\, q(z, w) \wedge q(w, z))$$

Applying equivalence rule 1 twice,

$$(\forall x \exists y\, p(x, y)) \wedge \exists z \exists w\, \neg(q(z, w) \wedge q(w, z))$$

Now we may apply the deMorgan's law to get

$$(\forall x \exists y\, p(x, y)) \wedge \exists z \exists w\, \neg q(z, w) \vee \neg q(w, z))$$

which is in NNF; this step is not required for obtaining PNF. We have two options now: either choose equivalence rule 3 or rule 5. To use rule 3, we have

$$\forall x\,(\exists y\,p(x,y)) \wedge \exists z \exists w\,\neg q(z,w) \vee \neg q(w,z))$$

To apply rule 5 three times, we have

$$\forall x \exists y \exists z \exists w\,p(x,y) \wedge (\neg q(z,w) \vee \neg q(w,z)),$$

which is a PNF. □

Note that the order of quantified variables in this formula is $xyzw$. There are five other possible outcomes where the order is $xzyw$, $xzwy$, $zxyw$, $zxwy$, or $zwxy$. If we are allowed to switch the quantifiers of the same type (the last two in Proposition 5.2.18), we obtain more equivalent formulas where the order is $xywz$, $xwyz$, $xwzy$, $wxyz$, $wxzy$, or $wzxy$. These formulas are all equivalent as we use the equivalence relations in obtaining PNF.

Proposition 5.4.5. *Every first-order formula can be transformed into an equivalent PNF.*

Proof. By Proposition 5.2.12, renaming quantifiers variable names preserve the equivalence. The equivalence rules for PNF are terminating (an exercise problem) and preserve the equivalence. Apply these rules repeatedly, we will obtain the desired result. □

5.4.2 Skolemization

Skolemization is the process in which we remove all the quantifiers, both universal and existential, leaving a formula with only free variables. This process is made easier by having the original formula in prenex normal form, and we assume this in this section.

Removing the universal quantifiers while preserving meaning is easy since we assume that the meaning of a free variable is the same as a universally quantified variable: The formula will be true for any value of a free variable. We need only to discard the universal quantifiers from a PNF.

Doing the same for an existentially quantified variable is not possible because a free variable can have only one meaning. The key idea is to introduce a new function to replace that variable. These are called *Skolem functions*. When the function are nullary they are called *Skolem constants*.

Example 5.4.6. Consider the formula $\forall x \exists y\, y^2 \leq x \wedge \neg(y+1)^2 \leq x$. This asserts the existence of a maximal number y whose square is no more than x for any value of x. We may drop the quantifier $\forall$ so that x becomes free and keep the meaning of the formula. The same effect could be achieved by asserting the existence of a function f, depending on x, satisfying

$$f(x)^2 \leq x \wedge \neg(f(x)+1)^2 \leq x.$$

The function f is indeed a Skolem function. We might then use our knowledge of algebra to prove that $f(x) = \lfloor \sqrt{x} \rfloor$, but this is beyond the scope of our present concerns. □

In general, the parameters to a Skolem function will be free variables, which have the same meaning as universally quantified variables, thus allowing a different value for each different values of free variables.

Algorithm 5.4.7. $Sko(A)$ takes as input a formula A and returns a formula without a formula without quantifiers. Procedure *PrenexNF*(A) will return a formula in PNF and equivalent to A. $free(B)$ returns the list of free variables in B.

proc $Sko(A)$
1 $A :=$ *PrenexNF*(A);
2 **while** A has a quantifier **do**
3 **if** $(A = \exists x\, B)$ $A := B[x \leftharpoonup f(free(B))]$; // f is a new Skolem function
4 **else if** $(A = \forall x\, B)$ $A := B$;
5 **return** A;

In the algorithm $Sko(A)$, we see that a formula is first converted into PNF, then each quantifier is peeled off from left to right, until they are all eliminated.

Example 5.4.8. In Example 5.4.4, let

$$A = \forall x \exists y \exists z \exists w\, p(x,y) \wedge (\neg q(z,w) \vee \neg q(w,z)),$$

then $Sko(A) = p(x, f_1(x)) \wedge (\neg q(f_2(x), f_3(x)) \vee \neg q(f_3(x), f_2(x)))$. For

$$B = \exists z \exists w \forall x \exists y\, p(x,y) \wedge (\neg q(z,w) \vee \neg q(w,z)),$$

$Sko(B) = p(x, f_3(x)) \wedge (\neg q(c_1, c_2) \vee \neg q(c_2, c_1))$. We know that $A \equiv B$. $Sko(A) \not\equiv Sko(B)$, however, because they contain different function symbols. □

The above example illustrates that Skolemization does not preserve the equivalence. The Skolemization process, applied to a formula in prenex normal form, terminates with a weakly equivalent formula. To see that the process terminates is easy, since the process of converting a formula into PNF is terminating and the number of quantifiers is reduced in the body of the while loop of $Sko(A)$. The loop continues while a quantifier remains at the top of the formula, so the final formula will have no quantifiers. The input and output formulas are not necessarily equivalent, but their meanings are closely related, and we will capture this relationship with the notion "equally satisfiable".

Recall that in propositional logic (Definition 3.3.6), we say two formulas, A and B, are equally satisfiable, denoted by $A \approx B$, if $\mathcal{M}(A) = \emptyset$ whenever $\mathcal{M}(B) = \emptyset$ and vice versa. This relation is weaker than the logical equivalence, which requires $\mathcal{M}(A) = \mathcal{M}(B)$. We will reuse $\approx$ for first-order formulas.

It is easy to see that elimination of universal quantifiers preserves equivalence. The difficulties arise with the introduction of Skolem functions/constants.

Lemma 5.4.9. *Let A be $\forall x \exists y\, B(x, y)$. Then there exists a function $f(x)$ such that (a) $\forall x\, B(x, f(x)) \models A$; and (b) $\forall x\, B(x, f(x)) \approx A$.*

Proof. (a): For any model I of $\forall x\, B(x, f(x))$, and for every $d \in D^I$, it must be the case that $I(B(x, f(x)), \{x \leftharpoonup d\}) = 1$. Let $f^I(d) = e \in D^I$, where $f^I : D^I \longrightarrow D^I$ is the interpretation of f in I, then $I(B(x, y), \{x \leftharpoonup d, y \leftharpoonup e\}) = 1$. So we have

$$I(\exists y\, B(x, y), \{x \leftharpoonup d\}) = 1$$

Since d is arbitrary in D^I, $I(\forall x \exists y\, B(x, y)) = 1$. In other words, I is a model of A.

(b): Since (a) says "if $\forall x\, B(x, f(x))$ is satisfiable, so is A", we need only to show that if A is satisfiable, so is $\forall x\, B(x, f(x))$. Suppose $I = (D, R, G)$ is a model of A. Then for any $d \in D$, there exists $e \in D$ such that

$$I(B(x, y), \{x \leftharpoonup d, y \leftharpoonup e\}) = 1$$

Let $g : D \longrightarrow D$ such that $f(d) = e$, and $I' = (D, R, G \cup \{g\})$ with $g = f^{I'}$. Then I' is a model of $\forall x\, B(x, f(x))$. □

Corollary 5.4.10. *Let $A = \forall x_1 \forall x_2 \cdots \forall x_k \exists y\, B(x_1, x_2, ..., x_k, y)$. Then there exists a function $f(x_1, x_2, ..., x_k)$ such that*
(a) $\forall x_1 \forall x_2 \cdots \forall x_k\, B(x_1, x_2, ..., x_k, f(x_1, x_2, ..., x_k)) \models A$; and
(b) $\forall x_1 \forall x_2 \cdots \forall x_k\, B(x_1, x_2, ..., x_k, f(x_1, x_2, ..., x_k)) \approx A$.

Proof. Replace x by $x_1, x_2, ..., x_k$ in the proof of Lemma 5.4.9. □

Theorem 5.4.11. *(a)* $Sko(A) \models A$*; and (b)* $Sko(A) \approx A$*.*

Proof. Since a free variable and a universally quantified variable have the same interpretation, drop a universal quantifier does not change the semantics of a formula. When we remove an existential quantifier, we introduce a Skolem function. By Corollary 5.4.10, the resulting formula is equally satisfiable to the original formula. Applying Corollary 5.4.10 to every existential quantifier, we have the desired result. □

By Corollary 5.4.10 (a), A is a logical consequence of $Sko(A)$, i.e., $Sko(A) \models A$, but not $A \models Sko(A)$. However, we have $A \approx Sko(A)$, this is what we mean by "weakly equivalent", and we will see that it is sufficient for our purposes.

5.4.3 Skolemizing Non-Prenex Formulas

Although Skolemization is easy to explain and justify on formulas in PNF, it can actually be applied directly to any formula. The process involved is not easily explained as in the case of PNF. Instead we will provide a recursive algorithm for Skolemization which takes two parameters: The first parameter is a formula to be Skolemized, which must be in NNF (negation normal form), and the second parameter is the list of free variables collected so far.

Algorithm 5.4.12. $Skol(A, V)$ takes as input a formula A which is in NNF and a list V of variables which are free in A, and returns a formula without quantifiers.

proc $Skol(A, V)$
1 **if** A is a literal or $A = \top$ or $A = \bot$ **return** A; // base case
2 **if** $(A = B \; op \; C)$ **return** $Skol(B, V) \; op \; Skol(C, V)$; // *op* is either $\wedge$ or $\vee$
3 **if** $(A = \exists x \, B)$ **return** $Skol(B[x \leftharpoonup f(V)], V)$; // f is a new Skolem function
4 **if** $(A = \forall x \, B)$ **return** $Skol(B, V \cup \{x\})$;

When A is a literal, i.e., either an atom, or the negation of an atom, A is returned. When the topmost operator is $\vee$ or $\wedge$, the situation is simple: *Skol* is merely called recursively on each of the arguments of the operator. If the topmost symbol is $\exists$, *Skol* introduces a new Skolem function f (a Skolemn constant if V is empty) and replace every occurrence of x by $f(V)$. If the topmost symbol is $\forall$, the variable x is collected in the second parameter for the recursive call on B. Assuming computing $B[x \leftharpoonup f(V)]$ takes linear time, *Skol* will take quadratic time in terms of the size of the input formula.

Example 5.4.13. Let A be the first NFF in Example 5.4.4

$$A = (\forall x \exists y\, p(x, y)) \wedge (\exists z \exists w\, \neg q(z, w) \vee \neg q(w, z))$$

□

Then $Skol(A, (\ ))$ calls $Skol$ on the two arguments of $\wedge$:

- $Skol(\forall x \exists y\, p(x, y), \emptyset)$ calls $Skol(\exists y\, p(x, y)), (x))$ and returns $p(x, f(x))$;
- $Skol(\exists z \exists w\, \neg q(z, w) \vee \neg q(w, z), \emptyset)$ returns $\neg q(c_1, c_2) \vee \neg q(c_2, c_1)$.

Hence $Skol(A)$ returns $p(x, f(x)) \wedge (\neg q(c_1, c_2) \vee \neg q(c_2, c_1))$.

Theorem 5.4.14. *(a)* $Skol(A) \models A$, *and (b)* $Skol(A) \approx A$.

The proof of the above theorem is similar to that of Theorem 5.4.11 and is omitted here. As pointed out in Example 5.4.4, we have many different PNFs for the same formula and each PNF may give us a different result after Skolemization. All PNFs of the same formula are logically equivalent and all Skolemized formulas of the same formula are equally satisfiable.

In practice, we wish to produce Skolemized formulas where Skolem functions have minimal number of arguments, so that the ground terms of these formulas will be simpler to deal with. Note that the equivalence rules for obtaining PNFs try to extend the scope of each quantified variable. To obtain Skolem functions with minimal number of arguments, we may need to use these equivalence rules backward.

5.4.4 Clausal Form

By CNF, we meant that the formula is a conjunction of clauses, and a clause is a disjunction of literals. To obtain CNF, a formula A goes through the following stages:

- stage 1: Convert A into NNF A';
- stage 2: Skolemize A' into quantifier-free B;
- stage 3: Convert B into CNF C.

Stage 3 converts Skolemized NNF B into clauses, as we did in propositional logic to convert NNF into clauses. That is, use the distribution law $A \vee (B \wedge C) \equiv (A \vee C) \wedge (A \vee C)$ repeatedly, until it no longer applies.

Stages 1 and 3 preserve the logical equivalence, $\equiv$, and stage 2 preserves only a weak equivalence, $\approx$. Thus, we have $A \equiv A'$, $A' \approx B$, and $B \equiv C$. As a result, $A \approx C$. So A is satisfiable iff C is satisfiable.

Example 5.4.15. Consider the following formulas and their intended meanings:

- $A = \forall x\, human(x) \rightarrow mortal(x)$: every human is mortal.
- $B = \forall x\, iowan(x) \rightarrow human(x)$: every Iowan is human.
- $C = \forall x\, iowan(x) \rightarrow mortal(x)$: every Iowan is mortal.

To show $A \wedge B \models C$, we convert $A \wedge B \wedge \neg C$ into a set S of four clauses, where c_1 comes from A, c_2 from B, and c_3 and c_4 from $\neg C$:

$$c_1 : (\neg human(x) \mid mortal(x)),$$
$$c_2 : (\neg iowan(y) \mid human(y)),$$
$$c_3 : (iowan(c)),$$
$$c_4 : (\neg mortal(c))$$

An instance of c_1 is $c_1' = (\neg human(c) \mid mortal(c))$, and an instance of c_2 is $c_2' = (\neg iowan(c) \mid human(c))$. Resolution between c_3 and c_2', we obtain $c5 = (human(c))$. Resolution between c_5 and c_1', we obtain the empty clause ( ). □

Our formulas are now in clausal form, and this is the normal form required by the resolution rule. Keep in mind the variables in a clause are assumed to be free and have the same semantics as the universally quantified variables, so that they can be renamed. These variables are different from clauses to clauses, even though they use the same name. A practice is to give different names for variables in different clauses. For example, in clause C_i, we use x_i, y_i, z_i, etc. This way, we ensure that no variable appears in more than one clause.

Stage 3 may blow up the sizes of the output formulas, while stage 2 outputs a formula of the linear size of the input formula. In propositional logic, to obtain a set of clauses of linear size of the original formula, we may introduce new variables for each occurrence of binary operators and the resulting clauses are 3CNF, which are not equivalent, but are equally satisfiable, to the original formula. We may do the same in stage 3 so that the final clauses are of linear size of the original formula.

Clausal form might seem a highly constrained and unnatural representation for mathematical knowledge, and it has been criticized on those grounds. We will see that it is in fact surprisingly natural in many cases. In fact, a problem can be expressed as a set of constraints over some variables, such as constraint satisfaction problems (CSP). A constraint can be expressed in CNF and a set of constraints give us a set of clauses.

We have seen that any predicate calculus formula can be converted into an equally satisfiable formula in CNF, which is useful for proving theoretical results by resolution.

5.4.5 Herbrand Models for CNF

When we consider interpretations for a set of formulas, there are simply too many ways to choose a set as the universe, too many ways to choose a relation for a predicate symbol, too many ways to choose a function for a function symbol, and use the elements of each universe in all possible ways for variables. We can improve on this considerably because there is a class of interpretations, called *Herbrand interpretations*, which can stand for all the others. The Herbrand interpretations share the same universe, called the *Herbrand universe*. What must this universe contain? For each n-ary function, f, in the formula (including constants) and n elements, $t_1, t_2, ..., t_n$, in the universe, it must contain the result of applying f to these n elements. This can be achieved by a simple syntactic device; we put all the ground terms in the universe and let each term denote the result of applying its function to its parameters. We will call this syntactic device "self-denotation". This way, function symbols are bounded to a unique interpretation over the universe.

Definition 5.4.16. *Given a CNF formula A in a first-order language $L = (P, F, X, Op)$, the* Herbrand universe *of A consists of all the ground terms, i.e., the set $T(F)$; the* Herbrand base *is the set of all ground atoms, denoted by $B(P, F)$. A Herbrand interpretation is a mapping $I : B(P, F) \longrightarrow \{1, 0\}$.*

Note that $T(F)$ is the set of variable-free terms that we can make from the constants and functions in the formula. If there are no constants in F, $T(F)$ would be empty. To avoid this case, we assume that there must exist at least one constant in F, say a. From now on it will be useful to distinguish the constants from non-constant functions, so by "function" we mean non-nullary function. If F does not contain function symbols, then $T(F)$ will be a finite set of constants.

The Herbrand base $B(P, F)$ consists of all the atoms like $p(t_1, t_2, ..., t_n)$, where p is an n-ary predicate symbol in P and $t_i \in T(F)$. If $n = 0$ for all predicates, then $B(P, F) = P$, reduced to propositional logic. Otherwise, $B(P, F)$ is finite iff $T(F)$ is finite. For convenience, we may represent I by a set H of ground literals such that for any $g \in B(P, F)$, $g \in H$ iff $I(g) = 1$ and $\overline{g} \in H$ iff $I(g) = 0$.

Example 5.4.17. In Example 5.2.13, we showed that the set S has no finite models:

$$S = \{\forall x \exists y\, p(x, y),\ \forall x\, \neg p(x, x),\ \forall x \forall y \forall z\, p(x, y) \wedge p(y, z) \rightarrow p(x, z)\}.$$

Note that S has no function symbols. Converting S to CNF, we obtain the following set S' of clauses:

$$S' = \{c_1 : (p(x, f(x))),\ c_2 : (\overline{p(x, x)}),\ c_3 : (\overline{p(x, y)} \mid \overline{p(y, z)} \mid p(x, z))\}.$$

Adding a constant a, $F = \{f, a\}$ and $P = \{p\}$, then the Herbrand universe of S' is

$$T(F) = \{a, f(a), f(f(a)), ..., f^i(a), ...\},$$

where $f^0(a) = a$ and $f^{i+1}(a) = f(f^i(a))$. The Herbrand base of S' is

$$B(P, F) = \{p(a, a), p(a, f(a)), p(f(a), a), p(a, f(f(a))), ..., p(f^i(a), f^j(a)), ...\}.$$

A Herbrand interpretation of S' is

$$H = \{p(f^i(a), f^j(a)) \mid 0 \leq i < j\} \cup \{\overline{p(f^i(a), f^j(a))} \mid 0 \leq j \leq i\}.$$

□

Proposition 5.4.18. *A Herbrand interpretation defines a unique interpretation for a formula in CNF.*

Proof. Let A be a formula in CNF and H is a Herbrand interpretation. Define interpretation $I = (D, R, G)$, where $D = T(F)$ (the Herbrand universe),

$$R = \{r_p \mid p \in P, r_p = \{\langle t_1, ..., t_n\rangle \mid H(p(t_1, ..., t_n)) = 1\}\}$$

and $G = \{g_f \mid f \in F, g_f(t_1, ..., t_n) = f(t_1, ..., t_n)\}$, then I is an interpretation as defined in Definition 5.2.1 and is uniquely defined by H. □

Since D and G are fixed in I, H is the only part which affects I. That is, the only way in which Herbrand interpretations differ from one to another is over the meanings of predicate symbols. The above proposition gives us the justification for calling H "an interpretation".

Once we have an interpretation, we can compute the truth value of a formula A in this interpretation using a procedure like *eval* (Procedure 5.2.3). If A is true under a Herbrand interpretation H, we say H is a *Herbrand model*. Since a CNF A can be represented by a set S of clauses, and the variables in each clause have the same meaning as universally quantified variables, for any interpretation I, $I(A) = I(S) = \bigcap_{C \in S} I(C)$, and for each clause C, $I(C) = I(\forall x_1 ... \forall x_k\, C)$, where x_1, ..., x_k are the free variables of C.

The definition of Herbrand interpretation gives us a convenient way of expressing these meanings by assigning the truth value to each ground atom in the Herbrand base. Of course, the truth values of some atoms are irrelevant if these atoms cannot be instances of any predicate in the formula, as the truth of the formula does not depend on these atoms. From now on, when we say "a set of clauses has a model" we meant "it has a Herbrand model".

5.5 Exercise Problems

1. Using a first-order language with variables ranging over people, and predicates $trusts(x, y)$, $politician(x)$, $crazy(x)$, $know(x, y)$, and $related(x, y)$, and $rich(x)$, write down first-order sentences asserting the following:

   (a) Nobody trusts a politician.

   (b) Anyone who trusts a politician is crazy.

   (c) Everyone knows someone who is related to a politician.

   (d) Everyone who is rich is either a politician or knows a politician.

2. Let the meaning of $taken(x, y)$ be "student x has taken CS course y". Please write the meaning of the following formulas:

   (a) $\exists x \exists y\, taken(x, y)$

   (b) $\exists x \forall y\, taken(x, y)$

   (c) $\forall x \exists y\, taken(x, y)$

   (d) $\exists y \forall x\, taken(x, y)$

   (e) $\forall y \exists x\, taken(x, y)$

3. Let $c(x)$ mean "x is a criminal" and let $s(x)$ mean "x is sane". Then formalize each of the following sentences.

   (a) All criminals are sane.

   (b) Every criminal is sane.

   (c) Only criminals are sane.

   (d) Some criminals are sane.

   (e) There is a criminal that is sane.

   (f) No criminal is sane.

   (g) Not all criminals are sane.

4. Given the predicates $male(x)$, $female(x)$, $parent(x, y)$ (x is a parent of y), define the following predicates in terms of *male, female, parent*, or other predicates defined from these three predicates, in first-order logic:

   (a) $father(x, y)$: x is the father of y.

   (b) $mother(x, y)$: x is the mother of y.

(c) *ancesotor*(x, y): x is an ancestor of y.

(d) *descendant*(x, y): x is an descendant of y.

(e) *son*(x, y): x is a son of y.

(f) *daughter*(x, y): x is a daughter of y.

(g) *sibling*(x, y): x is a sibling of y.

5. Let Set be defined by the following BNF:

$$\langle Set \rangle ::= \emptyset \mid adjoin(\langle Any \rangle, \langle Set \rangle)$$

where $adjoin(e, s)$ adjoins an element e into the set s. Please define the following predicates in terms of $\emptyset$ and $adjoin$ in first-order logic:

(a) *isSet*(x): x is an object of Set.

(b) *member*(x, s): x is a member of set s.

(c) *subset*(s, t): s is a subset of set t.

(d) *equal*(s, t): s and t contain the same elements.

6. Decide if the following formulas are valid, not valid but satisfiable, or unsatisfiable, using the definition of interpretations:

(a) $p(a, a) \lor \neg \exists x\, p(x, a)$

(b) $\neg p(a, a) \lor \exists x\, p(x, a)$

(c) $\neg q(a) \lor \forall x\, q(x)$

(d) $\neg(\exists x\, q(x)) \lor \forall x\, q(x)$

(e) $\exists x\, q(x) \lor \neg \forall x\, q(x)$

7. Prove the following equivalence relations:

(a) $\neg \exists x\, p(x) \equiv \forall x\, \neg p(x)$

(b) $\exists x\, p(x) \lor q(x) \equiv (\exists x\, p(x)) \lor (\exists x\, q(x))$

(c) $(\exists x\, p(x)) \land q \equiv \exists x\, (p(x) \land q)$

(d) $\exists x \exists y\, r(x, y) \equiv \exists y \exists x\, r(x, y)$

(e) $\exists x\, p(x) \to q(x) \equiv (\forall x\, p(x)) \to (\exists x\, q(x))$

8. Prove that the following entailment relations are true, but the converse of the relation is not true.

(a) $(\forall x\, p(x)) \vee (\forall x\, q(x)) \models \forall x\, p(x) \vee q(x)$

(b) $\exists x\, p(x) \wedge q(x) \models (\exists x\, p(x)) \wedge (\exists x\, q(x))$

(c) $(\forall x\, p(x)) \rightarrow (\forall x\, q(x)) \models \forall x\, p(x) \rightarrow q(x)$

(d) $(\exists x\, p(x)) \rightarrow (\exists x\, q(x)) \models \exists x\, p(x) \rightarrow q(x)$

(e) $(\exists x\, p(x)) \rightarrow (\forall x\, q(x)) \models \forall x\, p(x) \rightarrow q(x)$

(f) $\forall x\, p(x) \rightarrow q(x) \models (\exists x\, p(x)) \rightarrow (\exists x\, q(x))$

(g) $\forall x\, p(x) \rightarrow q(x) \models (\forall x\, p(x)) \rightarrow (\exists x\, q(x))$

(h) $\forall x\, p(x) \leftrightarrow q(x) \models (\forall x\, p(x)) \leftrightarrow (\forall x\, q(x))$

(i) $\forall x\, p(x) \leftrightarrow q(x) \models (\exists x\, p(x)) \leftrightarrow (\exists x\, q(x))$

9. Use the semantic tableau method to show the entailment relations are true in the previous problem.

10. Provide the pseudo-code for the algorithm *PrenexNF*(A) and show that it runs in $O(n)$ time when A is in NNF, where n is the size of A.

11. Show that the algorithm $Sko(A)$ runs in $O(n^2)$ time, where A is in NNF and n is the size of A.

12. Show that the algorithm $Skol(A)$ runs in $O(n^2)$ time, where A is in NNF and n is the size of A.

13. Convert each one of the following formulas into a set of clauses:

(a) $\neg((\forall x\, p(x)) \vee (\forall x\, q(x)) \rightarrow \forall x\, p(x) \vee q(x))$

(b) $\neg((\exists x\, p(x) \wedge q(x)) \rightarrow (\exists x\, p(x)) \wedge (\exists x\, q(x)))$

(c) $\neg(((\forall x\, p(x)) \rightarrow (\forall x\, q(x))) \rightarrow (\forall x\, p(x) \rightarrow q(x)))$

(d) $\neg(((\exists x\, p(x) \rightarrow q(x)) \rightarrow (\exists x\, p(x))) \rightarrow (\exists x\, q(x)))$

(e) $\neg((\exists x\, p(x)) \rightarrow (\forall x\, q(x)) \rightarrow (\forall x\, p(x) \rightarrow q(x)))$

(f) $\neg(((\forall x\, p(x) \rightarrow q(x)) \rightarrow (\exists x\, p(x))) \rightarrow (\exists x\, q(x)))$

(g) $\neg((\forall x\, p(x) \rightarrow q(x)) \rightarrow ((\forall x\, p(x)) \rightarrow (\exists x\, q(x))))$

(h) $\neg(((\forall x\, p(x) \leftrightarrow q(x)) \rightarrow (\forall x\, p(x))) \leftrightarrow (\forall x\, q(x)))$

(i) $\neg(((\forall x\, p(x) \leftrightarrow q(x)) \rightarrow (\exists x\, p(x))) \leftrightarrow (\exists x\, q(x)))$

14. Given a set S of clauses, $S = \{(p(f(x), c)), (\neg p(f(w), z) \mid p(f(f(w)), f(z))), (\neg p(y, f(y)))\}$, please answer the following questions: (a) What is the Herbrand universe of S? (b) What is the Herbrand base of S? (c) What is the minimum Herbrand model of S (no subset of this model is a model)? (d) What is the maximum Herbrand model of S (no superset of this model is a model)?

CHAPTER 6
UNIFICATION AND RESOLUTION

Resolution is known to be a refutational proof method from Chapter 3. By refutational method, we meant that the method will show that the input formula is unsatisfiable. If we would like to show that formula A is valid, we need to convert $\neg A$ into clauses and show that the clauses are unsatisfiable. To show that $A \models B$, we need to convert $A \wedge \neg B$ into CNF and show that the clauses are unsatisfiable. Note that A and B must be closed formulas. For example, if we want to express that $p(x, y)$ is a commutative relation, we may use the formula $p(x, y) \rightarrow p(y, x)$, where x and y are assumed to be arbitrary values. The negation of this formula is $\neg(\forall x \forall y\, p(x, y) \rightarrow p(y, x))$, not $\neg(p(x, y) \rightarrow p(y, x))$. The CNF of the former is $p(a, b) \wedge \neg p(b, a)$, where a and b are Skolem constants; the CNF of the latter is $p(x, y) \wedge \neg p(y, x)$. We can substitute y by x in this CNF and obtain two unit clauses $(p(x, x))$ and $(\overline{p(x, x)})$. These two clauses will generate the empty clause by resolution. The process of finding a substitution to make two atoms identical is called "unification", which is needed in the definition of the resolution rule for first-order formulas.

6.1 Unification

In Example 5.4.15, when we apply the resolution rule on two clauses, we need to find their instances such that one instance contains A and the other instance contains $\neg A$, where A is an atom. A clause has infinite many instances and the unification process helps us to find the right instances for resolution.

6.1.1 Substitutions and Unifiers

Given a first-order language $L = (P, F, X)$, a mapping $\sigma : X \longrightarrow T(F, X)$ is said to be a *substitution*. If $\sigma(x) \neq x$, we say x is *affected* by σ. If $\sigma(x)$ is a distinct variable for every affected x, σ is said to be a *renaming*.

When displaying σ, we only display the pair $x \leftharpoonup \sigma(x)$ for affected variable x, i.e., when $x \neq \sigma(x)$. For example, if $X = \{x, y\}$, then x is affected by $\sigma_1 = [x \leftharpoonup f(y)]$ and $\sigma_2 = [x \leftharpoonup f(x)]$; both x and y are affected by $\sigma_3 = [x \leftharpoonup f(y), y \leftharpoonup a]$.

A substitution $\sigma : X \longrightarrow T(F, X)$ can be extended to be a mapping $\sigma : T(F, X) \longrightarrow T(F, X)$ such that if $t = x$, then $\sigma(t) = \sigma(x)$; if $t = f(t_1, t_2, ..., t_k)$,

then $\sigma(t) = f(\sigma(t_1), \sigma(t_2), ..., \sigma(t_k))$. In a similar way, σ can be extended to be a mapping from formulas to formulas.

We used $p(t)$ to denote an instance of $p(x)$, i.e., $p(t) = p(x)[x \leftharpoonup t]$, where every occurrence of x is replaced by term t. Here, $\sigma = [x \leftharpoonup t]$ is a substitution and $p(t) = \sigma(p(x)) = p(x)\sigma$, as we are free to write either $p(x)\sigma$ or $\sigma(p(x))$.

Definition 6.1.1. (idempotent substitution) *σ is said be* idempotent *if for any term t, $t\sigma = (t\sigma)\sigma$.*

For example, $\sigma_1 = [x \leftharpoonup f(y)]$ is idempotent; $\sigma_2 = [x \leftharpoonup f(x)]$ is not idempotent, because $x\sigma_2 = f(x)$ and $(x\sigma_2)\sigma_2 = f(f(x))$. Neither $\sigma_3 = [x \leftharpoonup f(y), y \leftharpoonup a]$, because $x\sigma_3 = f(y)$ and $(x\sigma_3)\sigma_3 = f(a))$. Note that σ_3 can be modified to make it idempotent: $\sigma_3' = [x \leftharpoonup f(a), y \leftharpoonup a]$, while σ_2 cannot.

Proposition 6.1.2. *A substitution σ is* idempotent *iff for every $x \in X$, either $\sigma(x) = x$ or x does not appear in $\sigma(y)$ for any $y \in X$.*

Proof. Given any σ, if an affected x appears in $\sigma(y) = t(x)$, suppose $\sigma(x) = c$, then $y\sigma = t(x)$ and $(y\sigma)\sigma = t(c)$, so σ is not idempotent. Otherwise, for any term s, $s\sigma$ does not contain any variable affected by σ, so $(s\sigma)\sigma = s\sigma$ and σ is idempotent. □

Definition 6.1.3. (unifier) *Let s, t be either two terms of $T(F, X)$ or two atoms. We say s and t are* unifiable*, if there exists an idempotent substitution $\sigma : X \longrightarrow T(F, X)$ such that $s\sigma = t\sigma$; σ is said to be a* unifier *of s and t.*

A set S of pairs of terms or atoms is unifiable *if there exists a substitution σ such that for every pair $(s_i, t_i) \in S$. $s_i\sigma = t_i\sigma$ and σ is said to be a* unifier *of S.*

Example 6.1.4. Let $s = f(x, g(y))$ and $t = f(g(z), x)$, then s and t are unifiable with unifier $\sigma = [x \leftharpoonup g(z), y \leftharpoonup z]$, because

$$s\sigma = f(x, g(y))[x \leftharpoonup g(z), y \leftharpoonup z] = f(g(z), g(z)) = f(g(z), x)[x \leftharpoonup g(z), y \leftharpoonup z] = t\sigma.$$

In fact, s and t have infinite many unifiers like

$$[x \leftharpoonup g(k), y \leftharpoonup k, z \leftharpoonup k]$$

for any $k \in T(F, X)$. □

In the above example, we see that a pair of terms may have multiple unifiers. How do you compare these unifiers? We need some relation over unifiers and this relation is based on the composition of substitutions.

6.1.2 Combining Substitutions

Definition 6.1.5. (composition of substitution) *Given two substitutions σ and θ, the composition of σ and θ is the substitution γ, written $\gamma = \sigma \cdot \theta$, or simply $\sigma\theta$, where*

$$\gamma = [x \leftharpoonup t\theta \mid (x \leftharpoonup t) \in \sigma] \cup [y \leftharpoonup s \mid y = y\sigma, (y \leftharpoonup s) \in \theta]$$

We will assume that $t\sigma\theta = (t\sigma)\theta = \theta(\sigma(t))$. To denote the composition $\sigma\theta$ on t, we write $t(\sigma\theta)$.

Example 6.1.6. Given $\sigma = [x \leftharpoonup f(y), z \leftharpoonup f(u), v \leftharpoonup u]$ and $\theta = [y \leftharpoonup g(a), u \leftharpoonup z, v \leftharpoonup f(f(a))]$, then

$$\begin{aligned} \sigma\theta &= [x \leftharpoonup f(g(a)), y \leftharpoonup g(a), z \leftharpoonup f(z), u \leftharpoonup z, v \leftharpoonup z] \\ p(x,y,z,u,v)\sigma\theta &= p(f(y), y, f(u), u, u)\theta \\ &= p(f(g(a)), g(a), f(z), z, z) \\ &= p(x,y,z,u,v)(\sigma\theta) \end{aligned}$$

□

Note that the composition does not necessarily preserve the idempotent property as both σ and θ are idempotent but $z \leftharpoonup f(z)$ appears in $\sigma\theta$ in the above example.

Proposition 6.1.7. *For any substitutions σ and θ, and any term t, $t\sigma\theta = t(\sigma\theta)$.*

Proof. For any $x \in X$, if x is affected by σ, say $\sigma(x) = s$, then $x\sigma\theta = s\theta = x(\sigma\theta)$. If x is unaffected by σ, then $x\sigma\theta = x\theta = x(\sigma\theta)$. □

The above proposition states the property that applying two substitutions to a term in succession produces the same result as applying the composition to the term.

The composition operation is not commutative in general. For example, if $\sigma = [x \leftharpoonup a]$ and $\theta = [x \leftharpoonup b]$, then $\sigma\theta = \sigma$ and $\theta\sigma = \theta$. Thus, $\sigma\theta \neq \theta\sigma$. By definition, if x is affected by both σ and θ, then the pair $x \leftharpoonup \theta(x)$ is ignored in $\sigma\theta$ and the pair $x \leftharpoonup \sigma(x)$ is ignored in $\theta\sigma$.

Proposition 6.1.8. *The composition is associative, i.e., for any substitutions σ, θ, and γ, $(\sigma\theta)\gamma = \sigma(\theta\gamma)$.*

The proof is left as an exercise.

6.1.3 Rule-Based Unification

Definition 6.1.9. (**most general unifier**) *Let σ and θ be unifiers of s and t. We say σ is* more general *than θ, if there exists a substitution γ such that $\theta = \sigma\gamma$. σ is a* most general unifier *(mgu) of s and t, if σ is more general than any unifier of s and t.*

Example 6.1.10. In Example 6.1.4, $\sigma = [x \leftharpoonup g(z), y \leftharpoonup z]$ is a mgu of $s = f(x, g(y)$ and $t = f(g(z), x)$. For any unifier $\theta = [x \leftharpoonup g(t'), y \leftharpoonup t', z \leftharpoonup t']$, $\theta = \sigma\gamma$, where $\gamma = [z \leftharpoonup t']$. Another mgu of s and t is $\sigma' = [x \leftharpoonup g(y), z \leftharpoonup y]$, $\sigma = \sigma'[y \leftharpoonup z]$ and $\sigma' = \sigma[z \leftharpoonup y]$. □

As illustrate by the above example, there are more than one mgu for a pair of terms; these mgus differ by the renaming of variables. We can say *the* most general unifier if the renaming of variables is considered.

Given a pair s and t of terms, how to decide if s and t are unifiable? If they are, how to compute their mgu? These questions will be answered by unification algorithms.

Let $s \doteq t$ denote that "s and t are unifiable". Suppose $s = f(s_1, s_2, ..., s_m)$ and $t = g(t_1, t_2, ..., t_n)$. If $f \neq g$ or $m \neq n$, then $s \doteq t$ is false, i.e., s and t are not unifiable. This is called *clash* failure.

If $s \doteq t$ is true, so are $s_i \doteq t_i$ for $1 \leq i \leq m = n$. In other words, if $s_i \doteq t_i$ is false for some i, then $s \doteq t$ is false. If for some i, $s_i = x$ and $t_i = f(x)$ (x occurs in t_i) then there exists no substitution σ such that $x\sigma = f(x)\sigma$. Hence $x \doteq f(x)$ must be false. This is called *occur-check* failure, as x occurs in $f(x)$. If we accept $x \leftharpoonup f(x)$, then the unifier cannot be idempotent.

Rule-based unification algorithm uses a set of rules which transform $s \doteq t$ into a substitution σ through a set of pairs $\{s_i \doteq t_i\}$ and each rule preserves the unifiability of $s \doteq t$. These rules are given below.

Definition 6.1.11. *The unification rules for the rule-based unification algorithm are given below:*

- **decompose**: $\langle S \cup \{f(s_1, s_2, ..., s_m) \doteq f(t_1, t_2, ..., t_m)\}, \sigma\rangle \longrightarrow \langle S \cup \{s_i \doteq t_i \mid 1 \leq i \leq m\}, \sigma\rangle$.
- **clash**: $\langle S \cup \{f(s_1, s_2, ..., s_m) \doteq g(t_1, t_2, ..., t_n)\}, \sigma\rangle \longrightarrow \langle S, fail\rangle$.
- **occur-check**: $\langle S \cup \{x \doteq t \mid x \neq t, x \text{ occurs in } t\}, \sigma\rangle \longrightarrow \langle S, fail\rangle$.
- **redundant**: $\langle S \cup \{t \doteq t\}, \sigma\rangle \longrightarrow \langle S, \sigma\rangle$.
- **orient**: $\langle S \cup \{t \doteq x \mid t \text{ is not variable}\}, \sigma\rangle \longrightarrow \langle S \cup \{x \doteq t\}, \sigma\rangle$.

- **substitute**: $\langle S \cup \{x \doteq t \mid x \text{ does not occur in } t\}, \sigma\rangle \longrightarrow \langle S[x \leftharpoonup t], \sigma \cdot [x \leftharpoonup t]\rangle$

Algorithm 6.1.12. The algorithm *unify*(s,t) takes two terms s and t, and returns either $fail$ or σ, a mgu of s and t.

proc *unify*(s,t)
1 $S := \{s \doteq t\}$; $\sigma = \emptyset$;
2 **while** (true) **do**
3 $\langle S, \sigma\rangle :=$ apply one unification rule on $\langle S, \sigma\rangle$;
4 **if** $(\sigma = fail)$ **return** $fail$;
5 **if** $(S = \emptyset)$ **return** σ;

Example 6.1.13. Given $s = p(f(x, h(x), y), g(y))$ and $t = p(f(g(z), w, z), x)$, then *unify*(s,t) will transform $\langle\{s \doteq t\}, \emptyset\rangle$ into $\langle\emptyset, \sigma\rangle$ as follows:

$$
\begin{array}{rcl}
 & & \{s \doteq t\}, \emptyset \\
\textbf{decompose} & \longrightarrow & \{f(x, h(x), y) \doteq f(g(z), w, z), g(y) \doteq x\}, \emptyset \\
\textbf{orient} & \longrightarrow & \{f(x, h(x), y) \doteq f(g(z), w, z), x \doteq g(y)\}, \emptyset \\
\textbf{substitute} & \longrightarrow & \{f(g(y), h(g(y)), y) \doteq f(g(z), w, z)\}, [x \leftharpoonup g(y)] \\
\textbf{decompose} & \longrightarrow & \{g(y) \doteq g(z), h(g(y)) \doteq w, y \doteq z\}, [x \leftharpoonup g(y)] \\
\textbf{substitute} & \longrightarrow & \{g(z) \doteq g(z), h(g(z)) \doteq w\}, [x \leftharpoonup g(z), y \leftharpoonup z] \\
\textbf{redundant} & \longrightarrow & \{h(g(z)) \doteq w\}, [x \leftharpoonup g(z), y \leftharpoonup z] \\
\textbf{orient} & \longrightarrow & \{w \doteq h(g(z))\}, [x \leftharpoonup g(z), y \leftharpoonup z] \\
\textbf{substitute} & \longrightarrow & \emptyset, [x \leftharpoonup g(z), y \leftharpoonup z, w \leftharpoonup h(g(z))]
\end{array}
$$

□

The algorithm *unify*(s,t) does not specify the order of rules, so we may use these rules in any order when they are applicable. However, none of these rules can be applied forever. Hence *unify*(s,t) will terminate eventually.

Lemma 6.1.14. *Given any pair s and t,* unify(s,t) *will terminate with either $fail$ or σ.*

Proof. At first, we like to point out that the **substitute** rule may increase the sizes of terms in S but it moves variable x from S into σ, that is, x disappears from S, so this rule can be applied no more than the number of variables.

For the rules used between two applications of the **substitute** rule, if the **clash** rule or the **occur-check** rule apply, the process ends with $fail$ immediately. Otherwise, the **decompose** rule removes two occurrences of f. The **orient** rule switches $t \doteq x$ to $x \doteq t$, where t is not a variable. The **redundant** rule removes $t \doteq t$. None of these rule can be used forever. In the end, S becomes empty and σ will be returned. □

Lemma 6.1.15. *If $\langle S', \sigma' \rangle$ is the result of applying the* **substitute** *rule on $\langle S, \sigma \rangle$ and θ' is a mgu of S', then $\theta = [x \leftharpoonup t] \cdot \theta'$ is a mgu of S.*

Proof. By the definition of the **substitute** rule, $S' = (S - \{x \doteq t\})[x \leftharpoonup t]$ and $\sigma' = \sigma \cdot [x \leftharpoonup t]$, where x is not affected by σ. By the assumption, S' is unifiable with a mgu θ'. Since x does not appear in S', $\theta = [x \leftharpoonup t] \cdot \theta'$ is a mgu of $S' \cup [x \doteq t]$. It is easy to check that θ is also a mgu of S. □

Theorem 6.1.16. *The algorithm* unify(s, t) *will return a mgu of s and t iff s and t are unifiable.*

Proof. Lemma 6.1.14 ensures that *unify*(s, t) will terminate. Inside *unify*(s, t), when $\langle S, \sigma \rangle$ is transformed to $\langle S', \sigma' \rangle$, we can check for each rule that S is unifiable (σ does not affect the unifiability of S) iff $\sigma' \neq fail$ and S' is unifiable. That is, when σ' is $fail$, S cannot be unifiable. When $S' = \emptyset$, S' is trivially unifiable, so S must be unifiable.

Before the **substitute** rule moves $x \doteq t$ from S, x is not affected by σ; x is affected by $\sigma' = \sigma \cdot [x \leftharpoonup t] = \sigma[x \leftharpoonup t] \cup [x \leftharpoonup t]$, where t does not contain x. Since $S[x \leftharpoonup t]$ and $\sigma[x \leftharpoonup t]$ remove all the occurrences of x in S and σ, x has a unique occurrence in σ' and that is also true for all affected variables in σ. Thus, σ is idempotent by Proposition 6.1.2.

Now assume s and t are unifiable. Let $S_0 = \{s \doteq t\}$, $\sigma_0 = \emptyset$, and $\sigma_i = \sigma_{i-1} \cdot [x_i \leftharpoonup t_i]$ records every application of the **substitute** rule for $1 \leq i \leq m$. Let the corresponding S be $S_1, S_2, ..., S_m$, respectively, where $S_m = \emptyset$. Since the mgu of S_m is $\theta_m = \emptyset$. So by Lemma 6.1.15,

$$\theta_{m-1} = [x_m \leftharpoonup t_m]\theta_m = [x_m \leftharpoonup t_m]$$

is a mgu of S_{m-1}. Again by Lemma 6.1.15,

$$\theta_{m-2} = [x_{m-1} \leftharpoonup t_{m-1}]\theta_{m-1} = [x_{m-1} \leftharpoonup t_{m-1}, x_m \leftharpoonup t_m],$$

is a mgu of S_{m-2}, and so on. Finally,

$$\theta_0 = [x_1 \leftharpoonup t_1]\theta_1 = [x_1 \leftharpoonup t_1, x_{m-1} \leftharpoonup t_{m-1}, x_m \leftharpoonup t_m] = \sigma_m$$

is a mgu of $S_0 = \{s \doteq t\}$. □

Example 6.1.17. Consider the following pair of terms:

$$s = f(f(f(f(a, x_1), x_2), x_3), x_4),$$
$$t = f(x_4, f(x_3, f(x_2, f(x_1, a)))).$$

$$
\begin{array}{rcl}
 & & \{s \doteq t\}, \emptyset \\
\textbf{decompose} & \longrightarrow & \{f(f(f(a,x_1),x_2),x_3) \doteq x_4, x_4 \doteq f(x_3,f(x_2,f(x_1,a)))\}, \emptyset \\
\textbf{substitute} & \longrightarrow & \{f(f(f(a,x_1),x_2),x_3) \doteq f(x_3,f(x_2,f(x_1,a)))\}, \\
 & & [x_4 \leftharpoonup f(x_3,f(x_2,f(x_1,a)))] \\
\textbf{decompose} & \longrightarrow & \{f(f(a,x_1),x_2) \doteq x_3, x_3 \doteq f(x_2,f(x_1,a))\}, \\
 & & [x_4 \leftharpoonup f(x_3,f(x_2,f(x_1,a)))] \\
\textbf{substitute} & \longrightarrow & \{f(f(a,x_1),x_2) \doteq f(x_2,f(x_1,a))\}, \\
 & & [x_4 \leftharpoonup f(f(x_2,f(x_1,a)),f(x_2,f(x_1,a))), x_3 \leftharpoonup f(x_2,f(x_1,a))] \\
\textbf{decompose} & \longrightarrow & \{f(a,x_1) \doteq x_2, x_2 \doteq f(x_1,a)\}, [x_4 \leftharpoonup ..., x_3 \leftharpoonup f(x_2,f(x_1,a))] \\
\textbf{substitute} & \longrightarrow & \{f(a,x_1) \doteq f(x_1,a)\}, [x_4 \leftharpoonup ..., x_3 \leftharpoonup ..., x_2 \leftharpoonup f(x_1,a)] \\
\textbf{decompose} & \longrightarrow & \{a \doteq x_1, x_1 \doteq a\}, [x_4 \leftharpoonup ..., x_3 \leftharpoonup ..., x_2 \leftharpoonup f(x_1,a)] \\
\textbf{substitute} & \longrightarrow & \{a \doteq a\}, [x_4 \leftharpoonup ..., x_3 \leftharpoonup ..., x_2 \leftharpoonup f(a,a), x_1 \leftharpoonup a] \\
\textbf{redundant} & \longrightarrow & \emptyset, [x_4 \leftharpoonup ..., x_3 \leftharpoonup ..., x_2 \leftharpoonup f(a,a), x_1 \leftharpoonup a]
\end{array}
$$

Let $t_1 = a$, $t_2 = f(t_1,t_1)$, $t_3 = f(t_2,t_2)$, $t_4 = f(t_3,t_3)$, then the mgu of $s \doteq t$ is $[x_i \leftharpoonup t_i \mid 1 \leq i \leq 4]$.

In general, if s and t are the following pair of terms,

$$
\begin{array}{l}
s = f(f(f(...f(f(a,x_1),x_2),...),x_{n-1}),x_n) \\
t = f(x_n,f(x_{n-1},f(...,f(x_2,f(x_1,a))...)))
\end{array}
$$

then the mgu of s and t is $\sigma = [x_i \leftharpoonup t_i \mid 1 \leq i \leq n]$, where $t_1 = a$ and $t_{i+1} = f(t_i,t_i)$. The sizes of s and t are the same, i.e., $2n+1$. However, the size of t_n is $2^n - 1$. □

The above example illustrates that *unify*(s,t) will take $O(2^n)$ time and space for the input terms of size $O(n)$. This high cost cannot be avoided if we use trees to represent terms. Using the data structure of directed graphs for terms, identical subterms can be shared, so we do not need a space of exponential size.

6.1.4 Practically Linear Time Unification Algorithm

Corbin and Bidoit (1983) improved Robinson's unification algorithm to a quadratic one by using directed graphs. Ruzicka and Privara (1989) further improved it to an almost linear one, by using the union/find data structures and postponing the occur-check at the end of unification. The union/find data structure stores a collection of disjoint sets, each of the sets represents an equivalence class. Thus, the disjoint sets represent an equivalence relation and are a partition of the involved elements. The *Union* operation will merge two sets into one, and the *Find* operation will return a representative member of a set, so that it allows to find out efficiently if any two elements are in the same or different sets.

In the following, we introduce an almost-linear time unification algorithm different from that of Ruzicka and Privara.

Given two terms s and t, the *term graph* for s and t is the directed acyclic graph $G = (V, E)$ obtained by combining the formula trees of s and t with the following operation: All the nodes labeled with the same variable in s and t is merged into a single node of G.

The graph G has the following features:

- Each node v of G is labeled with a symbol in s or t, denoted by $label(v)$.
- The roots of the formula trees are the two nodes in G with zero in-degree; the other non-variable nodes' in-degrees are one. The in-degree of variable nodes is the sum of the occurrences of that variable in s and t.
- The leaf nodes of the formula trees are the nodes with zero out-degree in G, which are labeled with variables or constants.
- If a node is labeled with a function symbol f whose arity is k, then this node has k ordered successors. We will use $child(v, i)$ to refer to the i^{th} successor of v.

The term associated with each node n of G can be recursively defined as follows:

1. If $label(n)$ is a variable or constant, $term(n) = label(n)$;
2. If $label(n)$ is a function f other than a constant, $term(n) = f(s_1, s_2, ..., s_k)$, where $s_i = term(child(n, i))$ for $1 \leq i \leq arity(f)$.

We say $term(n)$ is the term represented by n in G.

The unification algorithm will work on $G = (V, E)$ and is divided into two stages: In stage 1, *unify*1 checks if s and t have no clash failure; in stage 2, *postOccurCheck* returns true iff they have no occur-check failure.

In stage 1, the algorithm *unify*1 takes the two nodes representing s and t, respectively, and visits G in tandem by depth-first search. During the search, we will assign to each node v of G another node u as the *representative* of v, denoted by $up(v) = u$. Initially, $up(v) = v$ for every node of G. If $up(v) = u$ and $v \neq u$, it means the terms represented by v and u, respectively, are likely unifiable (no clash failure) and we use u as the representative of v in the rest of the search. At the end of the search, *unify*1 returns true if no clash failure was found. In this case, the relation R defined by up, i.e., $R = \{(u, up(u)) \mid u \in V\}$, can generate an equivalence relation E (i.e., E is the reflexive, symmetric and transitive closure of R). The relation E partitions V into equivalence classes and the nodes in the same equivalent class represent the terms which must be unifiable if s and t are unifiable.

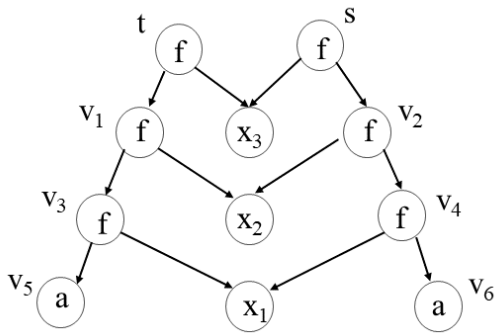


Figure 6.1.1: The term graphs for t and s

Example 6.1.18. Given $t = f(f(f(a, x_1), x_2), x_3)$ and $s = f(x_3, f(x_2, f(x_1, a)))$, their term graph is shown in Figure 6.1.1. The *up* relation is initialized as $up(v) = v$.

The algorithm *unify*1(t, s) starts the depth-first search in tandem and goes to the first children of (t, s), i.e., (v_1, x_3). Since x_3 is a variable node, *unify*1(v_1, x_3) will do $up(x_3) := v_1$ and backtrack. The second children of (t, s) are (x_3, v_2). Since $up(x_3) = v_1$, we use v_1 for x_3 and call *unify*1(v_1, v_2). Since v_1 and v_2 have the same label, the search goes to their children. The first children of (v_1, v_2) are (v_3, x_2). Since x_2 is a variable node, we do $up(x_2) := v_3$ and backtrack. The second children of (v_1, v_2) are (x_2, v_4). Using v_3 for x_2 and call *unify*1(v_3, v_4). Since v_3 and v_4 have the same label, the search goes to their children. The first children of (v_3, v_4) are (v_5, x_1). We do $up(x_1) := v_5$ and backtrack. The second children of (v_3, v_4) are (x_1, v_6). Using v_5 for x_1 and call *unify*1(v_5, v_6). Since (v_5, v_6) have the same constant label, the search backtracks after doing $up(v_6) := v_5$. Now, all children of (v_3, v_4) are done. After $up(v_4) := v_3$, the search backtracks. Similarly, all children of (v1, v2) are done, and we backtrack after doing $up(v_2) := v_1$. Finally, all children of (t, s) are done and *unify*1(t, s) finishes with $up(s) := t$.

The content of $up(v)$ after the termination of *unify*1(t, s) is shown below:

node	s	t	v_1	v_2	v_3	v_4	v_5	v_6	x_1	x_2	x_3
up	t	t	v_1	v_1	v_3	v_3	v_5	v_5	v_5	v_3	v_1

The equivalence relation E generated by $(v, up(v))$ can be represented by the following partition of V:

$$\{s, t\}, \{v_1, v_2, x_3\}, \{v_3, v_4, x_2\}, \{v_5, v_6, x_1\}$$

□

It is clear from the above example that each node of V is in a singleton set initially. When the terms represented by the two nodes are unifiable, the sets containing the two nodes are merged into one thorough the assignment $up(v) := u$, i.e., we obtain "union" of the two sets containing u and v, respectively. To know which node is represented by which node, we need the "find" operation. Thus, we may employ the well-known union/find data structure to support the implementation of *unify*1.

In view of the union/find data structure, each disjoint set is represented by a tree: $up(v) = u$ means the parent of v is u in the tree if $v \neq u$. If $up(u) = u$, then u is the root of the tree, which also serves as the name of the set. $Find(u)$ will find the name of the set containing u and will compress the tree if necessary. $Union(u, v)$ will merge the two sets whose names are u and v, respectively. A typical implementation of $Union$ will let the root of shorter/smaller tree point to the root of a taller/larger tree and can be done in constant time.

As stated earlier, the algorithm *unify*1 takes the two nodes representing s and t, respectively, and visits G in tandem by depth-first search. *unify*1 will first replace the current two nodes by their set names, which are also the nodes in G. If the two set names are the same, backtrack; If one of them is a variable node, make the other as its parent in the union/find tree, and backtrack. If both are function nodes and their labels are different, report *clash failure*; otherwise, recursively visit their children. If no failure is found from the children, merge the two sets named by the current two nodes and backtrack. During the depth-first search, we also use a marking mechanism to detect if a cycle is created because a variable is unified with a term; if yes, an occur-check failure happened. Only a portion of occur-check failures can be found this way. The pseudo-code of the *unify* algorithm is given below.

Algorithm 6.1.19. The algorithm *lunify*(s, t) will take terms s and t, as input, create the formula graph $G = (V, E)$ from s and t, merging all the nodes of the same variable into one, and call *unify*$1(v_s, v_t)$, where v_s and v_t are the nodes representing s and t, respectively. If *unify*$1(v_s, v_t)$ return true, call *postOccurCheck*(v_s, v_t), which returns true iff there exist no cycles in $G = (V, E)$ by the depth-first search.

proc *lunify*(s, t): Boolean
1 $G =$ *createGraph*(s, t); // $up(v) = v$ for each v in G
2 **if** *unify*$1(v_s, v_t)$ // v_s, v_t represent s, t in G
3 **return** *postOccurCheck*(v_s, v_t)
4 **else return** false;

```
proc unify1(v1, v2) : Boolean
1     s1 := Find(v1); s2 := Find(v2);
2     if s1 = s2 return true; // already unified
3     if (label(s1) and label(s2) are variables) Union(s1, s2); return true;
4     if (label(s1) is variable) up(s1) := s2; return true;
5     if (label(s2) is variable) up(s2) := s1; return true;
6     if label(s1) ≠ label(s2) return false // clash failure
      // use mark(s) to detect if G has a cycle
7     for i := 1 to arity(label(s1)) do
8       c1 := child(s1, i); c2 := child(s2, i);
9       if (marked(c1) or marked(c2)) return false // occur-failure
10      mark(c1); mark(c2);
11      if unify1(c1, c2) = false return false;
12      unmark(c1); unmark(c2);
13    Union(s1, s2);
14    return true;

proc Union(s1, s2)
      // Initially, for any node s, h(s) = 0, the height of a singleton tree.
1     if h(s1) < h(s2) up(s1) := s2; return ;
2     up(s2) := s1; // the parent of s2 is s1
3     if h(s1) = h(s2) h(s1) := h(s1) + 1;

proc Find(v)
1     r := up(v); // look for the root of the tree containing v
2     while (r ≠ up(r)) r := up(r); // r is root iff r = up(r)
3     if (r ≠ v) // compress the path from v to r
4       while (r ≠ v) up(v) := r; v := up(v);
5     return r;
```

As said earlier, in the union-find data structure, each set of the nodes is represented by a tree (defined by $up(v) = u$, where the parent of v is u) and the root of the tree is the name of the set. The union operation implements the "union by height", making the root of the higher tree as the parent of the root of the shorter tree. $Find(v)$ will return the name of the set, i.e., the root of the tree containing v; this is done by the first while loop. For the efficiency of future calls to $Find$, the path from v to the root is compressed after the root is found; this is completed by the second while loop.

There are two parent-child relations used in the above algorithms. The procedure *unify*1 uses the depth-first search based on the parent-child relation defined by the formula graph, not the parent-child relation defined by $up(v) = u$. The later induces an equivalence relation and is used by $Find$ and $Union$.

Example 6.1.20. Let $s = f(x, x)$ and $t = f(y, a)$. A graph $G = (V, E)$ will be created with $V = \{v_s, v_t, v_x, v_y, v_a\}$ and $E = \{(v_s, v_x)_1, (v_s, v_x)_2, (v_t, v_y)_1, (v_t, v_a)_2\}$, where the subscripts are the order of children. *unify*1(v_s, v_t) will call *unify*1(v_x, v_y), which changes $up(v_x) = v_y$, and *unify*(v_x, v_y), which changes $up(v_y) = v_a$. After the two recursive calls, *unify*1(v_s, v_t) will change $up(v_s) = v_t$ and return true. V is partitioned into two sets: $\{v_s, v_t\}$ and $\{v_x, v_y, v_a\}$. □

In algorithm *unify*1, the graph $G = (V, D)$ never changes; only $up(v)$ changed. The value of $up(v)$ defines the equivalence relation in the union/find data structure: v and $up(v)$ are in the same disjoint set. The value of $up(v)$ is changed indirectly either by $Find$ (called at line 1 of *unify*1; $up(v)$ is changed at line 4 of $Find$), or by $Union$ (called at lines 3 and 13; $up(v)$ is modified at lines 1 and 2 of $Union$), or directly at lines 4 and 5 of *unify*1, which are special cases of $Union$: a variable node's parent is set to be a function node (not the other direction). Thus, the algorithm uses only linear space. The algorithm uses almost linear time because of the following observations:

- Let $G = (V, E)$ be the term graph of s and t, and $n = |V|$ is bound by the total size of s and t.
- The number of find operations performed in *unify*1 is twice of the number of recursive calls of *unify*1.
- Each union operation reduces the number of node sets by one, thus the number of union operations is bound by n.
- Once two subterms are shown to be unifiable, the two nodes representing them are put into the same set, thus we never try to unify again the same pair of subterms and the number of recursive calls of *unify*1 is bound n.
- The total cost of $O(n)$ finds and unions is $O(n\alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann function and grows very, very slow. In fact, $\alpha(n)$ is practically a constant function, e.g., $\alpha(10^9) \leq 4$.
- The special unions performed at lines 4 and 5 of *unify*1 do not affect the almost linear complexity, because the total cost of such unions between a variable node and a non-variable is bounded by the number of variable nodes.

- The procedure *postOccurCheck* will check if $G = (V, E)$ contains a cycle and can be implemented in $O(n)$ by the depth-first search.

Theorem 6.1.21. *The algorithm* lunify(s, t) *takes* $O(n\alpha(n))$ *time, where* n *is the total sizes of* s *and* t*, and* $\alpha(n)$ *is the inverse of Ackermann function;* lunify(s, t) *returns* `true` *iff* s *and* t *are unifiable.*

The algorithm *lunify* tells us if two terms expressions are unifiable or not. However, this is different from knowing what their unifiers actually are. We leave the problem of finding the mgu of s and t after *lunify*$(s,)$ returns true as an exercise.

Since *lunify* does not change the term graph of s and t, we may wonder how to get the term $t\sigma$ from G if σ is the mgu of s and t. The term $t\sigma$, which is equal to $s\sigma$, can be obtained by $term'(v_s)$ (or $term'(v_t)$), where $term'$ can be recursively defined as follows: For any node v of G:

1. If $label(v)$ is a variable and $up(v) \neq v$, $term'(v) = term'(up(v))$;
2. If $label(v)$ is a constant or a variable and $up(v) = v$, $term'(v) = label(v)$;
3. If $label(v)$ is a function f other than a constant, $term'(v) = f(s_1, s_2, ..., s_k)$, where $s_i = term'(child(v, i))$ for $1 \leq i \leq arity(f)$.

It is easy to check that $term'(v) = term(v)$ before the execution of *lunify* as at that time, $up(v) = v$ for any v. We may view $\{(v, up(v)) \mid up(v) \neq v\}$ as a new set of edges added to $G = (V, E)$ during the execution of *lunify*. This set of edges defines not only the equivalence relation represented by the union/find data structure, but also the mgu σ when s and t are unifiable.

The contrived examples in Table 6.1.1 illustrate behavior of *lunify*. The size of a term is the total number of symbols in the term, excluding parentheses and commas. The column **visited** gives the number of nodes visited before a result is returned.

In P1, the two terms are unifiable, and the occur-check can be awfully expensive if we do not postpone the check. In P2, the first occur-check (on $z \leftharpoondown g(z)$) will fail; though *lunify* will not know it until `postOccurCheck` is called. In P3, the last occur-check will fail. It is interesting to notice that *unify*1 will report this failure by the marking mechanism in a constant number of steps, after visiting only 8 nodes of the term graph. In P4, a clash-failure takes place at the bottom of the depth-first-search. P5 is similar to P3, though an occur-check failure can be found only after a thorough search. P6 is similar to P1, though the last occur-check will locate the failure.

	pair of terms	size	failure	visited
P1	$f(f(...f(f(a,x_1),x_2),...),x_n)$ $f(x_n,f(x_{n-1},f(...,f(x_1,a)...)))$	$4n+2$	no	$4n+2$
P2	$f(z,g^n(x))$ $f(g(z),g^n(y))$	$n+3$	occur	$n+3$
P3	$h(x,f(x,f(x,...,f(x,f(x,x))...)),x)$ $h(f(y,...,f(y,f(y,y))...),y,y)$	$8n+3$	occur	8
P4	$f(f(...f(f(a,x_1),x_2),...),x_n)$ $f(x_n,f(x_{n-1},f(...,f(x_1,b)...)))$	$4n+2$	clash	$4n+2$
P5	$h(x,g^n(x),x)$ $h(g^n(y),y,y)$	$2n+8$	occur	$2n+8$
P6	$f(f(f(...f(f(a,x_1),x_2),...),x_n),z)$ $f(f(x_n,f(x_{n-1},f(...,f(x_1,a)...))),g(z))$	$4n+7$	occur	$4n+7$

Table 6.1.1: The last column gives the number of visited nodes by *unify*1 in the term graph for each example.

6.2 Resolution

6.2.1 The Resolution and Factoring Rules

Definition 6.2.1. (**resolution**) *Suppose clause C_1 is $(A \mid \alpha)$ and C_2 is $(\neg B \mid \beta)$, where A and B are atoms, α and β are the rest literals in C_1 and C_2, respectively,* resolution *is the following inference rule:*

$$\frac{(A \mid \alpha) \qquad (\neg B \mid \beta)}{(\alpha \mid \beta)\sigma}$$

where σ is the mgu of A and B. The clause $(\alpha \mid \beta)\sigma$ produced by the resolution rule is called resolvent *of the resolution; C_1 and C_2 are the* parents *of the resolvent. We use $resolve(C_1, C_2)$ to denote the clause $(\alpha \mid \beta)\sigma$.*

This resolution rule is also called *binary resolution*, as it involves two clauses as premises.

Example 6.2.2. Given a pair of clauses $(p(f(x_1)) \mid q(x_1))$ and $(\overline{p(x_2)} \mid \overline{q(g(x_2))})$. The resolution on p will generate the resolvent $(q(x_1) \mid \overline{q(g(f(x_1)))})$ with the mgu $[x_2 \leftharpoonup f(x_1)]$; the resolution on q will generate the resolvent $(p(f(g(x_2))) \mid \overline{p(x_2)})$ with the mgu $[x_1 \leftharpoonup g(x_2)]$. □

Proposition 6.2.3. *The resolution rule is sound, that is,* $C_1 \wedge C_2 \models resolve(C_1, C_2)$.

Proof. Suppose C_1 is $(A \mid \alpha)$ and C_2 is $(\neg B \mid \beta)$, A and B are unifiable and their mgu is σ. Then $resolve(C_1, C_2) = (\alpha, \beta)\sigma$. For any model I of $C_1 \wedge C_2$, I is also a model of $C_1\sigma = A\sigma \vee \alpha\sigma$ and $C_2\sigma = \neg B\sigma \vee \beta\sigma$, because free variables are treated as universally quantified variables. Since σ is a unifier of A and B, $\alpha\sigma = \beta\sigma$. If $A\sigma$ is false in I, then $\alpha\sigma$ must be true in I, otherwise $C_1\sigma$ will be false in I; If $A\sigma$ is true in I, then $\neg B\sigma$ is false in I, so $\beta\sigma$ must be true in I. In both cases, either $\alpha\sigma$ or $\beta\sigma$ is true in I, so $(\alpha \vee \beta)\sigma$ is true in I. □

By the above proposition, every resolvent is a logical consequence of the input clauses, because the entailment relation is transitive. If the empty clause is generated, then we know that the input clauses are unsatisfiable. In other words, we may use the resolution rule to design a refutation prover. Suppose we have some axioms, A, and we want to see if some conjecture, B, is a logical consequence of A. By the refutational strategy, we want to show that $A \wedge \neg B$ is unsatisfiable, by putting it into clausal form and finding a contradiction through resolution. We wish the following properties hold.

- B is a logical consequence of A, i.e., $A \models B$, iff
- $A \wedge \neg B$ is unsatisfiable, iff
- S, the clause set derived from $A \wedge \neg B$, is unsatisfiable, iff
- The empty clause can be generated by resolution from S.

Unfortunately, the last "iff" does not hold in first-order logic. Indeed, when S is satisfiable, resolution will never generate the empty clause. If the empty clause can be generated, we can claim that S is unsatisfiable, because resolution is sound. However, when S is unsatisfiable, resolution may not generate the empty clause.

Example 6.2.4. Let $S = \{(p(x_1) \mid p(y_1)), (\overline{p(x_2)} \mid \overline{p(y_2)})\}$. From the first clause, we have an instance of $(p(a) \mid p(a))$, which can be simplified to $(p(a))$; from the second clause, we have an instance $(\overline{p(a)} \mid \overline{p(a)})$, which is simplified to $(\overline{p(a)})$. A resolution between $(p(a))$ and $(\overline{p(a)})$ will generate the empty clause. This shows that S is unsatisfiable.

One resolvent of the two clauses in S is $p(x_3) \mid \overline{p(y_3)})\}$, and we may add it into S. Any other resolvent from S will be a renaming of the three clauses. In other words, resolution alone cannot generate the empty clause from S. □

In propositional logic, we defined a proof system which uses the resolution rule as the only single inference rule and showed that this proof system is a decision procedure for propositional logic. In first-order logic, we do not have such a luck.

To get rid of the problem illustrated in the above example, we need the following inference rule:

Definition 6.2.5. (**factoring**) *Let clause C be $(A \mid B \mid \alpha)$, where A and B are literals and α are the rest literals in C.* factoring *is the following inference rule:*

$$\frac{(A \mid B \mid \alpha)}{(A \mid \alpha)\sigma}$$

where σ is the mgu of A and B.

For the clause $(p(x_1) \mid p(y_1))$ in the previous example, factoring will generate $(p(x_1))$; for $(\overline{p(x_2)} \mid \overline{p(y_2)})$, factoring will generate $(\overline{p(x_2)})$. Now resolution will generate the empty clause from $(p(x_1))$ and $(\overline{p(x_2)})$.

Proposition 6.2.6. *The factoring rule is sound, that is, $C \models C'$, where C' is the clause generated by factoring from C.*

Proof. C' is an instance of C, i.e., $C' = C\sigma$, where σ is the unifier used in factoring. Any instance of a formula is a logical consequence of the formula. □

Armed with resolution and factoring, now we can claim the following result without a proof.

Theorem 6.2.7. (**refutational completeness of resolution**) *A set S of clauses is unsatisfiable iff the empty clause can be generated by resolution and factoring.*

Though factoring is needed for the completeness of resolution, in practice, factoring is rarely needed.

6.2.2 A Refutational Proof Procedure

For propositional logic, we introduced three deletion strategies for deleting clauses without worrying about missing a proof:

- **pure literal deletion**: Clauses containing pure literals are discarded.
- **tautology deletion**: Tautology clauses are discarded.
- **subsumption deletion**: Subsumed clauses are discarded.

The pure literal deletion and tautology deletion can be used without modification. For the subsumption deletion, the rule is modified as follows:

Definition 6.2.8. (**subsumption**) *Clause C_1 subsumes clause C_2 if there exists a substitution σ such that every literal of $C_1\sigma$ appears in C_2.*

For example, $(p(x,y) \mid \overline{p(y,x)})$ subsumes $(p(a,b) \mid \overline{p(b,a)} \mid q(a))$ with $\sigma = [x \leftharpoonup a, y \leftharpoonup b]$.

The above deletion strategies can be integrated into the algorithm *resolution* as follows.

Procedure 6.2.9. The procedure *resolution*(C) takes a set C of clauses and returns false iff C is unsatisfiable. It uses *preprocessing*(C) to simplify the input clauses C. The procedure *subsumedBy*(A, S) checks if clause A is subsumed by a clause in S. The procedure *factoring*(A) takes clause A as input and generates the set of results from the factoring rule on A, including A itself.

```
proc resolution(C)
1     G :=preprocessing(C); // G: given clauses
2     K := ∅; // K: kept clauses
3     while G ≠ ∅ do
4         A := pickClause(G); // heuristic for picking a clause
5         G := G − {A};
6         N := ∅; // new clauses from A and K by resolution
7         for B ∈ K if resolvable(A, B) do
8             D  := resolve(A, B);
9             if D = ( ) return false; // the empty clause is found
10            if subsumedBy(D, G ∪ K) continue;
11            N := N ∪ factoring(D);
12        K := K ∪ {A};
13        for A ∈ G if subsumedBy(A, N) do G := G − {A};
14        for B ∈ K if subsumedBy(B, N) do K := K − {B};
15        G := G ∪ N;
16    return true; // K is saturated by resolution
```

The above procedure is almost identical to Algorithm 3.3.26 with two differences: (*a*) *factoring* is used (line 11); (*b*) the termination of the procedure is not guaranteed as it may generate an infinite number of new clauses. Thus, it is not an algorithm. Despite of this, this procedure is a base for many resolution

theorem provers, including McCune's Otter and its successor Prover9, which will be introduced shortly. In Prover9, the subsumption check at line 10 is called *forward subsumption* and the subsumption checks at lines 13 and 14 are called *back subsumption*.

Prover9 also uses another simplification rule called *unit deletion*.

Definition 6.2.10. (**unit deletion**) *Unit deletion is the simplification rule that deletes literal B from clause C if there exists a unit clause (A) and a substitution σ such $A\sigma = \neg B$ (or $\neg A\sigma = B$).*

Unit deletion is sound because by the requirement, there is a resolution between (A) and C and the resolvent of this resolution will subsume C. For example, if we have a unit clause $(\overline{p(x,b)})$, then the literal $p(a,b)$ in $(p(a,b) \mid q(a))$ can be deleted to get a new clause $(q(a))$, which subsumes $(p(a,b) \mid q(a))$.

Unit deletion can be used at line 10, so that the new clause D is simplified by the unit clauses in $G \cup K$. This is called *forward unit deletion*. *Back unit deletion* can be inserted at lines 13 and 14 when the unit clauses in N are used to simplify clauses in $G \cup K$.

6.3 Ordered Resolution

In section 3.3.2, we listed some well-known resolution strategies which add restrictions to the use of the resolution rule: unit resolution, input resolution, ordered resolution, positive resolution, negative resolution, set of support, and linear resolution. They can be carried over to first-order logic without modification, with the exception of ordered resolution. The procedure *resolvable*(A, B) (line 7 of *resolution*) is the place where various restrictions are implemented so we can use these restricted resolution strategies.

For unit resolution and input resolution, they are still incomplete in general, but are equivalent in the sense that there is a resolution proof for one strategy, then there exists a resolution proof for the other strategy.

6.3.1 Simplification Orders

To use ordered resolution in first-order logic, we need a partial order to compare atoms. We call this order as "term order", although atoms, literals, and clauses are not terms, the term orders we present here apply to those objects as well.

There are several requirements on the term orders used for ordered resolution and we term it as "simplification order".

Definition 6.3.1. (**stable and monotonic relation**) *A binary relation R over $T(F, X)$ is said to be* stable *if for any term s and t, $R(s, t)$ implies $R(s\sigma, t\sigma)$ for any substitution $\sigma : X \longrightarrow T(F, X)$. R is said to be* monotonic *if $R(s, t)$ implies $R(f(..., s, ...), f(..., t, ...))$ for any function or predicate symbol f.*

For a stable order $\succ$, we cannot have $f(y) \succ x$ because $f(y)\sigma \succ x\sigma$ does not hold for $\sigma = [x \leftarrow f(y)]$.

Definition 6.3.2. (**simplification order**) *The order $\succ$ is called a* simplification order *if it is well-founded, stable and monotonic. We write $s \prec t$ if $t \succ s$; $s \succeq t$ if $s \succ t$ or $s \approx t$ (i.e., s and t are equivalent in the order).*

The reason we stick to well-founded orders because well-founded orders allow us to use mathematical induction. Let S be a set with a well-found order $\succ$. To show a property $P(x)$ holds for every $x \in S$, we need to do two things:

- For every minimal element $m \in S$, $P(m)$ is true.
- Prove that $P(x)$ is true, assuming $P(x')$ is true for every x' and $x \succ x'$.

The above is called the *well-founded induction principle*.

There are many well-founded orders over different sets. For example, $>$ over the set of natural numbers is well-founded; $>$ on the set of integers is not well-founded. Any partial order over a finite set is well-founded.

There are many ways to obtain well-found orders. A *multiset* or *bag* over a set S of elements is a modification of the concept of a subset of S, that, unlike a set, allows for multiple instances for each of its elements from S. The union, intersection, and subtraction operations can be extended over multisets. For example, let $S_1 = \{a, b, c\}$ and $S_2 = \{a, a, b, b\}$, then $S_1 \cup S_2 = \{a, a, a, b, b, b, c, c\}$, $S_1 \cap S_2 = \{a, b, c\}$, and $S_1 - S_2 = \{a\}$. If there exists an order over the elements of S, we may extend this order to compare the multisets of S.

Proposition 6.3.3. *Let $\succ$ be a well-founded order over a set S. The order $\succ^{mul}$ over the multisets of S is well-founded, where, for any multisets S_1, S_2 over S, $S_1 \succ^{mul} S_2$ iff for any $y \in S_2 - S_1$, there exists $x \in S_1 - S_2$, $x \succ y$.*

For example, if $c \succ b \succ a$, then $S_1 \succ^{mul} S_2$ for $S_1 = \{a, b, b, c\}$ and $S_2 = \{a, a, b, c\}$, because $S_1 - S_2 = \{b\}$, $S_2 - S_1 = \{a\}$, and $b \succ a$.

One popular way is to use lexicographic combination from well-found orders of simpler sets.

Proposition 6.3.4. *Let $\succ_i$ be a well-founded order over the set S_i, $1 < i \leq n$. The lexicographic order $\succ^{lex}$ over $S_1 \times S_2 \times \cdots \times S_n$ is well-founded, where for any $\langle x_1, x_2, ..., x_n\rangle, \langle y_1, y_2, ..., y_n\rangle \in S_1 \times S_2 \times \cdots \times S_n$, $\langle x_1, x_2, ..., x_n\rangle \succ^{lex} \langle y_1, y_2, ..., y_n\rangle$ iff there exists k, $1 \leq k \leq n$, such that $x_i \succeq_i y_i$ for $1 \leq i < k$ and $x_k \succ_k y_k$.*

The lexicographic combination can be extended to compare two sequences of different lengths, assuming a non-empty sequence is greater than the empty sequence, like the dictionary order.

Given a first-order language $L = (P, F, X, Op)$, let $\Sigma = P \cup F \cup Op$. We say $\succ$ is a *precedence* if $\succ$ is a well-founded total order over Σ. Note that if Σ is finite, $\succ$ is trivially well-founded. For any term $t \in T(F, X)$, let $var(t)$ denote the set of variables appearing in t. We may extend any precedence to terms, as well as to atoms and literals, by several different ways.

- LPO (Lexicographic Path Order). The term order is determined entirely by the symbol precedence.
- RPO (Recursive Path Order). The term order is determined entirely by the symbol precedence.
- KBO (Knuth-Bendix Order). This order uses a weighting function on symbols as well as the symbol precedence. The weighting function is used first, and the symbol precedence breaks ties.

The formal definitions of LPO, RPO, and KBO are here:

Definition 6.3.5. ($\succ_{lpo}$) *The* lexicographical path order $\succ_{lpo}$ *over $T(F, X)$ is defined recursively as follows: $s \succ_{lpo} t$ if $s \neq t$ and either $t \in var(s)$ or $s = f(s_1, ..., s_m) \succ_{lpo} t = g(t_1, ..., t_n)$ by one of the following conditions:*

1. *$f \succ g$ and $s \succ_{lpo} t_i$ for $1 \leq i \leq n$, or*
2. *$f \approx g$, $[s_1, ..., s_m] \succ^{lex}_{lpo} [t_1, ..., t_n]$, and $s \succ_{lpo} t_i$ for $1 \leq i \leq n$, or*
3. *$f \prec g$ and there exists j, $1 \leq j \leq m$, $s_j \succeq_{lpo} t$.*

Example 6.3.6. Let $* \succ +$, s be $x * (y + z)$ and t be $(x * y) + (x * z)$. Then $s \succ_{lpo} t$ by condition 1, because $* \succ +$, $s \succ_{lpo} x * y, x * z$, both by condition 2. That is, from $[x, (y + z)] \succ^{lex}_{lpo} [x, y], [x, z]$, we have $s \succ_{lpo} x * y, x * z$. □

Proposition 6.3.7. *$\succ_{lpo}$ is a simplification order and is total on ground terms, if the precedence $\succ$ is a well-founded total order.*

Definition 6.3.8. ($\succ_{rpo}$) *The* recursive path order $\succ_{rpo}$ *over* $T(F,X)$ *is defined recursively as follows:* $s \succ_{rpo} t$ *if* $s \neq t$ *and either* $t \in var(s)$ *or* $s = f(s_1,...,s_m) \succ_{rpo} t = g(t_1,...,t_n)$ *by one of the following conditions:*

1. $f \succ g$ *and* $s \succ_{rpo} t_i$ *for* $1 \leq i \leq n$, *or*
2. $f \approx g$ *and* $\{s_1,...,s_m\} \succ_{rpo}^{mul} \{t_1,...,t_n\}$, *or*
3. $f \prec g$ *and there exists* j, $1 \leq j \leq m$, $s_j \succeq_{rpo} t$.

Proposition 6.3.9. $\succ_{rpo}$ *is a simplification order, if the precedence* $\succ$ *is a well-founded total order.*

Example 6.3.10. Let $s = (x * y) * z$ and $t = x * (y * z)$, then $s \succ_{lpo} t$ because of condition 2: $[x * y, z] \succ_{lpo}^{lex} [x, y * z]$. However, $\succ_{rpo}$ cannot compare these two terms, because $\{x * y, z\}$ and $\{x, y * z\}$ are not comparable by $\succ_{rpo}^{mul}$. Moreover, $\succ_{rpo}$ treats $(a * a) * a$ and $a * (a * a)$ as "equal terms". □

By definition, a partial order is an antisymmetry, transitive, and reflexive relation. Since $\succ_{rpo}$ violates the antisymmetry condition, it is sometimes called "quasi-order". We can easily modify $\succ_{rpo}$ so that when $\{s_1,...,s_m\} \approx_{rpo} \{t_1,...,t_n\}$, we may compare them using $\succ_{rpo}^{lex}$, so that $\succ_{rpo}$ can be a total order on ground terms.

Because $\succ_{rpo}$ cannot compare $s = x*(y*z)$ and $t = (x*y)*z$, and is neither a partial order nor a total order on ground terms in the strict sense, it is less popular than $\succ_{lpo}$ in practice.

The *Knuth-Bendix order* ($\succ_{kbo}$) uses the weights first to compare terms. When two terms have the same weight, the precedence relation is used to break ties. Using the weights gives us flexibility as well as complexity.

Definition 6.3.11. (**weight function**) *A function* $w : X \cup \Sigma \longrightarrow N$ *is said be a* weight function *if it satisfies (i)* $w(c) > 0$ *for every constant* $c \in \Sigma$, *and (ii) there exists at most one unary function symbol* $f \in \Sigma$ *such that* $w(f) = 0$ *and* f *must maximal in the precedence if* $w(f) = 0$.

The weight function w can be extended a function over terms or formulas as follows:

$$\begin{aligned} w(x) &= w_0 \text{ for any variable } x; \\ w(f(t_1,...,t_n)) &= w(f) + w(t_1) + \cdots + w(t_n) \end{aligned}$$

where w_0 denotes the minimal weight of all constants.

Definition 6.3.12. ($\succ_{kbo}$) *Let w be a weight function and $mv(t)$ be the multiset of variables appearing in t. The* Knuth–Bendix order $\succ_{kbo}$ *over $T(F,X)$ defined recursively as follows: $s \succeq_{kbo} t$ if $w(s) \geq w(t)$ and $mv(s) \supseteq mv(t)$; $s \succ_{kbo} t$ if $s \succeq_{kbo}$ and (a) $w(s) > w(t)$ or (b) $s = f(s_1, ..., s_m)$, $t = g(t_1, ..., t_n)$, and one of the following conditions is true:*

1. *$f \succ g$, or*
2. *$f \approx g$ and $[s_1, ..., s_m] \succ^{lex}_{kbo} [t_1, ..., t_n]$, or*
3. *$f \prec g$ and there exists j, $1 \leq j \leq m$, $s_j \succeq_{kbo} t$.*

We like to point out that the condition $mv(s) \supseteq mv(t)$ is necessary. Let $s = f(g(x), y)$ and $t = f(y, y)$. If $w(g) > 0$, then $w(s) > w(t)$, but $mv(s) = \{x, y\} \supseteq \{y, y\} = mv(t)$ does not hold. If we ignore the condition $mv(s) \supseteq mv(t)$, then $s \succ_{kbo} t$ would imply $s\theta \succ_{kbo} t\theta$ for $\theta = [y \leftarrow f(x)]$, where $s\theta = t\theta = f(g(x), g(x))$.

Example 6.3.13. Let $w(f) = w(a) = 1$, $w(b) = 2$, then $f(x, b, a) \succ_{kbo} f(a, a, b)$ by condition $(b.2)$, because $x \succeq_{kbo} a$ and $b \succ_{kbo} a$. Note that neither $\succ_{lpo}$ nor $\succ_{rpo}$ can compare $p(x, b, y)$ and $p(a, a, y)$. □

Proposition 6.3.14. *$\succ_{kbo}$ is a simplification order and is total on ground terms, if the precedence $\succ$ is a well-founded total order.*

Example 6.3.15. Let the precedence relation be $b/1 \succ a/1 \succ e/0$, where a and b are unary function symbols and e is a constant. We will write $a(a(b(a(e))))$ as $aaba$ for brevity. Let $w(e) = 1$ and the weights of a and b, as well as the least twelve terms (ignoring e), with the exception of the last row, are listed (in the order of $\prec_{kbo}$) in the following table:

$w(a)$	$w(b)$	the least 12 terms in the order of $\prec_{kbo}$
1	1	$a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, ...$
2	1	$b, a, bb, ab, ba, bbb, aa, abb, bab, bba, bbbb, abb, ...$
2	3	$a, b, aa, ab, ba, aaa, bb, aab, aba, baa, aaaa, abb, ...$
2	0	$b, bb, ..., b^i, ..., a, ab, ..., ab^i, ..., ba, bab, babb, ...$

On the other hand, the least terms under $\succ_{lpo}$ or $\succ_{rpo}$ are

$$a, ..., a^i, ..., b, ab, ..., a^ib, ..., ba, aba, ..., a^iba, ..., baa, abaa, ..., a^ibaa, ...$$

□

The above example shows the flexibility of $\succ_{kbo}$, which provides various ways of comparing two terms by changing weight functions as needed. However, $\succ_{kbo}$ lacks the flexibility of comparing terms like $x * (y + z)$ and $x * y + x * z$. We cannot make $x * (y + z) \succ_{kbo} x * y + x * z$, while $x * (y + z) \succ_{lpo} x * y + x * z$ if $* \succ +$ and $x * y + x * z \succ_{lpo} x * (y + z)$ if $+ \succ *$.

Now when we say to use a simplification order $\succ$, we could use either $\succ_{lpo}$, $\succ_{rpo}$, or $\succ_{kbo}$.

Example 6.3.16. Let the operators $\leftrightarrow, \oplus, \rightarrow, \neg, \vee, \wedge$ be listed in the descending order of the precedence $\succ$, then all the right sides of the following equivalence relations are less than their left sides by either $\succ_{lpo}$ or $\succ_{rpo}$.

$$\begin{aligned} A \oplus B &\equiv (A \vee B) \wedge (\neg A \vee \neg B); \\ A \leftrightarrow B &\equiv (A \vee \neg B) \wedge (\neg A \vee B); \\ A \rightarrow B &\equiv \neg A \vee B; \\ \neg\neg A &\equiv A; \\ \neg(A \vee B) &\equiv \neg A \wedge \neg B; \\ \neg(A \wedge B) &\equiv \neg A \vee \neg B; \\ A \vee (B \wedge C) &\equiv (A \vee B) \wedge (A \vee C). \end{aligned}$$

These equivalence relations are used to translate a formula into CNF. Thus, the term orderings like $\succ_{rpo}$ and $\succ_{lpo}$ can be used to show the termination of the process of converting a formula into CNF. □

If we want to convert formulas into DNF, then the precedence relation should be $\wedge \succ \vee$, so that the termination of converting a formula into DNF can be shown by $\succ_{rpo}$ and $\succ_{lpo}$.

6.3.2 Completeness of Ordered Resolution

Let $\succ$ be a simplification order, which is a well-found, stable partial order over the set of atoms and total over the set of ground atoms, we will use $\succ$ in ordered resolution. Given $\succ$, an atom $A \in S$ is said to be *maximal* if there is no $B \in S$ such that $B \succ A$. A is said to be *minimal* if there is no $B \in S$ such that $A \succ B$. S may have several minimal or maximal atoms. In fact, an atom A can be both minimal and maximal in S if A cannot be compared by $\succ$ with any other atom of S. Literals are compared by comparing their atoms. Since a clause is represented by a set of literals, we may say a literal is maximal or minimal in a clause. For example, the three atoms in $(\overline{p(x, y)} \mid \overline{p(y, z)} \mid p(x, z))$ are not comparable under any stable $\succ$, so each atom is both minimal and maximal.

Definition 6.3.17. (**ordered resolution**) *Given a simplification order $\succ$ over the set of atoms, the ordered resolution is the resolution with the condition that A is a maximal atom in α and B is a maximal atom in β:*

$$\frac{(A \mid \alpha) \qquad (\neg B \mid \beta)}{(\alpha \mid \beta)\sigma}$$

Given a set S of clauses, let S^* denote the set of clauses *saturated* by ordered resolution and factoring. That is, any resolvent from ordered resolution on any two clauses of S^* is in S^* and any clause from factoring on any clause in S^* is also in S^*. Let GC be the set of all ground instances of S^*.

Lemma 6.3.18. *GC is saturated by ordered resolution.*

Proof. Suppose $(g \mid \alpha)$ and $(\overline{g} \mid \beta)$ are in GC, where g is the maximal atom in both clauses. The resolvent of the two clauses is $(\alpha \mid \beta)$. We need to show that $(\alpha \mid \beta) \in GC$.

Let $(g \mid \alpha)$ be an instance of $(A \mid \alpha') \in S^*$, i.e., $g = A\theta$ and $\alpha = \alpha'\theta$ for some substitution θ. We assume that A is the only literal in $(A \mid \alpha')$ such that $A\alpha = g$; otherwise, we use factoring to achieve this condition. Similarly, let $(\overline{g} \mid \beta)$ be an instance of $(\overline{B} \mid \beta') \in S^*$, i.e., $\overline{g} = \overline{B}\gamma$ and $\beta = \beta'\gamma$ for some γ, and B is the only literal in $(\overline{B} \mid \beta')$ such that $B\gamma = g$.

Since $g = A\theta = B\gamma$ is maximal, A and B must be unifiable and maximal in $(A \mid \alpha')$ and $(\overline{B} \mid \beta)$, respectively. Let λ be the mgu of A and B, then the resolvent of $(A \mid \alpha')$ and $(\overline{B} \mid \beta')$ by ordered resolution is $(\alpha' \mid \beta')\lambda$, which must be in S^*. Since $\theta = \lambda\theta'$ and $\gamma = \lambda\gamma'$ for some θ' and γ', we must have $(\alpha \mid \beta) = (\alpha' \mid \beta')\lambda\theta'\gamma'$, which is an instance of $(\alpha' \mid \beta')\lambda$. Hence, $(\alpha \mid \beta) \in GC$. □

Theorem 6.3.19. (**refutational completeness of ordered resolution**) *S is unsatisfiable iff the empty clause is in S^*.*

Proof. If the empty clause is in S^*, since $S \models \bot$, S must be unsatisfiable.

If the empty clause is not in S^*, we will construct a Herbrand model for S from GC, which is the set of all ground instances of S^*. Obviously, the empty clause is not in GC. Let GA be the set of ground atoms appearing in GC. By the assumption, $\succ$ is a well-founded total order over GA.

We start with an empty model H in which no atoms have a truth value. We then add ground literals into H one by one, starting from the minimal atom, which exists because $<$ is well-founded. Let g be any atom in GA, and

$$\begin{aligned} pc(g) &= \{(g \mid \alpha) \mid g \text{ is maximal in } (g \mid \alpha) \in G\} \\ nc(g) &= \{(\overline{g} \mid \beta) \mid g \text{ is maximal in } (\overline{g} \mid \beta) \in G\} \\ upto(g) &= \textstyle\bigcup_{g' \in GA, g' \leq g} (pc(g') \cup nc(g')) \end{aligned}$$

That is, $pc(g)$ contains all the clauses which contains g as its maximal literal; $pc(g)$ contains all the clauses which contains $\overline{g}$ as its maximal literal; and $upto(g)$ contains all the clauses from GC whose maximal literals are less or equal to g.

We claim that after g or $\overline{g}$ is added into H, the following the property is true:

Claim: H is a model of $upto(p)$ after g or $\overline{g}$ is added into H.

Let g_1 be the minimal atom of GA. We cannot have both (g_1) and $(\overline{g_1}) \in GC$; otherwise the empty clause will be generated from (g_1) and $(\overline{g_1})$. By Lemma 6.3.18, the empty clause will appear in S^*, a contradiction. If $(g_1) \in GC$, we add g_1 into H (it means $H(g_1) = 1$); otherwise, we add $\overline{g_1}$ into H (it means $H(g_1) = 0$). It is trivial to check that H is a model of $upto(g_1)$, so the claim is true for g_1. This is the base case of the induction based on $<$.

Let g be the minimal atom in GA which has no truth value in H. Assume as the induction hypothesis that the claim is true for all $g' < g$. If there exists a clause $(g \mid \alpha) \in pc(g)$ such that $H(\alpha) = 0$, we add g into H; otherwise, add $\overline{g}$ into H.

Suppose g is added into H, then every clause in $pc(g)$ will be true under H. If there is exists a clause $(\overline{g} \mid \beta) \in nc(g)$ such that $H(\beta) = 0$, then there exists an ordered resolution between $(g \mid \alpha)$ and $(\overline{g} \mid \beta)$, and their resolvent is $(\alpha \mid \beta)$. By Lemma 6.3.18, $(\alpha \mid \beta) \in GC$. Since every literal g' of $(\alpha \mid \beta)$ is less than g, by the induction hypothesis, $(\alpha \mid \beta)$ is true in H. If $H(\alpha) = 0$, then $H(\beta) = 1$. That means no clause in $nc(g)$ will be false in H.

Now suppose $\overline{g}$ is added into H, the above analysis still holds by changing the role of p and $\overline{p}$. In both cases, all clauses in $pc(g)$ and $nc(g)$ are true in H after adding g or $\overline{g}$ into H. Thus the claim is true for every $g \in GA$. Once every atom in GA are processed, we have found a model H of GC by the claim. Since GC is the set of all ground instances of S^*, H is also a model of S^* or S. In other words, S is satisfiable. □

Note that the model construction in the above proof is not algorithmic, because GC is infinite in general. The induction is sound because it is based on the well-founded order $\succ$.

Now it is easy to prove Theorem 6.2.7: If the empty clause is generated from S, then S must be unsatisfiable; if S is unsatisfiable, then the empty clause will be generated by ordered resolution, which is a special case of resolution.

A traditional proof of Theorem 6.2.7 uses Herbrand's Theorem:

Theorem 6.3.20. (**Herbrand's Theorem**) *A set S of clauses is unsatisfiable iff there exists a finite, unsatisfiable, ground instances of S.*

A proof of Theorem 6.2.7 goes as follows:

1. If S is unsatisfiable, then by Herbrand's Theorem, there exists a finite set G of ground instances of S.

2. Resolution for propositional logic will find a proof from G, treating each ground atom in G as a propositional variable.

3. This ground resolution proof can be lifted to a general resolution proof in S, using a lemma similar to Lemma 6.3.18.

6.4 Prover9: A Resolution Theorem Prover

Prover9 is a full-fledged automated theorem prover for first-order logic based on resolution. Prover9 is the successor of the Otter prover; both Otter and Prover9 are created by William McCune (1953-2011).

6.4.1 Input Formulas to Prover9

Prover9 has a fully automatic mode in which the user simply gives it formulas representing the problem. A good way to learn about Prover9 is to browse and study the example input and output files that are available with the distribution of Prover9. Let us look at an example.

Example 6.4.1. Once we can specify the following puzzle in first-order logic, it is trivial to find a solution by Prover9:

> Jack owns a dog. Every dog owner is an animal lover. No animal lover kills an animal. Either Jack or Curiosity killed the cat, who is named Tuna. Did Curiosity kill the cat?

The following first-order formulas come from the statements of the puzzle:

(1) $\exists x(dog(x) \wedge owns(Jack, x))$
(2) $\forall x(\exists y(dog(y) \wedge owns(x, y)) \rightarrow animalLover(x))$
(3) $\forall x(animalLover(x) \rightarrow (\forall y(animal(y) \rightarrow -kills(x, y))))$
(4) $kills(Jack, Tuna) \vee kills(Curiosity, Tuna)$
(5) $cat(Tuna)$
(6) $\forall x(cat(x) \rightarrow animal(x))$

The goal is to show that $kills(Curiosity, Tuna)$. The above formulas can be written in Prover9's syntax as follows:

```
exists x (dog(x) & owns(Jack, x)).
all x (exists y (dog(y) & owns(x, y)) -> animalLover(x)).
all x (animalLover(x) -> (all y (animal(y) -> -kills(x, y)))).
kills(Jack, Tuna) | kills(Curiosity, Tuna).
cat(Tuna).
all x (cat(x) -> animal(x)).
```

□

From this example, we can see that each formula in Prover9 is ended with ".", and we use "`->`" for $\rightarrow$, "|" for $\vee$, "&" for $\wedge$, "$-$" for $\neg$, "`all`" for $\forall$, and "`exists`" for $\exists$. In fact, the top most "all" is optional as free variables are assumed to universally quantified. Prover9 will convert formulas in the "assumptions" list and the negation of the formulas in the "goals" list into clauses before resolution is called.

Note that Prover9 uses a different precedence for the quantifiers "all" and "exists"; they are higher than other Boolean operators. For instance,

```
exists x dog(x) & owns(Jack, x)
```

represents the formula $(\exists x\, dog(x)) \wedge owns(Jack, x)$, not $\exists x(dog(x) \wedge owns(Jack, x))$ as assumed in this book. To avoid confusions, insert a pair of parentheses when you are uncertain about the scope of a quantified variable.

Prover9 implements the procedure *resolution* (Procedure 6.2.9) with bells and whistles. The given list is called the "sos" (set of support) list and the kept list is called the "usable" list. If you already have a set of clauses ready, you may use the "usable" list and the "sos" list as follows:

```
formulas(usable). % the kept list
...
end_of_list.

formulas(sos). % the given list
...
end_of_list.
```

.

The resolution is always done between a clause from the sos list (the given list) and some clauses from the usable list (the kept list).

A basic Prover9 command on a linux machine will look like

```
prover9 -f Tuna.in > Tuna.out
```

or

```
prover9 < Tuna.in > Tuna.out
```

where the file named "Tuna.in" contains the input to Prover9 and the file named "Tuna.out" contains the output of Prover9. If the file "Tuna.in" contains the formulas in Example 6.4.1, then the file "Tuna.out" will contain the following proof:

```
============================ PROOF =============================

% -------- Comments from original proof --------
% Proof 1 at 0.03 (+ 0.05) seconds.
% Length of proof is 19.
% Level of proof is 6.
% Maximum clause weight is 6.
% Given clauses 0.

1 (exists x (dog(x) & owns(Jack,x))). [assumption].
2 (all x ((exists y (dog(y) & owns(x,y))) -> animalLover(x))).
      [assumption].
3 (all x (animalLover(x) -> (all y (animal(y) -> -kills(x,y))))).
      [assumption].
4 (all x (cat(x) -> animal(x))). [assumption].
5 kills(Curiosity,Tuna).  [goal].
6 -dog(x) | -owns(y,x) | animalLover(y).  [clausify(2)].
7 dog(c1).  [clausify(1)].
8 -owns(x,c1) | animalLover(x).  [resolve(6,a,7,a)].
9 owns(Jack,c1).  [clausify(1)].
10 animalLover(Jack).  [resolve(8,a,9,a)].
11 -animalLover(x) | -animal(y) | -kills(x,y).  [clausify(3)].
12 -cat(x) | animal(x).  [clausify(4)].
13 cat(Tuna).  [assumption].
14 animal(Tuna).  [resolve(12,a,13,a)].
15 -animal(x) | -kills(Jack,x).  [resolve(10,a,11,a)].
16 kills(Jack,Tuna) | kills(Curiosity,Tuna).  [assumption].
17 -kills(Curiosity,Tuna).  [deny(5)].
18 -kills(Jack,Tuna).  [resolve(14,a,15,a)].
19 $F.  [back_unit_del(16),unit_del(a,18),unit_del(b,17)].
```

```
========================== end of proof ======================
```

In Prover9, the empty clause is denoted by `$F`. In each clause, the literals are numbered by a, b, c, ... For example, to generate clause 18, `-kills(Jack, Tuna)`, from

```
14 animal(Tuna).  [resolve(12,a,13,a)].
15 -animal(x) | -kills(Jack,x).  [resolve(10,a,11,a)].
```

Prove9 performed a resolution on the first literal (a) of clause 14 and the first literal (a) of clause 15, with the mgu `x` $\leftharpoonup$ `Tuna`, and the resolvent is

```
18 -kills(Jack,Tuna).  [resolve(14,a,15,a)].
```

The message followed by each clause can be checked by either human or machine, to ensure the correctness of the proof.

6.4.2 Inference Rules and Options

There are two types of parameters in Prover9: Boolean flags and numeric parameters. To change the default values of the former type, use `set(flag)` or `clear(flag)`. The latter can be changed by `assign(parameter, value)`. For example, the default automatic mode is set by

```
set(auto).
```

Turning "auto" on will cause a list of other flags to turn on, including setting up "hyper-resolution" as the main inference rule for resolution. Hyper-resolution reduces the number of intermediate resolvents by combining several resolution steps into a single inference step.

Definition 6.4.2. (**hyper-resolution**) Positive hyper-resolution *consists of a sequence of positive resolutions between one non-positive clause (called* nucleus*) and a set of positive clauses (called* satellites*), until a positive clause or the empty clause is produced.*

The number of positive resolutions in a hyper-resolution is equal to the number of negative literals in the nucleus clause.

Example 6.4.3. Let the nucleus clause be $(\overline{p(x)} \mid \overline{q(x)} \mid r(x))$, and the satellite clauses be $(q(a) \mid r(b))$ and $(p(a) \mid r(c))$. The first resolution between the nucleus

and the first satellite produces $(\overline{p(a)} \mid r(a) \mid r(b))$. The second resolution between the resolvent and the second satellite produces the second resolvent $(r(a) \mid r(b) \mid r(c))$, which is the result of positive hyper-resolution between the nucleus and the satellites. □

Negative hyper-resolution can be defined similarly by replacing "positive" by "negative". Hyper-resolution means both positive hyper-resolution and negative hyper-resolution.

Example 6.4.4. In section 3.2.2, we discussed the Hilbert system which consist of three axioms and one inference rule, i.e., *Modus Ponens*. To prove $x \to x$ in the Hilbert system, we may use the following input: □

```
op(400, infix_right, ["->"]).  % infix operator

formulas(usable).
P((n(y) -> n(x)) -> (x -> y)).
-P(x) | -P(x -> y) | P(y).
end_of_list.

formulas(sos).
P(x -> (y -> x)).
P((x -> (y -> z)) -> ((x -> y) -> (x -> z))).
-P(a -> a).
end_of_list.
```

Prover9 will produce the following proof:

```
1 -P(x) | -P(x -> y) | P(y).  [assumption].
2 P(x -> y -> x).  [assumption].
3 P((x -> y -> z) -> (x -> y) -> x -> z).  [assumption].
4 -P(a -> a).  [assumption].
5 P(x -> y -> z -> y).  [hyper(1,a,2,a,b,2,a)].
7 P((x -> y) -> x -> x).  [hyper(1,a,2,a,b,3,a)].
15 P(x -> x).  [hyper(1,a,5,a,b,7,a)].
16 $F.  [resolve(15,a,4,a)].
```

The hyper-resolution is the default inference rule for the automatic mode. If you like to see a binary resolution proof, you have to turn off "auto" and turn on "binary_resolution":

```
clear(auto).
set(binary_resolution).
```

Turning off `hyper-resolution`, Prover9 will produce the following proof:

```
1 -P(x) | -P(x -> y) | P(y).  [assumption].
2 P(x -> y -> x).  [assumption].
3 P((x -> y -> z) -> (x -> y) -> x -> z).  [assumption].
4 -P(a -> a).  [assumption].
5 -P(x) | P(y -> x).  [resolve(2,a,1,b)].
6 -P(x -> y -> z) | P((x -> y) -> x -> z).  [resolve(3,a,1,b)].
10 P((x -> y) -> x -> x).  [resolve(6,a,2,a)].
19 P(x -> (y -> z) -> y -> y).  [resolve(10,a,5,a)].
20 -P(x -> y) | P(x -> x).  [resolve(10,a,1,b)].
45 P(x -> x).  [resolve(20,a,19,a)].
46 $F.  [resolve(45,a,4,a)].
```

Definition 6.4.5. Unit-resulting resolution *consists of a sequence of unit resolutions between one non-unit clause (called* nucleus*) and a set of unit clauses (called* electrons*), until a unit clause or the empty clause is produced.*

Since positive resolution is complete and unit resolution is not complete, hyper-resolution is complete and unit-resulting resolution is not complete. To use "unit-resulting resolution", use the command

```
set(ur_resolution).
```

To use "hyper-resolution" in Prover9, use the command

```
set(hyper_resolution).
```

This option will cause the flags `pos_hyper_resolution` and `neg_hyper_resolution` to be true.

The resolution inference rules provided by Prover9 include "binary resolution", "hyper resolution", "ordered resolution", and "unit-resulting resolution". To use "ordered resolution", use the command

```
set(ordered_res).
```

This option puts restrictions on the binary and hyper-resolution inference rules. It says that resolved literals in one or more of the parents must be maximal in the clause. Continuing from the previous example, if we turn on the ordered resolution flag, the proof generated by Prover9 will be the following.

```
1 -P(x) | -P(x -> y) | P(y).  [assumption].
2 P(x -> y -> x).  [assumption].
3 P((x -> y -> z) -> (x -> y) -> x -> z).  [assumption].
4 -P(a -> a).  [assumption].
5 -P(x) | P(y -> x).  [resolve(2,a,1,b)].
6 -P(x -> y -> z) | P((x -> y) -> x -> z).  [resolve(3,a,1,b)].
8 P(x -> y -> z -> y).  [resolve(5,a,2,a)].
9 P(x -> y -> z -> u -> z).  [resolve(8,a,5,a)].
10 P(x -> y -> z -> u -> w -> u).  [resolve(9,a,5,a)].
14 P((x -> y) -> x -> x).  [resolve(6,a,2,a)].
19 -P(x -> y) | P(x -> x).  [resolve(14,a,1,b)].
21 P(x -> x).  [resolve(19,a,10,a)].
22 $F.  [resolve(21,a,4,a)].
```

6.4.3 Simplification Orders in Prover9

Prover9 has several methods available for comparing terms or literals. The term orders are partial (and sometimes total on ground terms), and they are used to decide which literals in clauses are admissible for application of ordered resolution. Several of the resolution rules require that some of the literals be maximal in their clause.

The symbol precedence is a total order on function and predicate symbols (including constants). The symbol weighting function maps symbols to non-negative integers. Prover9 supports three term orders: LPO (Lexicographic Path Order), RPO (Recursive Path Order), and KBO (Knuth-Bendix Order), which are introduced in secton 6.3.1.

Here is the command for choosing a simplification order.

```
assign(order, string). % default string=lpo, range [lpo,rpo,kbo]
```

This option is used to select the primary term order to be used for determining maximal literals in clauses. The choices are "lpo" (Lexicographic Path Order), "rpo" (Recursive Path Order), and "kbo" (Knuth-Bendix Order).

The default symbol precedence (for LPO, RPO, and KBO) is given by the following rules (in order).

- function symbols $<$ non-equality predicate symbols;
- for function symbols: $c/0 < f/2 < g/1 < h/3 < i/4 < ...$, where c is any constant, f is any function of arity 2, g is any function of arity 1, ... (note the position of $g/1$);

- for predicate symbols: lower arity < higher arity;
- non-Skolem symbols < Skolem symbols;
- for Skolem symbols, the lower index is the lesser;
- for non-Skolem symbols, more occurrences < fewer occurrences;
- the lexical ASCII order (UNIX strcmp() function).

The `function_order` and `predicate_order` commands can be used to change the default symbol precedence. They contain lists of symbols ordered by increasing precedence. For example,

```
predicate_order([=, <=, P, Q]).          % = < <= < P < Q
function_order([a, b, c, +, *, h, g]).   % a < b < c < + < * < h < g
```

We need two separate commands for defining the precedence, because we assume that predicate symbols are always greater than function symbols in the precedence. The used symbol precedence for a problem is always printed in the output file (in the section PROCESS INPUT).

6.5 Exercise Problems

1. Let $\theta = [x \leftharpoondown u, y \leftharpoondown f(g(u)), z \leftharpoondown f(u)]$ and $\sigma = [u \leftharpoondown x, x \leftharpoondown f(u), y \leftharpoondown a, z \leftharpoondown f(u)]$. Compute $\theta\sigma$ and $\sigma\theta$. Among θ, σ, $\theta\sigma$, and $\sigma\theta$, which substitutions are idempotent?

2. Use the rule-based unification algorithm *unify* to decide if the following pairs of terms are unifiable or not, and provide each state in *unify* after applying a transformation rule.

   (a) $p(x, a) \doteq p(f(y), y)$

   (b) $p(x, f(x)) \doteq p(f(y), y)$

   (c) $p(f(a), g(x)) \doteq p(y, y)$

   (d) $q(x, y, f(y)) \doteq q(u, h(v, v), u)$

   (e) $q(a, x, f(g(y))) \doteq q(z, f(z), f(u))$

3. How to modify *unify*(s, t) so that it can be used to decide if a set S of terms are unifiable? That is, find an idempotent substitution σ and a term t such that such that $s\sigma = t$ for every $s \in S$.

4. Find the mgu of $s = f(f(...f(f(a, x_1), x_2), ...), x_n)$ and $t = f(x_n, f(x_{n-1}, f(..., f(x_1, a)...)))$ for $n = 5$.

5. Create the term graph $G = (V, E)$ for each of the following pairs of terms:

   (a) $s = p(x, a)$ and $t = p(f(y), y)$

   (b) $s = p(x, f(x))$ and $t = p(f(y), y)$

   (c) $s = q(x, y, f(y))$ and $t = q(u, h(v, v), u)$

   (d) $s = q(a, x, f(g(y)))$ and $t = q(z, f(z), f(u))$

   (e) $s = f(f(f(f(a, x_1), x_2), x_3), x_4)$ and $t = f(x_4, f(x_3, f(x_2, f(x_1, a))))$

   and show the equivalence classes of V after calling *unify*1 with a success.

6. Provide the pseudo code for the procedure *postOccurCheck* used in *lunify*. What is the complexity of your algorithm?

7. Design an efficient algorithm and provide its pseudo code for creating the mgu when the procedure *lunify* returns true. What is the complexity of your algorithm?

8. Show that all the left side s of each equivalence relation in Example 6.3.16 is greater than their right side t by $\succ_{lpo}$, and explain what precedence is used and what conditions of $\succ_{lpo}$ are used for $s \succ_{lpo} t$.

9. Show that all the left side s of each following equation is greater than their right side t by $\succ_{lpo}$, and explain what precedence is used and what conditions of $\succ_{lpo}$ are used for $s \succ_{lpo} t$.

$$\begin{aligned} x \times (y + z) &= (x \times y) + (x \times z) \\ (y + z) \times x &= (y \times x) + (z \times x) \\ (x \times y) \times z &= x \times (y \times z) \end{aligned}$$

   Can you use $\succ_{rpo}$ or $\succ_{kbo}$ to achieve the same result? Why?

10. Choose with justification a simplification order from $\succ_{lpo}$, $\succ_{rpo}$ and $\succ_{kbo}$ such that the left side s is greater than the right side t for each of the following equations (which defines the Ackermann's function):

$$\begin{aligned} A(0, y) &= s(y) \\ A(s(x), 0) &= A(x, s(0)) \\ A(s(x), s(0)) &= A(x, A(s(x), y)) \end{aligned}$$

11. Given the following statements:

    > Tony, Tom and Liz belong to the Hoofers Club. Every member of the Hoofers Club is either a skier or a mountain climber or both. No mountain climber likes rain, and all skiers like snow. Liz dislikes whatever Tony likes and likes whatever Tony dislikes. Tony likes rain and snow.

    The "Hoofers Club problem" asks for the solution of the following question: "Is there a member of the Hoofers Club who is a mountain climber but not a skier?" Please find the following resolution proofs for "Hoofers Club problem": (a) unit, (b) input, (c) positive, (d) negative, and (e) linear resolutions.

12. Use Prover9 to answer the question in the "Hoofers Club problem" of the previous problem. Please prepare the input to Prover9 and turn in the output file of Prover9.

13. Use binary resolution, hyper-resolution and ur-resolution, respectively, of Prover9 to answer the question that "Is John happy?" from the following statements:

    > Anyone passing his logic exams and winning the lottery is happy. But anyone who studies or is lucky can pass all his exams. John did not study but he is lucky. Anyone who is lucky wins the lottery.

14. Use hyper-resolution and ur-resolution, respectively, of Prover9 to show that "the sprinklers are on" from the following statements:

    > Someone living in your house is wet. If a person is wet, it is because of the rain, the sprinklers, or both. If a person is wet because of the sprinklers, the sprinklers must be on. If a person is wet because of rain, that person must not be carrying any umbrella. There is an umbrella in your house, which is not in the closet. An umbrella that is not in the closet must be carried by some person who lives in that house. Nobody are carrying any umbrella.

    Please prepare the input to Prover9 and turn in the output file of Prover9.

15. Following the approach illustrated in Example 6.4.4, use Prover9 to prove that

the following properties are true in the Hilbert system:

(a) $(x \to y) \to ((y \to z) \to (x \to z))$
(b) $(x \to (y \to z)) \to (y \to (x \to z))$
(c) $\neg x \to (x \to y)$
(d) $x \to (\neg x \to y))$
(e) $\neg\neg x \to x$
(f) $x \to \neg\neg x$
(g) $(x \to \neg x) \to \neg x$
(h) $(\neg x \to x) \to x$
(i) $(x \to y) \to (\neg y \to \neg x)$

CHAPTER 7
EQUATIONAL LOGIC

There are many equivalence relations in logic. When looking at a semantic level, we have logical equivalence and equal satisfiability, which are denoted by the symbols $\equiv$ and $\approx$, respectively, for both propositional and first-order logics. When looking at a syntactic level, we have the logical operator $\leftrightarrow$ to express the equality between two statements. Since a first-order language has two types of objects, i.e., formulas and terms, $\leftrightarrow$ is used for the formulas, the conventional symbol for the equality of terms is "=", which is a predicate symbol. Instead, we have been using "=" as a syntactical relation outside of the logical languages.

7.1 Equality of Terms

The equality is an omnipresent and important relation in every field of the mathematics. For example, how would you specify that "for every x, there exists a unique y such that the relation $p(x, y)$ holds"? For example, if the meaning of $p(x, y)$ is "x is the mother of y", then how would you state in first-order logic that "everybody has a unique mother"? If the meaning of $p(x, y)$ is $f(x) = y$, then how would you state that "the function $f(x)$ has a unique value"? Using $=$, the answer is easy:

$$\forall x \exists y\, p(x, y) \wedge (\forall z\, p(x, z) \rightarrow y = z)$$

The above formula is often denoted by $\forall x \exists! y\, p(x, y)$ in mathematics. In fact, when we translate a first-order formula into a propositional formula for searching a finite model, we used the formula

$$(f(x_1, ..., x_k) \neq y_1 \mid f(x_1, ..., x_k) \neq y_2 \mid y_1 = y_2),$$

which states that $f(x_1, ..., x_k)$ has a unique value. Using "=" gives us higher expressive power to specify statements.

- We can express "there are at least two elements x such that $A(x)$ holds" as $\exists x \exists y\, x \neq y \wedge A(x) \wedge A(y)$.
- We can express "there are at most two elements x such that $A(x)$ holds" as $\forall x \forall y \forall z\, (A(x) \wedge A(y) \wedge A(z)) \rightarrow x = y \vee y = z \vee x = z)$ This states that if we have three elements a for which $A(a)$ holds, then two of them must be equal.

- We can express "there are exactly two elements x such that $A(x)$ holds" as the conjunction of the above two statements.

7.1.1 Axioms of Equality

The axioms of the equality consist of five types of formulas, each is sufficiently famous to have earned itself a name. The variables appearing in the formulas are assumed universally quantified.

- **reflexivity**:
  $x = x$. That is, an object is always equal to itself.
- **symmetry**:
  $(x = y) \to (y = x)$. That is, it does not matter how the parameters of $=$ are ordered.
- **transitivity**:
  $(x = y) \wedge (y = z) \to (x = z)$. That is, the equality relation can be inherited.
- **function congruence**:
  $x_i = y_i \to f(..., x_i, ...) = f(..., y_i, ...)$. That is, if the arguments of a function are pair-wisely equal, the composed term will be equal.
- **predicate congruence**:
  $x_i = y_i \wedge p(..., x_i, ...) \to p(..., y_i, ...)$. That is, if the augments of a predicate are pair-wisely equal, then the resulting atoms are logically equivalent.

The first three axioms together say that "=" is an equivalence relation. The fourth and fifth are very similar and share the name *congruence*. The first congruence rule governs terms and the second congruence rule governs formulas. Both of them assert that having equal arguments ensures equal results. In fact, these are axiom schemata as we need a version for each function f of arity k and for each value i, $1 \leq i \leq k$. The similar requirement applies to each predicate p. For example, if the predicate p is the equality $=$, we obtain two axioms from the predicate congruence:

$$x_1 = y_1 \wedge (x_1 = x_2) \to (y_1 = x_2),$$
$$x_2 = y_2 \wedge (x_1 = x_2) \to (x_1 = y_2).$$

Of course, these two axioms can be easily deduced from the symmetry and transitivity of $=$. In other words, the predicate congruence is needed for predicates other than $=$. We will need different versions of the fourth and fifth axioms in different

first-order languages. These congruence axioms are not to be confused with the substitution rule, which allows the congruence of terms for variables.

All the five axioms can be easily converted into clauses so that we can use resolution to prove theorems involving =.

Example 7.1.1. Let $f(a) = b$ and $f(b) = a$, prove that $f(f(a)) = a$. Since resolution is a refutational prover, we need to add the negation of $f(f(a)) = a$, $f(f(a)) \neq a$, into the clauses from the equality axioms and the premises, and show the clause set is unsatisfiable. □

1	$(f(a) = b)$	premise
2	$(f(b) = a)$	premise
3	$(f(f(a)) \neq a)$	negation of the goal
4	$(x \neq y \mid y \neq z \mid x = z)$	transitivity
5	$(x \neq y \mid f(x) = f(y))$	congruence
6	$(f(f(a)) = f(b))$	resolvent from 1 and 5
7	$(f(b) \neq z \mid f(f(a)) = z)$	resolvent from 6 and 4
8	$(f(f(a)) = a)$	resolvent from 2 and 7
9	$(\,)$	resolvent from 3 and 8

The above example shows clearly that without equality axioms, we cannot have a resolution proof of $f(f(a)) = a$.

7.1.2 Semantics of "="

The equality symbol is meant to model the equality of objects in a domain. For example, when we say "2 + 0 = 2", or $add(s(s(0)), 0) = s(s(0))$, we meant that "2+0" and "2" represent the same object in a domain. That is, $s = t$ is true if s and t represent "equal" or "identical" objects in any domain. We are asserting that two different descriptions refer to the same object. Because the notion of identity can be applied to virtually any domain of objects, the equality is often assumed to be omnipresent in every logic. However, talk of "equality" or "identity" raises messy philosophical questions. "Am I the same person I was three days ago?"

We need a simple and clear way to define the meaning of =. For a first-order language $L = (P, F, X, Op)$, a Herbrand model for a CNF A can be simply represented by a set H of ground atoms such that only those atoms in H are interpreted to be true. Because of the existence of the equality axioms, the set H must hold some properties.

Definition 7.1.2. (**congruence**) *A relation* $=_m$ *over the set of terms* $T(F, X)$ *is*

called a congruence *if* $=_m$ *is an equivalence relation satisfying the congruence:*

$$f(s_1, ..., s_k) =_m f(s_2, ..., s_k) \text{ if for any } 1 \leq i \leq k, s_i =_m t_i)$$

for every function f/k.

Proposition 7.1.3. *Let* H *be a Herbrand model of* A *which is in CNF and contains all the equality axioms. Define* $s =_H t$ *iff* $(s = t) \in H$ *for every* $s, t \in T(F)$. *Then* $=_H$ *is a congruence over* $T(F)$.

Proof. The first three equality axioms ensure that $=_H$ is an equivalence relation and the fourth axiom ensures that $=_H$ is monotonic. □

For any congruence relation $=_m$, since $=_m$ is an equivalence relation, we may partition $T(F)$ into equivalence classes such that each class contains equivalent objects. For every $s \in T(F)$, let

$$[s] = \{t \mid s =_m t, t \in T(F)\},$$

the *congruence class* of s. Let

$$T(F)/=_m \quad = \{[s] \mid s \in T(F)\},$$

the set of all congruence classes, which is also called the *Herbrand base modulo* $=_m$.

Example 7.1.4. Let $F = \{0, s^{(1)}\}$ and A contains $s(s(s(x))) = x$ and the equality axioms. Let H be a Herbrand model of A in which $(0 = s(0)) \notin H$, then □

$$\begin{aligned}
T(F) &= \{0, s(0), s(s(0)), s(s(s(0))), s^4(0), ...\},\\
[0] &= \{0, s(s(s(0))), s^6(0), s^9(0), ...\},\\
[s(0)] &= \{s(0), s^4(0), s^7(0), s^{10}(0), ...\},\\
[s(s(0))] &= \{s(s(0)), s^5(0), s^8(0), s^{11}(0), ...\},\\
T(F)/=_H &= \{[0], [s(0)], [s(s(0))]\}.
\end{aligned}$$

Now, it is easy to check if two ground terms s and t are equal or not under H: $(s = t) \in H$ iff $t \in [s]$ (or $s \in [t]$), i.e., $s =_H t$. In fact, when we define a Herbrand model for a formula with equality, at first we can define a congruence $=_m$, and then regard $T(F)/=_m$ as the domain of the Herbrand model with equality: $(s = t) \in H$ iff $s =_m t$. This way, we do not have to concern about the equality axioms explicitly.

7.1.3 Theory of Equations

A set of equations is simply a set of positive unit clauses where the only predicate symbol is "=".

Example 7.1.5. In modern algebra, a *group* is $G = (S, *)$, where S is a set of elements and $*$ is a binary operator $*$ satisfying the properties that (a) $*$ is associative and closed on S; (b) there exists an identity element in S; and (c) every element of S has an inverse in S. Using the equational approach, the identity is denoted by e, for each $x \in S$, the inverse of x is denoted by $i(x)$, with the following three equations as axioms:

1	$x * e$	$=$	x	e is the identity element
2	$x * i(x)$	$=$	e	$i(x)$ is the inverse of x
3	$(x * y) * z$	$=$	$x * (y * z)$	$*$ is associative

From these three equations, we may prove many interesting properties of the group theory, such as $i(i(x)) = x$. □

Definition 7.1.6. *Given a set E of equations, the relation $=_E$ is recursively defined as follows:*

1 $s =_E t$ if $s = t \in E$;
2 $t =_E t$ for any $t \in T(F, X)$;
3 $s =_E t$ if $t =_E s$;
4 $s =_E t$ if $s =_E r, r =_E t$;
5 $f(..., s_i, ...) =_E f(..., t_i, ...)$ if $s_i =_E t_i$ for any $f \in F$;
6 $s\sigma =_E t\sigma$ if $s =_E t$ for any $\sigma : X \longrightarrow T(F, X)$.

Recall that every binary relation R, including $=_E$, can be regarded as a set of pairs of items, i.e., $\{(s, t) \mid R(s, t)\}$. Similarly, E can be also regarded as a set of pairs of terms.

Definition 7.1.7. *Given a binary relation R over $T(F, X)$, R is said to be* closed *under*

- reflexivity *if $(t, t) \in R$ for any $t \in T(F, X)$;*
- symmetry *if $(t, s) \in R$ whenever $(s, t) \in R$;*
- transitivity *if $(r, t) \in R$ whenever $(r, s), (s, t) \in R$;*
- congruence *if $(f(..., s, ...), f(..., t, ...)) \in R$ whenever $(s, t) \in R$;*

- instantiation *if* $(s\sigma, t\sigma) \in R$ *whenever* $(s,t) \in R$ *for any* $\sigma : X \longrightarrow T(F,X)$.

The transitive closure *is the minimum superset of* R *that is closed under transitive; the* equivalence closure *is the minimum superset of* R *that is closed under reflexivity, symmetry and transitivity; the* congruence closure *is the minimum equivalence closure of* R *that is closed under congruence.*

Proposition 7.1.8. *For any set* E *of equations,* $=_E$ *is a minimum stable congruence containing* E *and closed under instantiation.*

Proof. Let us check the conditions for $s =_E t$ to be true: Condition 1 says $E \subset =_E$; condition 2 says $(t,t) \in =_E$. Conditions 3-6 says $=_E$ is closed under symmetry, transitivity, congruence, and instantiation. □

Example 7.1.9. Using F from Example 7.1.4, let $E = \{s(s(s(x))) = x\}$ and $X = \{x\}$. Since $=_E$ is a congruence, we can check that $T(F)/=_E = T(F)/=_H = \{[0], [s(0)], [s(s(0))]\}$, $[x] = \{x, s(s(s(x))), s^6(x), ...\}$, and

$$T(F,X)/=_E = \{[0], [s(0)], [s(s(0))], [x], [s(x)], [s(s(x))]\}.$$

□

Proposition 7.1.10. *Given* E*, let* E_{ax} *be the equality axioms associated with* E*, then for every* $(s,t) \in =_E$*,* $E \cup E_{ax} \models (s = t)$.

Proof. (Sketch) Check all the six conditions for $s =_E t$: Condition 1 is sound because $A \models A$. Conditions 2-5 are sound because of E_{ax}. Condition 6 is sound because the free variables are assumed to universally quantified. □

Since $=_E$ contains all the equations that are logical consequence of E and the equality axioms, we call $= E$ the *theory* of E.

Example 7.1.11. Let E be the three equations in Example 7.1.5, we may show that $e * x =_E x$, $i(x) * x =_E e$, $i(e) =_E e$, $i(i(x)) =_E x$, $i(x * y) =_E i(y) * i(x)$, etc. The proofs will be provided in the next section. □

Given any two terms, say s and t, does $s =_E t$? This is an important decision problem with many applications in mathematics. In computation theory, all computable functions can be constructed by equations. Unfortunately, it is undecidable in general to answer $(s,t) \in =_E$, in other words we do have an algorithm which takes E, s, and t as input and returns yes if $s =_E t$.

In computer science, the *congruence closure problem* refers to the problem of deciding $s =_E t$ when E is a set of ground equations. This is a decidable problem

and there exist efficient algorithms, some of them are based on the union-find data structures we used for the unification algorithm. There are a number of applications using congruence closures. The detailed discussion on this topic can be found in Chapter 12.

7.2 Rewrite Systems

In Example 7.1.1, we showed how to prove $f(f(a)) = a$ from $f(a) = b$ and $f(b) = a$, using resolution and the equality axioms. A straightforward proof is to replace $f(a)$ in $f(f(a)) = a$ by b to get $f(b) = a$, and then replace $f(b)$ by a to get $a = a$. In this simple example, we treat $f(a) = b$ and $f(b) = a$ as two rewrite rules and each replacement is one application of a rewrite rule, called *rewriting*. To replace "equal" by "equal" is a common practice in mathematics and logics. For example, we used equivalence relations as rewrite rules to simplify formulas to normal forms. rewrite rules are not only a common inference system in everyday mathematics, but also a very efficient computational technique, as they can involve very little search. We will devote this section to a brief study of them, including a discussion of some interesting theoretical results.

7.2.1 Rewrite Rules

Definition 7.2.1. *A* rewrite rule *is an ordered pair of terms (l, r), often denoted by $l \to r$, where l is the left side and r is the right side of the rule. A term t is* rewritten *to t' by $l \to r$, denoted by $t \Rightarrow t'$, if there exists a substitution σ and a position p of t such that $l\sigma = t/p$ and $t' = t[p \leftarrow r\sigma]$, where t/p denotes the subterm of t at position p.*

To rewrite t by $l \to r$, we have to find a subterm of t at position p and a substitution σ such that $t/p = l\sigma$. In this case, we say l *matches* t/p and σ is the *matching*. Matching is one-way unification and finding matching is easier than finding a unifier as the variables in t cannot be affected by σ. If we treat the variables of t as constants, then finding a matching is the same as finding a unifier of t/p and l.

Once the matching is found, we need to replace the matched subterm in t by the matched right side. Note that $t[p \leftarrow s]$ denote the term where the subterm at position p, t/p, (ref. Definition 5.1.4) is replaced by s. For example, if $g(x) \to h(x)$ is a rewrite rule, then term $t = f(a, g(b))$ can be rewritten to $t' = f(a, h(b))$, because $t/1 = g(b) = g(x)\sigma$ and $t'/1 = h(x)\sigma$, where $\sigma = [x \leftarrow b]$.

Definition 7.2.2. *A rewrite system R is a set of rewrite rules. A term t is written*

to t' by R, denoted by $t \Rightarrow_R t'$, or simply $t \Rightarrow t'$, if there exists a rewrite rule in R which rewrites t to t'.

As a convention, we will use $\Rightarrow^+$ and $\Rightarrow_*$ to denote the transitive closure and the reflexive and transitive closure of $\Rightarrow$, respectively. That is, $s \Rightarrow_R^* t$ means that s is rewritten to t by R using any number of rewriting.

While our major use of R is to represent a stable congruence relation, R can be used for representing any transitive, stable and monotonic relation. This might be inequality (like $>$ or $\geq$) or implication.

Example 7.2.3. In Chapter 1 (section 1.3.4.1), we have seen that the functions can be constructed by equations. For instance, we used equations to define the predecessor, addition, subtraction, and multiplication functions over the natural numbers. We may treat these equations as rewrite rules. □

$$\begin{aligned}
pre(0) &\rightarrow 0; \\
pre(s(x)) &\rightarrow x. \\
add(0, y) &\rightarrow y; \\
add(s(x), y) &\rightarrow s(add(x, y)). \\
sub(x, 0) &\rightarrow x; \\
sub(x, s(y)) &\rightarrow sub(pre(x), y)). \\
mul(0, y) &\rightarrow 0; \\
mul(s(x), y) &\rightarrow add(mul(x, y), y).
\end{aligned}$$

To compute $2 \times 2 = 4$, we rewrite $mul(s(s(0)), s(s(0)))$ to $s(s(s(s(0))))$. We give below also the positions for each rewriting and you may find out which rule is used for each rewriting.

$$\begin{array}{lll}
 & mul(s(s(0)), s(s(0))) & \text{at } \epsilon \\
\Rightarrow & add(mul(s(0), s(s(0))), s(s(0))) & \text{at } 1 \\
\Rightarrow & add(add(mul(0, s(s(0))), s(s(0))), s(s(0))) & \textit{at } 1.1 \\
\Rightarrow & add(add(0, s(s(0))), s(s(0))) & \text{at } 1 \\
\Rightarrow & add(s(s(0)), s(s(0))) & \text{at } \epsilon \\
\Rightarrow & s(add(s(0), s(s(0)))) & \text{at } 1 \\
\Rightarrow & s(s(add(0, s(s(0))))) & \text{at } 1.1 \\
\Rightarrow & s(s(s(s(0)))) &
\end{array}$$

In this language, if a term is a ground, it is not hard to prove by induction that the rewriting process will eventually terminate with a specific number, i.e. a term made only from 0 and s. Using this rewrite system, we can do all the addition, subtraction, and multiplication over the natural numbers by rewriting, if we do not care about the speed.

7.2.2 Termination of Rewrite Systems

A nice property of many rewrite systems, including the previous example, is that the application of rules to terms, ground or not, cannot go on forever; it will eventually terminate.

Definition 7.2.4. *A rewrite system R is said to be* terminating *if there exists no infinite sequences of terms $t_0, t_1, ..., t_i, ...$ such that $t_i \Rightarrow t_{i+1}$.*

To prove a rewrite system R is terminating, we may use a simplification order $\succ$ (ref. section 6.3.1).

Proposition 7.2.5. *Let $\succ$ be a simplification order and R a rewrite system. If $l \succ r$ for all every $l \to r$ in R, then R is terminating.*

Proof. By definition, $\succ$ is a well-founded, stable and monotonic order on terms. If $t_i \Rightarrow t_{i+1}$ by $l \to r$, then $t_i \succ t_{i+1}$, because $l \succ r$, and $\succ$ is stable and monotonic. If there exists an infinite sequence, $t_0, t_1, ..., t_i, ...$ such that $t_i \succ t_{i+1}$, then $\succ$ cannot be well-founded. □

The above proposition provides a sufficient condition for the termination of a rewrite system. In section 6.3.1, we discussed lexicographic path order (LPO), recursive path order (RPO), and Knuth-Bendix order (KBO). These orders are simplification orders and can be used to prove the termination of R.

Example 7.2.6. To show the termination of R in Example 7.2.3, we may use the lexicographic path order $\succ_{lpo}$ with the precedence $s \prec pre \prec add \prec sub \prec mul$ and every binary operator has the left-to-right status, with the exception of *sub*: *sub* must have the right-to-left status, as we want $[s(y), x] \succ^{lex} [y, pre(x)]$. □

In general, how to show that the rewriting process using R will terminate? Unfortunately, this is an undecidable problem. That is, we do not have an algorithm to answer this question for general R.

Definition 7.2.7. *Given a rewrite system R, if $s \Rightarrow_R^* t$ and there exists no t' such that $t \Rightarrow_R t'$, we say t is a* normal form *of s.*

Why is termination an important property of a rewrite system? the termination of R ensures that there exists a normal form for every term. Without termination, we will have hard time to control rewriting. In computation theory, termination is a divider for telling whether a decision problem is decidable or not. Some equations, such as the commutativity of $+$ or $\vee$, can never be made into a terminating rule. It is undecidable to tell if a rewrite system is terminating or not.

7.2.3 Confluence of Rewriting

Termination is one important property of a set of rewrite rules. Other important properties are being confluent and being canonical.

Example 7.2.8. In Example 7.1.5, we used three equations to specify a group $(S, *)$. These equations can be made as a rewrite system R:

$$\begin{array}{lrcl} 1 & e * x & \to & x \\ 2 & i(x) * x & \to & e \\ 3 & (x * y) * z & \to & x * (y * z) \end{array}$$

□

Let $t = (i(x) * x) * y$, and there are two ways for R to rewrite t:

$$\underline{(i(x) * x)} * y \;\Rightarrow \left\langle \begin{array}{ll} (3) & i(x) * (x * y) \\ (2) & \underline{e} * y \;\Rightarrow (1)\; y \end{array} \right.$$

That is, there are more than one way to rewrite a term. For t, we have two normal forms: $i(x) * (x * y)$ and y.

Definition 7.2.9. *A rewrite system R is said to be* confluent *if, for any term s, if $s \Rightarrow_R^* t_1$ and $s \Rightarrow_R^* t_2$, then there exists a term t' such that $t_1 \Rightarrow_R^* t'$ and $t_2 \Rightarrow_R^* t'$. R is said to be* canonical *if R is both terminating and confluent.*

That is, if R is confluent, it does not matter how you go about applying the rules to a term, you will get the same result. There are several concepts of "confluence" proposed in the literature. They are equivalent for terminating rewrite systems. The termination of R ensures the existence of a normal form for every term; the confluence of R ensures the uniqueness of normal forms. When R is canonical, you do not need to worry about at which positions and choose which rule to rewrite a term. Any choices can be made and the end result will be the same. Let us denote the unique normal form of t in a canonical rewrite system by $t \downarrow_R$, the canonical form of t.

Definition 7.2.10. *A canonical rewrite system R is a decision procedure for a set E of equations if for any two terms s and t, $s =_E t$ iff $s \downarrow_R = t \downarrow_R$.*

We say R is a decision procedure for E because the computation of $t \downarrow_R$ is terminating and the following theorem is true.

Theorem 7.2.11. *Let R be a canonical rewrite system satisfying the following conditions:*

1. *For each rewrite rule* $l \to r$ *of* R, $l =_E r$;

2. *For each equation* $s_1 = s_2$ *of* E, $s_1 \downarrow_R = s_2 \downarrow_R$.

Then R *is a decision procedure for* E.

Proof. For each rule $l \to r$ of R, since $l =_E r$, if $s \Rightarrow t$ by $l \to r$, then $s =_E t$ by applying the instantiation and congruence rules on $l =_E r$. Applying this result to each rewriting, we have $s =_E s \downarrow_R$ and $t =_E t \downarrow_R$. If $s \downarrow_R = t \downarrow_R$, then $s =_E t$.

If $s =_E t$, check all the six conditions for obtaining $s =_E t$: If $s =_E t$ because $s = t \in E$, then $s \downarrow_R = t \downarrow_R$. If $s =_E t$ because $t =_E s$, then $t \downarrow_R = s \downarrow_R$ implies $s \downarrow_R = t \downarrow_R$. If $s =_E t$ because $s =_E r$ and $r =_E t$, then $s \downarrow_R = r \downarrow_R = t \downarrow_R$. If $s = f(..., s_i, ...) =_E f(..., t_i, ...) = t$ because $s_i =_E t_i$, then $s_i \downarrow_R = t_i \downarrow_R$ implies $s \downarrow_R = t \downarrow_R$. If $s = s'\sigma =_E t'\sigma = t$ because $s' =_E t'$, then $s' \downarrow_R = t' \downarrow_R$ implies $s \downarrow_R = t \downarrow_R$. In all cases, we have $s \downarrow_R = t \downarrow_R$ if $s =_E t$. □

Even when a rewrite system is canonical we may still want to exercise carefully our choices of rules and positions in a term to apply rules, in order to minimize the amount of computational effort required to find the canonical form. We may want to explore only the shortest rewriting steps. For instance, choosing a position for rewriting, we may use an outermost-position-first strategy or an innermost-position-first strategy. A careful inspection of the rules to be applied may reveal that which strategy works better.

7.2.4 The Knuth-Bendix Completion Procedure

In Example 7.2.8, we have seen that term $t = (i(x) * x) * y$ has two normal forms, i.e., $i(x) * (x * y)$ and y. The rewrite system in the example is not confluent. Knuth and Bendix suggested a procedure which takes a set of equations and a simplification order, and each equation is oriented into rewrite rules according to the simplification order. So-called *critical pairs* are then computed from the rewrite rules. If all the critical pairs have the same normal form, we obtain a canonical rewrite system. If not, such critical pairs are added into the set of equations and continue.

Definition 7.2.12. *Given two rewrite rules which share no variables and may differ only by variable names, say* $l_1 \to r_1$ *and* $l_2 \to r_2$, *a* critical pair *of the two rules is a pair of terms* $s = t$, *where* $s = r_1\sigma$, $t = l_1[p \leftarrow r_2]\sigma$, p *is a non-variable position of* l_1, *and* σ *is the mgu of* l_1/p *and* l_2. *We say* l_2 *is* superposed *at* p *of* l_2 *to produce the critical pair and this process is called* superposition.

In Example 7.2.8, we are given three rewrite rules. If we rename the variables in rule 3 as $(x_3 * y_3) * z_3 \to x_3 * (y_3 * z_3)$, we can compute a critical pair from rule 3 and rule 2, by superposing $l_2 = i(x) * x$ at position 1 of l_3, that is, unifying l_2 and $l_3/1 = x_3 * y_3$ with the mgu $\sigma = [x_3 \leftharpoonup i(x), y_3 \leftharpoonup x, z_3 \leftharpoonup y]$ and generate the critical pair $(i(x) * (x * y), e * y)$ from $l_3\sigma = (i(x) * x) * y$ as illustrated in Example 7.2.8.

During the computation of a critical pair, applying the mgu σ to l_1, $l_1\sigma$ can be rewritten to $r_1\sigma$ by $l_1 \to r_1$. At the same time, since $l_1/p\sigma = l_2\sigma$, $l_1\sigma$ contains an instance of l_2 at position p, so $l_1\sigma$ can be also rewritten to $l_1[p \leftharpoonup l_2]\sigma$ by $l_2 \to r_2$. In other words, the critical pair is obtained by two rewritings on $l_1\sigma$ using the two rules, respectively:

$$l_1\sigma \;\Rightarrow \left\langle \begin{array}{ll} r_1\sigma & \text{by } l_1 \to r_1 \\ l_1[p \leftharpoonup r_2]\sigma & \text{by } l_2 \to r_2 \end{array} \right.$$

Lemma 7.2.13. *If $l_1 =_E r_1$ and $l_2 =_E r_2$, and (s, t) is a critical pair from them, then $s =_E t$.*

Proof. From $l_1 =_E r_1$, we have $l_1\sigma =_E r_1\sigma$. From $l_2 =_E r_2$ and $l_1/p\sigma = l_2\sigma$, we have $l_1\sigma =_E l_1[p \leftharpoonup r_2]\sigma$. Hence $s = r_1\sigma =_E l_1[p \leftharpoonup r_2]\sigma = t$. □

Example 7.2.14. We will show to obtain a canonical rewrite system from the three rewrite rules in Example 7.2.8: □

$$\begin{array}{lrcl} 1 & e * x & \to & x \\ 2 & i(x) * x & \to & e \\ 3 & (x * y) * z & \to & x * (y * z) \end{array}$$

The lexicographical path order is used with the precedence $e \prec * \prec i$. When computing a critical pair of the two rules, we assume that the variables in one of the rules are implicitly subscribed by the rule number. For example, the third rule is $(x_3 * y_3) * z_3 \to x_3 * (y_3 * z_3)$.

The first critical pair is $i(x) * (x * y) = e * y$, which, after rewriting $e * y$ to y, will be oriented into the fourth rewrite rule:

$$4 \quad i(x) * (x * y) \quad \to \quad y$$

A critical pair comes from rule 4 and itself, and the mgu of $l_4/2$ and l_4 is $\sigma = [x_4 \leftharpoonup i(x), y_4 \leftharpoonup x * y]$:

$$i(i(x)) * \underline{(i(x) * (x * y))} \;\Rightarrow \left\langle \begin{array}{ll} (4) & x * y \\ (4) & i(i(x)) * \underline{y} \end{array} \right.$$

The critical pair is $x * y = i(i(x)) * y$, which produces the fifth rule:

$$5 \quad i(i(x)) * y \quad \rightarrow \quad x * y$$

A critical pair comes from rule 4 and rule 2, and the mgu of $l_4/2$ and l_2 is $\sigma = [x_4 \leftharpoonup i(x), y_4 \leftharpoonup x]$:

$$i(i(x)) * \underline{(i(x) * x)} \Rightarrow \left\langle \begin{array}{ll} (4) & x \\ (2) & i(i(x)) * \underline{e} \rightarrow (5) \quad x * e \end{array} \right.$$

The critical pair is $x = x * e$, which produces the sixth rule:

$$6 \quad x * e \quad \rightarrow \quad x$$

A critical pair comes from rule 6 and rule 2 with $\sigma = [x_6 \leftharpoonup i(e)]$ and generates the seventh rule:

$$7 \quad i(e) \quad \rightarrow \quad e$$

A critical pair comes from rule 5 and rule 2 with $\sigma = [x_2 \leftharpoonup i(x), y \leftharpoonup i(x)]$:

$$i(i(x)) * i(x) \Rightarrow \left\langle \begin{array}{ll} (5) & x * i(x) \\ (2) & e \end{array} \right.$$

and generates the eighth rule:

$$8 \quad x * i(x) \quad \rightarrow \quad e$$

A critical pair comes from rule 6 and rule 5 with $\sigma = [x_6 \leftharpoonup i(i(x)), y \leftharpoonup e]$:

$$i(i(x)) * e \Rightarrow \left\langle \begin{array}{ll} (6) & i(i(x)) \\ (5) & x * e \rightarrow (6) \quad x \end{array} \right.$$

and generates the ninth rule:

$$9 \quad i(i(x)) \quad \rightarrow \quad x$$

A critical pair comes from rule 4 and rule 3, and the mgu of $l_4/2$ and l_3 is $\sigma = [x_4 \leftharpoonup x * y, y_4 \leftharpoonup z]$:

$$i(x * y) * \underline{((x * y) * z)} \Rightarrow \left\langle \begin{array}{ll} (4) & z \\ (3) & i(x * y) * (x * (y * z)) \end{array} \right.$$

and generates the tenth rule:

$$10 \quad i(x * y) * (x * (y * z)) \quad \rightarrow \quad z$$

A critical pair comes from rule 10 and rule 8, and the mgu of $l_{10}/2.2$ and l_9 is $\sigma = [x_9 \leftharpoonup y, z \leftharpoonup i(y)]$:

$$i(x * y) * (x * \underline{(y * i(y))}) \Rightarrow \left\langle \begin{array}{l} (10)\ \ i(y) \\ (8)\ \ i(x * y) * (x * \underline{e}) \Rightarrow (6)\ \ i(x * y) * x \end{array} \right.$$

and generates the eleventh rule:

$$11 \quad i(x * y) * x \quad \rightarrow \quad i(y)$$

A critical pair comes from rule 11 and rule 4, and the mgu of $l_{11}/1.1$ and l_4 is $\sigma = [x_{11} \leftharpoonup i(x), y_{11} \leftharpoonup x * y]$:

$$i(\underline{i(x) * (x * y)}) * i(x) \Rightarrow \left\langle \begin{array}{l} (11)\ \ i(x * y) \\ (4)\ \ i(\underline{y}) * i(x) \end{array} \right.$$

and generates the twelfth rule:

$$12 \quad i(x * y) \quad \rightarrow \quad i(y) * i(x)$$

The final critical pair comes from rule 4 and rule 5, and generates the thirteenth rule:

$$13 \quad x * (i(x) * y) \quad \rightarrow \quad y$$

Since rules 5, 10, and 11 can be rewritten to the identity i.e., $t = t$, by rules 5 and 12, the final set of rewrite rules are

$$\begin{array}{lrcl} 1 & e * x & \rightarrow & x \\ 2 & i(x) * x & \rightarrow & e \\ 3 & (x * y) * z & \rightarrow & x * (y * z) \\ 4 & i(x) * (x * y) & \rightarrow & y \\ 6 & x * e & \rightarrow & x \\ 7 & i(e) & \rightarrow & e \\ 8 & x * i(x) & \rightarrow & e \\ 9 & i(i(x)) & \rightarrow & x \\ 12 & i(x * y) & \rightarrow & i(y) * i(x) \\ 13 & x * (i(x) * y) & \rightarrow & y \end{array}$$

All the critical pairs from this set of 10 rules can be reduced to the identity. These 10 rules are sufficient to solve the *word problem* for a free group with no other properties. The *word problem* is to find a decision procedure to decide whether any two terms are equal, according to the axioms of group theory. The above rewrite system can serve as a decision procedure by checking if the two terms have the same canonical form.

The above example illustrates the execution of the Knuth-Bendix completion procedure, which is described by the following pseudo-code.

Procedure 7.2.15. The procedure *KnuthBendix*$(E, \succ)$ takes a set E of equations and a simplification order $\succ$, and generates a canonical rewrite system from E when it succeeds. It calls four procedures:

- *pickEquation*(E) will pick an equation from E;
- $NF(t, R)$ returns a normal form of term t by R.
- *rewritable*(t, r) will check if rewrite rule r can rewrite term t;
- *criticalPairs*(r, R) will return the set of all critical pairs between rule r and every rule in R.

proc *KnuthBendix*$(E, \succ)$
1 $R := \emptyset$
2 **while** $(E \neq \emptyset)$ **do**
3 $(s, t) :=$ *pickEquation*(E); $E := E - \{(s, t)\}$
4 $s := NF(s, R); t := NF(t, R)$ // normalize s and t by R
5 **if** $(s = t)$ **continue** // identity is discarded
6 **else if** $(s \succ t)$ $r := (s \to t)$
7 **else if** $(t \succ s)$ $r := (t \to s)$
8 **else return** "failure" // s and t are not comparable by $\succ$
9 $R := R \cup \{r\}$; // add new rule into R
10 **for** $(s \to t) \in R - \{r\}$ **do** // inter-reduction
11 **if** *rewritable*(s, r) $R := R - \{(s, t)\}; E := E \cup \{(s, t)\}$
12 **if** *rewritable*(t, r) $R := R - \{(s, t)\} \cup \{(s, NF(t, R)\}$
13 $E := E \cup$ *criticalPairs*(r, R)
14 **return** R;

The procedure is not an algorithm because it may go on forever due to ever-growing set of rules and critical pairs. Note that the procedure may fail if an equation cannot be oriented into a rewrite rule. There are many attempts to reduce the cases of failures. For instance, if an equation like $f(x, g(x)) = h(y, g(y))$ cannot be oriented, we may introduce a new constant c to make two equations $f(x, g(x)) = c$ and $h(y, g(y)) = c$. In general, if we have an equation $s(x, z) = t(y, z)$, where x does not appear in t and y does not appear in s, we may introduce a new function $h(z)$ to create two new equations: $s(x, z) = h(z)$ and $t(y, z) = h(z)$. For the axioms $AC = \{(x * y) * y = x * (y * z), x * y = y * x\}$, we may consider the rewriting over $T(F, X)/=_{AC}$, the so-called rewriting modulo AC (associativity and commutativity).

Definition 7.2.16. *A rewrite system R is said to be* inter-reduced *if for any rule $l \to r \in R$, l and r are not rewritable by $R - \{l \to r\}$.*

Lemma 7.2.17. *If KnuthBendix$(E, \succ)$ returns R, then R is inter-reduced.*

Proof. Line 11 (resp. 12) ensures the left (resp. right) side of each rule in R is not rewritable by any other rule. □

When *KnuthBendix* terminates successfully, we obtain an inter-reduced canonical rewrite system.

Theorem 7.2.18. *If KnuthBendix$(E, \succ)$ returns R, then R is inter-reduced and canonical.*

Proof. R is inter-reduced by the above lemma. For canonicality, at first, R is terminating, because for every $l \to r$ of R, $l \succ r$, where $\succ$ is a simplification order. Secondly, all the critical pairs of R are computed and each pair is rewritten to the same term by R (if not, a new rule will be added into R).

To show that R is confluent, we do induction on t based on $\succ$. As a base case, if t is not rewritable by R, then R is confluent trivially at t. Suppose, for $i = 1, 2$, $t \Rightarrow t_i$ by $l_i \to r_i \in R$ at position p_i of t, i.e., $t_i = t[p \leftarrow r_i\sigma_i]$.

Case 1: If p_1 and p_2 are disjoint positions, then both t_1 and t_2 can be written to a common term $t' = t[p_1 \leftarrow r_1\sigma_1, p_2 \leftarrow r_2\sigma_2]$.

Case 2: If p_1 and p_2 are not disjoint positions, without loss of generality, assume $p_1 = \epsilon$, then $t = l_1\sigma_1$, and $t/p_2 = l_1\sigma_1/p_2 = l_2\sigma_2$. There are two subcases to consider:

Case 2.1: If p_2 is a non-variable position of l_1, then $l_1\sigma_1/p_2 = l_1/p_2\sigma_1 = l_2\sigma_2$, that is, l_1/p_2 and l_2 are unifiable. Thus, there exists a critical pair (s_1, s_2) between the two rules such that $t_1 = s_1\theta$ and $t_2 = s_2\theta$ for some substitution θ. Since s_1 and s_2 are rewritten to the same term by R, so are t_1 and t_2.

Case 2.2: When p_2 is not a non-variable position of l_1, finding a common successor of t_1 and t_2 is left as an exercise. Note that in this case there must exist a variable position p of l_1 such that p is a prefix of p_2.

By induction hypotheses, R is confluent at t_1 and t_2, respectively, because $t_1 \succ t$ and $t_2 \succ t$. Now we have shown that t_1 and t_2 are rewritten to a common term in all cases, R must be confluent at t. □

Example 7.2.19. To illustrate case 2.2 in the above proof, let $t = i(i(i(z))) * i(i(z))$ and R be the ten rules in Example 7.2.14. Then t can be rewritten to e by rule 2 and to $i(z) * i(i(z))$ by rule 9 at position 1.1. $i(z) * i(i(z))$ can be rewritten to $i(z) * z$ by rule 9 at position 2, and to e by rule 2. Note that both 1.1 and 2 are variable positions of t. □

Theorem 7.2.20. *If KnuthBendix$(E, \succ)$ returns R, then R is a decision procedure for E.*

Proof. (sketch) The previous theorem tells us that R is canonical. In light of Theorem 7.2.11, the condition 1 is ensured by Lemma 7.2.13; the condition 2 holds because every equation is either made into a rule or simplified to identity. □

In conclusion, rewrite systems can be used to put terms in normal form and to prove terms equal, equivalent or some transitive monotonic relations. Their application represents a powerful method of mathematical reasoning, because they can sometimes overcome the combinatorial explosions caused by other proof procedures, e.g. resolution. The desirable properties of rewrite systems are: termination and confluence. Rewrite systems can be shown to terminate with the aid of simplification orders and be confluent by looking at critical pairs formed from pairs of rules.

7.2.5 Rewrite Systems for Ground Equations

A *ground equation* is a pair of ground terms, i.e., no variables appear in the equation. The Knuth-Bendix procedure can be greatly simplified on ground equations because of the following:

- There exists a total simplification order for ground terms. This can be achieved by defining a total precedence over the function symbols and then applying one of the known simplification orderings, such as the Knuth-Bendix order (KBO), to terms.
- The critical pair computation is not needed if the rewrite system is interreduced. That is, if there exists a critical pair between two rewrite rules, then one rule can rewrite the left side of the other rule.

Example 7.2.21. Let f^3a and f^5a denote $f(f(f(a)))$ and $f(f(f(f(f(a)))))$, respectively. Feeding $E = \{f^5a = a, f^3a = a\}$ to the Knuth-Bendix procedure, the rewrite rules made from E should be $\{(1)\ f^5a \to a,\ \ (2)\ f^3a \to a\}$. Since $f^5a = f^2f^3a$, the critical pair from (1) and (2) is $f^2a = a$, and this is the same as we rewrite (1) to $f^2s = s$ by (2). In fact, rewriting happens first (line 11) in the Knuth-Bendix procedure. From $f^2a = a$, we have (3) $f^2 \to a$, which rewrites (2) to $f^1a = a$. The Knuth-Bendix procedure halts with $\{f^1a \to a\}$. □

Theorem 7.2.22. *For any set E of ground equations, there exists a canonical rewrite system R serving as a decision procedure of E.*

We leave the proof of this theorem as an exercise.

7.3 Inductive Theorem Proving

In Example 7.2.3, we converted a set E of equations into rewrite rules, where add is defined using two equations: $E = \{add(0, y) = y, add(s(x), y) = s(add(x, y))\}$. It is easy to prove by induction on i that $add(s^i(0), s^j(0)) = s^{i+j}(0) = add(s^j(0), s^i(0))$. Since $add(s^i(0), s^j(0)) = add(s^j(0), s^i(0))$, it is natural to state that $add(x, y) = add(y, x)$ for all ground terms x and y. Obviously, $add(x, y) = add(y, x)$ is not a logical consequence of E and E_{ax} (the equality axioms), because $E \cup E_{ax}$ may have models in which $add(x, y) = add(y, x)$ does hold. It is known that some properties of a first-order formula cannot be proved by deduction or contradiction; they can be proved only by induction. That is why we call these properties *inductive theorems*, $add(x, y) = add(y, x)$ is just one of them.

Mathematical induction is a fundamental and powerful rule of inference. Inductive theorem proving has important applications in formal verification and has been used to prove the correctness of computer architecture or algorithms. Like the previous chapters, we are particularly interested in inductive proving methods which can be automated in a computer.

7.3.1 Inductive Theorems

Definition 7.3.1. *Given a first-order language $L = (P, F, X, Op)$, a quantifier-free formula B is said to be an* inductive theorem *of A, if for all substitution $\sigma : X \longrightarrow T(F)$, $A \models B\sigma$.*

In other words, a quantifier-free formula is an inductive theorem of A if all its ground instances are logical consequence of A. We can show that $add(x, y) = add(y, x)$ and $add(add(x, y), z) = add(x, add(y, z))$ are inductive theorems of $E = \{add(0, y) = y, add(s(x), y) = s(add(x, y))\}$. Typically, A is also quantifier-free, with the understanding that all free variables act as universally quantified variables.

To prove an inductive theorem, we need an induction inference rule, which has many different versions.

Example 7.3.2. In Example 7.2.3, we convert a set E of equations into a rewrite

system so that the computation can be carried out by rewriting.

$$
\begin{array}{lrcl}
1 & pre(0) & \rightarrow & 0; \\
2 & pre(s(x)) & \rightarrow & x. \\
3 & add(0,y) & \rightarrow & y; \\
4 & add(s(x),y) & \rightarrow & s(add(x,y)). \\
5 & sub(x,0) & \rightarrow & x; \\
6 & sub(x,s(y)) & \rightarrow & sub(pre(x),y)). \\
7 & mul(0,y) & \rightarrow & 0; \\
8 & mul(s(x),y) & \rightarrow & add(mul(x,y),y).
\end{array}
$$

□

In the above example, the function symbols are $F = \{0, s, pre, add, sub, mul\}$, which can be divided into two parts: $C = \{0, s\}$ and $D = \{pre, add, sub, mul\}$. For each function in D, we have a complete definition over 0 and s:

- $pre(0) = 0$ and $pre(s^i(0)) = s^{i-1}(0)$ for $i > 0$;
- $add(s^i(0), s^j(0)) = s^{i+j}(0)$ for any $i, j \geq 0$;
- $sub(s^i(0), s^j(0)) = 0$ if $i \leq j$ and $pre(s^i(0), s^j(0)) = s^{i-j}(0)$ for $i > j$;
- $mul(s^i(0), s^j(0)) = s^{i*j}(0)$ for any $i, j \geq 0$;

It is easy to check that $T(F)/=_E = \{[0], [s(0)], ..., [s^i(0)], ...\}$, which is the domain of the Herbrand model with equality. We say $T(F)/=_E$ and $T(C)$ are *isomorphic* because there is a one-to-one mapping between $T(F)/E$ and $T(C)$. C is called *constructors* and D is called *defined functions* (or *interpreted functions*). Inductive theorems are about properties of defined functions.

7.3.2 Structural Induction

One basic induction rule is called the *structural induction rule*, which is based on the structure of terms built up by the constructors.

Structural Induction Rule

$$\frac{B(0) \qquad B(y) \rightarrow B(s(y))}{B(x)}$$

where $B(x)$ stands for an inductive theorem to be inducted on x.

The above rule states that to prove an inductive theorem $B(x)$, we need to prove two formulas $B(0)$ and $B(x) \rightarrow B(s(x))$. Its soundness is based on the condition that the isomorphic relation of $T(F)/=_E$ and $T(C)$.

Example 7.3.3. Let $B(x)$ be $add(x, 0) = x$. Applying the structural induction on this one, we need to prove (1) $add(0, 0) = 0$ and (2) $add(x, 0) = x \rightarrow add(s(x), 0) = s(x)$. (1) is trivial; (2) is simplified by rule 4, the definition of add, to: $add(x, 0) = x \rightarrow s(add(x, 0)) = s(x)$, or

$$(2') \qquad (add(x, 0) \neq x \mid s(add(x, 0)) = s(x)),$$

which is true by *equality crossing*, as described below. □

Equality Crossing

$$\frac{(s \neq t \mid A(s))}{(s \neq t \mid A(t))}$$

For instance, let $s \neq t$ be $add(x, 0) \neq x$ in $(2')$ of the above example, then $s(add(x, 0)) = s(x)$ is simplified to $s(x) = s(x)$. In other words, we used the induction hypothesis $add(x, 0) = x$ to show that $s(add(x, 0)) = s(x)$ is ture.

Equality crossing, called *crossing fertilization* by Boyer and Moore, is a simplification rule in the presence of equality axioms, due to the following result.

Proposition 7.3.4. (**soundness of equality crossing**) $E_{ax} \models (s \neq t \mid A(s)) \leftrightarrow (s \neq t \rightarrow A(t))$, *where* E_{ax} *is the equality axioms.*

Proof. From the congruence of equality, i.e., $x_i = y_i \wedge p(..., x_i, ...) \rightarrow p(..., y_i, ...)$, we can deduce $(s = t) \wedge A(s) \rightarrow A(t)$, whose clausal form is

$$(s \neq t \mid \overline{A(s)} \mid A(t))$$

The resolvent of this clause with $(s \neq t \mid A(s))$ is $(s \neq t \mid A(t))$. Switching s and t, the other implication also holds. □

The fundamental difference between using an inference rule like resolution and using an induction rule is not that a well-founded order, e.g. $s(x) \succ x$, is needed when applying induction. When using resolution for theorem proving by refutation, the empty clause is the goal and the resolution rule is used to derive this goal from the input clauses. When using an induction rule for theorem proving, the goal is the formula to be proved. However, we never try (and it is almost impossible) to

derive this goal from the input formulas by the induction rule. Instead, we try to generate subgoals from the goal by using the induction rule backward. That is, to prove $B(x)$ by induction, we generate two subgoals $B(0)$ and $B(y) \rightarrow B(s(y))$ from $B(x)$ and try to establish the validity of these subgoals. Remember that when $T(F)/=_E$ is isomorphic to $T(C)$, every defined function is completely defined on the constructors and these definition equations ensure the soundness of the induction rule from these equations.

Example 7.3.5. Continuing from Example 7.3.2, let us prove that $sub(add(x, y), y) = x$. If we do induction on x, the proof would not go through. On the other hand, we can do induction on y. By inspecting the definitions of *add* and *sub*, we see that x is the induction position for *add* and y is the induction position for *sub*.

The base case is $sub(add(x, 0), 0) = x$, which is simplified to true by rules 3 and 5. The inductive case is

$$sub(add(x, y), y) = x \rightarrow sub(add(x, s(y)), s(y)) = x$$

To simplify $add(x, s(y))$, we need $add(x, s(y)) = s(add(x, y))$, which is also an inductive theorem and its proof is left as an exercise. Suppose this theorem is available to us, then

$$\begin{array}{lll} & sub(add(x, s(y)), s(y)) & \\ = & sub(s(add(x, y)), s(y)) & \text{by } add(x, s(y)) = s(add(x, y)), \\ = & sub(pre(s(add(x, y)), y) & \text{by rule 6,} \\ = & sub(add(x, y), y) & \text{by rule 2,} \\ = & x & \text{by equality crossing.} \end{array}$$

□

7.3.3 Induction on Two Variables

The previous example shows that we need to choose the right variable to do the induction. Sometimes, we need to do induction on multiple variables.

Example 7.3.6. Adding the following two predicates to Example 7.3.2:

$$\begin{array}{llll} 11 & y < 0 & \rightarrow & \bot; \\ 12 & 0 < s(x) & \rightarrow & \top; \\ 13 & s(y) < s(x) & \rightarrow & y < x. \\ 14 & 0 \leq y & \rightarrow & \top; \\ 15 & s(x) \leq 0 & \rightarrow & \bot; \\ 16 & s(x) \leq s(y) & \rightarrow & x \leq y. \end{array}$$

To prove $(x \leq y) = \neg(y < x)$ by structural induction, neither induct on x alone nor on y alone will work. We need an induction on two variables. □

Structural Induction Rule on Two Variables

$$\frac{B(0,y) \qquad B(s(x),0) \qquad B(x,y) \rightarrow B(s(x),s(y))}{B(x,y)}$$

where $B(x,y)$ stands for an inductive theorem to be inducted on both x and y.

This is not the only induction rule for the two variables. Why should we use this rule for $(x \leq y) = \neg(y < x)$? The clue is in the definitions of $\leq$ and $<$: For $(x \leq y)$, we defined $\leq$ on $\langle 0,y\rangle, \langle s(x),0\rangle$ and $\langle s(x),s(y)\rangle$ with a recursive call on $\langle x,y\rangle$. For $(y < x)$, we also defined $<$ on $\langle 0,y\rangle, \langle s(x),0\rangle$, $\langle s(x),s(y)\rangle$ with a recursive call on $\langle x,y\rangle$ (the positions of x and y are switched here). In other words, the definition equations of these two predicates tell us what cases needed to consider: the base cases come from the definition equations without recursion; the inductive cases come from the definition equations with recursion. This is the major induction heuristic in Boyer and Moore's computational logic, where lisp functions are used instead of equational definitions.

Now back on the proof of $(x \leq y) = \neg(y < x)$, by the above induction rule, we need to prove three subgoals:

1 $(0 \leq y) = \neg(y < 0)$, which can be easily proved by showing $0 \leq y = \top$ by rule 14 and $\neg(y < 0) = \neg(\bot) = \top$ by rule 11.

2 $(s(x) \leq 0) = \neg(0 < s(x))$, which can be proved by showing $(s(x) \leq 0) = \bot$ by rule 15 and $\neg(0 < s(x)) = \neg(\top)) = \bot$ by rules 12.

3 $(x \leq y) = \neg(y < x) \rightarrow (s(x) \leq s(y)) = \neg(s(y) < s(x))$. Proof of 3 is also easy: $s(x) \leq s(y)$ is simplified to $x \leq y$ by rule 16 and $\neg(s(y) < s(x))$ to $\neg(y < x)$ by rule 13, then 3 becomes a trivial implication: $(x \leq y) = \neg(y < x) \rightarrow (x \leq y) = \neg(y < x)$.

7.3.4 Multi-Sort Algebraic Specifications

By far, we have focused on the natural number functions based on the constructors 0 and s. In many applications, we need other data structures, such as lists, trees, etc., which have their own constructors. Multi-sort algebraic specification will provide a convenient tool for such applications. For example, to work on the lists of natural numbers, we need two types: natural numbers and lists, and their domains can be specified as follows:

$$\begin{array}{lcl} \langle nat \rangle & ::= & 0 \mid s(\langle nat \rangle) \\ \langle list \rangle & ::= & nil \mid cons(\langle nat \rangle, \langle list \rangle) \end{array}$$

where the constructors for nat are $0 : nat$ and $s : nat \longrightarrow nat$ and the constructors for $list$ are $nil : list$ and $cons : nat, list \longrightarrow list$. We have seen the functions on nat; the functions over $list$ can be also defined using equations.

Example 7.3.7. The *append* and *reverse* operations of lists can be defined as follows:

$$\begin{array}{lrcl} & app : list, list & \longrightarrow & list \\ 1 & app(nil, y) & \rightarrow & y; \\ 2 & app(cons(x, y), z) & \rightarrow & cons(x, app(y, z)). \\ & rev : list & \longrightarrow & list \\ 3 & rev(nil) & \rightarrow & nil; \\ 4 & rev(cons(x, y)) & \rightarrow & app(rev(y), cons(x, nil)). \end{array}$$

□

To prove $rev(app(y, z)) = app(rev(z), rev(y))$ for all lists y and z, we need the following rule:

Structural Induction Rule for List

$$\frac{B(nil) \qquad B(y) \rightarrow B(cons(x, y))}{B(y)}$$

where $B(y)$ stands for an inductive theorem to be inducted on y.

The above rule reduces $rev(app(y, z)) = app(rev(z), rev(y))$ to two subgoals:

$$\begin{array}{ll} G_1 & rev(app(nil, z)) = app(rev(z), rev(nil)) \\ G_2 & rev(app(y, z)) = app(rev(z), rev(y)) \rightarrow \\ & \qquad rev(app(cons(x, y), z)) = app(rev(z), rev(cons(x, y))) \end{array}$$

We can simplify G_1 by rules 1 and 3 to $rev(z) = app(rev(z), nil)$, which suggests a more general property: $app(y, nil) = y$. This heuristic is called *generalization* and can be implemented automatically. Indeed, $app(y, nil) = y$ is an inductive theorem and its proof is left as an exercise. Using this lemma, $rev(z) = app(rev(z), nil)$ becomes true.

For G_2, we can simplify

$$rev(app(cons(x, y), z)) = app(rev(z), rev(cons(x, y)))$$

by rule 2 to $rev(cons(x, app(y, z)) = app(rev(z), rev(cons(x, y)))$, then to

$$app(rev(app(y, z)), cons(x, nil)) = app(rev(z), app(rev(y), cons(x, nil)))$$

by rule 4. Using equality crossing, $rev(app(y,z))$ is rewritten to $app(rev(z),rev(y))$, and we are facing

$$app(app(rev(y),rev(z)),cons(x,nil)) = app(rev(z),app(rev(y),cons(x,nil)))$$

A generalization of the above formula is the associativity of *app*:

$$app(app(x,y),z) = app(x,app(y,z))$$

which can be shown to be an inductive theorem separately. Once this is done, the proof of G_2 is complete.

The above example illustrates an interesting feature of inductive theorem proving: In order to prove $rev(app(y,z)) = app(rev(z),rev(y))$, we need to prove extra lemmas: $app(y,nil) = y$ and $app(app(x,y),z) = app(x,app(y,z))$. These lemmas can sometimes be generated automatically, and sometimes need to be provided by the user. For rigorousness, they are needed to be proved before they are used in a proof.

Let us prove another theorem $rev(rev(y)) = y$ by the same induction rule. The induction rule reduces $rev(rev(y)) = y$ to two subgoals: $G_3 : rev(rev(nil)) = nil$, which is easy to prove to be true by rule 2, and

$$G_4 : rev(rev(y)) = y \rightarrow rev(rev(cons(x,y))) = cons(x,y).$$

Applying rule 4, we rewrite $rev(rev(cons(x,y)))$ to $rev(app(rev(y),cons(x,nil)))$, which is then simplified to $app(rev(cons(x,nil)),rev(rev(y)))$. Since $rev(cons(x,nil))$ is rewritten to $cons(x,nil)$ by rules 4, 3 and 1, so $app(rev(cons(x,nil)),rev(rev(y)))$ is rewritten $cons(x,rev(rev(y)))$ by rules 2 and 1. Thus G_4 becomes

$$G_4 : rev(rev(y)) = y \rightarrow cons(x,rev(rev(y))) = cons(x,y)$$

Applying equality crossing, $rev(rev(y))$ is replaced by y and G_4 becomes true.

We have shown how use the language of algebraic specification of abstract data types to write axioms (definitions of functions and predicates), theorems or lemmas (intermediate inferences), etc. for inductive reasoning. In particular, we require that the axioms as well as theorems in an algebraic specification be expressed as a set of (conditional) equations. In other words, we are interested in automatic methods which can prove equations from equations by mathematical induction.

Example 7.3.8. We may define the insertion sort algorithm using equations.

$$
\begin{array}{rrcl}
 & isort : list & \longrightarrow & list \\
5 & isort(nil) & \rightarrow & nil; \\
6 & isort(cons(x, y)) & \rightarrow & insert(x, isort(y)). \\
 & insert : nat, list & \longrightarrow & list \\
7 & insert(x, nil) & \rightarrow & cons(x, nil); \\
8 & insert(x, cons(y, z)) & \rightarrow & cons(x, cons(y, z)) \textbf{ if } x \leq y; \\
9 & insert(x, cons(y, z)) & \rightarrow & cons(y, insert(x, z)) \textbf{ if } \neg(x \leq y); \\
 & sorted : list & \longrightarrow & boolean \\
10 & sorted(nil) & \rightarrow & \top; \\
11 & sorted(cons(x, nil)) & \rightarrow & \top; \\
12 & sorted(cons(x, cons(y, z))) & \rightarrow & (x \leq y) \wedge sorted(y, z);
\end{array}
$$

where $\leq$ is defined in Example 7.3.6. □

We may prove that $sorted(isort(x))$ is an inductive theorem (an exercise problem). If we define $perm(x, y)$ to be true iff x is a permutation of y, then we may also prove that $perm(x, isort(x))$ is an inductive theorem. Both theorems ensure the correctness of the insertion sort algorithm. These inductive theorems can be easily proved by today's inductive theorem prover. Many algorithms can be formally verified automatically using this approach.

7.4 Resolution with Equality

Up to this point in this chapter, our focus was on equations, which are positive unit clauses with the equality as the only predicate. Equations are expressive enough to describe all the computations. For instance, all Turing machines can be expressed as a set of equations. On the other hand, it is more convenient to express axioms and theorems in general clauses. The Knuth-Bendix completion procedure suggests that we do not need the equality axioms explicitly if we have an inference like superposition which computes critical pairs from two rewrite rules. This is also true in resolution-based proving as Robinson and Wos suggested the paramodulation rule, shortly after the invention of the resolution rule.

7.4.1 Paramodulation

Definition 7.4.1. (**paramodulation**) *Suppose clause C_1 is $(s = t \mid \alpha)$ and C_2 is $(A \mid \beta)$, where A is a literal and p is a non-variable position of A, α and β are the rest literals in C_1 and C_2, respectively, the* paramodulation *rule is defined by the*

following schema:

$$\frac{(s = t \mid \alpha), \qquad (A \mid \beta)}{(A[p \leftharpoonup t] \mid \alpha \mid \beta)\sigma}$$

where σ is the mgu of s and A/p. The clause $(A[p \leftharpoonup t] \mid \alpha \mid \beta)\sigma$ produced by the paramodulation rule is called paramodulant *of the paramodulation; C_1 and C_2 are the* parents *of the paramodulant. We use $paramod(C_1, C_2)$ to denote the clause $(A[p \leftharpoonup t] \mid \alpha \mid \beta)\sigma$.*

Example 7.4.2. Let C_1 be $(f(g(y)) = a \mid r(y)$ and C_2 be $(p(g(f(x))) \mid q(x))$, then $f(g(y))$ and $p(g(f(x)))/1.1 = f(x)$ are unifiable with $\sigma = [x \leftharpoonup g(y)]$. The paramodulant of this paramodulation is $(p(g(a)) \mid q(g(y)) \mid r(y)$. □

Proposition 7.4.3. (**soundness of paramodulation**) *For any clauses C_1 and C_2, if $paramod(C_1, C_2)$ exists, then $E_{ax} \cup \{C_1, C_2\} \models paramod(C_1, C_2)$, where E_{ax} is the equality axioms.*

Proof. Let σ be the mgu used in the paramodulation, $C_1\sigma$ is $(s' = t') \mid \alpha'$ and $C_2\sigma$ is $(A'[s'] \mid \beta'$, then $paramod(C_1, C_2)$ is $(A'[t'] \mid \alpha' \mid \beta')$.

From the monotonic axiom, we can deduce $(s' = t') \wedge A'(s') \rightarrow A'(t')$, whose clausal form is $C_3 : \ (s' \neq t' \mid \overline{A'(s')} \mid A'(t'))$. $resolve(C_1\sigma, C_3) = (\overline{A'(s')} \mid A'(t') \mid \alpha')$, named as C_4. $resolve(C_2\sigma, C_4) = (A'(t') \mid \alpha' \mid \beta')$, which is the same as $paramod(C_1, C_2)$. □

Using paramodulation, we do not need most of the equality axioms. It can be shown that any consequence from the equality axioms can be deduced from paramodulation, with the only exception of the reflexivity: $x = x$. The axiom $x = x$ is indispensable in showing that $A = \{f(x) \neq f(y)\}$ is unsatisfiable. We may introduce the following inference rule which makes the axiom $x = x$ redundant.

Definition 7.4.4. (**reflexing**) *Suppose clause C is $(s \neq t \mid \alpha)$,* reflexing *is an inference rule defined by the following schema:*

$$\frac{(s \neq t \mid \alpha)}{(\alpha)\sigma}$$

where σ is the mgu of s and t.

Reflexing works in the same way as factoring: factoring removes one literal by unifying two literals; reflexing removes $s \neq t$ by unifying s and t.

Paramodulation combined with resolution, factoring and reflexing is refutationally complete for theorem proving with equality. That is, if $A \cup E_{ax}$ is unsatisfiable, where E_{ax} is the equality axioms, then the empty clause can be derived from

A by paramodulation, resolution, factoring, and reflexing. It remains complete with any complete resolution strategies, such as positive resolution or ordered resolution. We may add restrictions on paramodulation as we put restrictions on resolution, without sacrificing the refutational completeness.

Definition 7.4.5. (**ordered paramodulation**) *Given a simplification order* $\succ$, *suppose* $s = t$ *is maximal in clause* $C_1\,(s = t \mid \alpha)$, $s \not\prec t$, *and literal* A *is maximal in* $C_2\,(A \mid \beta)$, p *is a non-variable position of* A, α *and* β *are the rest literals in* C_1 *and* C_2, *respectively. The* ordered paramodulation *rule will generate the paramodulant* $(A[p \leftharpoonup t] \mid \alpha \mid \beta)\sigma$, *where* σ *is the mgu of* s *and* A/p.

The paramodulation in Example 7.4.2 is an ordered paramodulation if $f(g(y)) = a$ is maximal in C_1 and $p(g(f(x)))$ is maximal in C_2.

Using the same simplification order, ordered paramodulation can be combined with ordered resolution without losing the refutational completeness. When a set of clauses is just a set of equations (and regarded as a set of rewrite rules), ordered paramodulation is reduced to superposition. Robinson and Wos's work on paramodulation preceded, but apparently did not influence, the work of Knuth and Bendix, who independently formulated superposition and rewriting, called paramodulation and demodulation by Robinson and Wos, and then applied them to the notion of a canonical rewrite system in their 1970 paper.

7.4.2 Simplification Rules

For a theorem prover based on resolution and paramodulation, huge number of new clauses are generated and we need to control redundancy in large search spaces. Restricted strategies for using resolution and paramodulation are effective to reduce redundant clauses, deleting unnecessary clauses in the search for the empty clause is also important to control the search space. As shown before, we may apply the following deletion strategies in a resolution-based theorem prover:

- **pure literal deletion**: Clauses containing pure literals are discarded.
- **tautology deletion**: Tautology clauses are discarded.
- **subsumption deletion**: Subsumed clauses are discarded.
- **unit deletion**: Those literals which are the negation of instances of some unit clauses are discarded from a clause.

Simplification strategies allow us to keep clauses in simplified forms so that multiple copies of equivalent clauses are not needed. For instance, if a clause contains a term s, say $(p[s] \mid q)$, and we have a rewrite rule $s \to t$ or simply $s = t$, then this clause can be rewritten to $(p[t] \mid q)$ by $s = t$. Rewriting is not the only way to obtain $(p[t] \mid q)$, because paramodulation can generate it from $(p[s] \mid q)$ and $s = t$. However, there is no need to keep both $(p[s] \mid q)$ and $(p[t] \mid q)$, because both are logically equivalent under equality. In other words, when $(p[s] \mid q)$ is rewritten to $(p[t] \mid q)$, we keep the latter and throw the former away. That is why we call this kind of strategies a *simplification strategy*.

To work with ordered resolution and paramodulation, we may put a restriction on equality crossing as a simplification rule: **Ordered Equality Crossing**

$$\frac{(s \neq t) \mid A(s)}{(s \neq t) \mid A(t)}$$

if $s \succ t$ for the simplification order $\succ$ used in the ordered paramodulation.

Its soundness follows from Proposition 7.3.4.

Definition 7.4.6. (**contextual rewriting**) *Given a clause $C_1 : (s = t \mid \alpha)$, where $s \succ t$, clause $C_2 : (A \mid \beta)$ can be rewritten by C_1 if p is a position of A, and there exists a substitution σ such that $A/p = s\sigma$, and $\alpha\sigma \subseteq \beta$. The result of this rewriting is $C_2' : (A[p \leftarrow t\sigma] \mid \beta)$.*

Example 7.4.7. Let C_1 be $(f(g(y)) = a \mid q(y)$ and C_2 be $(p(f(g(b))) \mid q(b))$, then C_1 can rewrite C_2 to $(p(a) \mid q(b)$ because $p(f(g(b)))/1 = f(g(b)) = f(g(y))\sigma$, where $\sigma = [y \leftarrow b]$, and $q(y)\sigma$ appears in C_2. □

Note that if α is empty, then C_1 is a positive unit clause and contextual rewriting is reduced ordinary rewriting.

Proposition 7.4.8. (**soundness of contextual rewriting**) *If C_1 rewrites C_2 to C_2' by contextual rewriting, then $E_{ax} \cup \{C_1\} \models C_2 \leftrightarrow C_2'$, where E_{ax} is the equality axioms.*

Proof. Suppose C_1 is $(s = t \mid \alpha)$, $s \succ t$, C_2 is $(A[s] \mid \beta)$, and C_2' is $(A[s] \mid \beta)$. Then C_2' can be obtained from C_1 and C_2 by paramodulation. Switching the role of s and t, C_2' can also be obtained from C_1 and C_2' by paramodulation. The soundness of paramodulation ensures the equivalence of C_2 and C_2' under C_1 and E_{ax}. □

There are two possible extensions to contextual rewriting. In contextual rewriting, clause $C_1\,(s = t \mid \alpha)$ serves as a conditional rewrite rule $s \to t$ **if** $\neg\alpha$. If the second clause C_2 contains a term s' such that $s' = s\sigma$ for some substitution

σ, assuming C_2 is $(A[s'] \mid \beta)$, then s' is replaced by $t\sigma$ if the condition $\neg\alpha$ is true by enforcing $\alpha\sigma \subseteq \beta$. This condition can be replaced by the condition $\neg\beta \models \neg\alpha\sigma$, where $\neg\beta$ is called the *context* of $\neg\alpha\sigma$.

The second extension of contextual rewriting is to treat non-equality literals as rewrite rules. That is, if C_1 is $(A \mid \alpha)$, where A is a maximal non-equality literal in C_1, then we regard C_1 as a rewrite rule $A = \top$ **if** $\neg\alpha$ if A is positive; or $B = \bot$ **if** $\neg\alpha$ if $A = \neg B$. Then C_1 can rewrite atoms in other clauses to $\top$ or $\bot$. If the literals in the other clauses are rewritten to $\top$, this is equivalent to subsumption. If they are rewritten to false, the other clauses will have one less literal. In particular, if α is empty, this is equivalent to unit deletion (Definition 6.2.10).

The soundness of contextual rewriting remains true with the two extensions, thus contextual rewriting can serve as a good simplification rule which covers ordinary rewriting, subsumption, and unit deletion.

Theorem 7.4.9. *Ordered paramodulation combined with ordered resolution, factoring, relaxing, contextual rewriting, and ordered equality crossing is refutationally complete for theorem proving with equality.*

The proof of this theorem follows the same proof line of Theorem 3.3.12: If the empty clause is generated, then the input clauses are unsatisfiable because all inference and simplification rules are sound. If the empty clause can never be generated and the clause set is saturated by the inference rules, then a Herbrand model can be constructed using the simplification order on $T(F)$.

7.4.3 Equality in Prover9

All the inference and simplification rules discussed in this section have been implemented in Prover9, with the exception of contextual rewriting. Rewriting using equations, called *demodulation*, is supported in Prover9. In fact, most examples provided in the distribution of Prover9 contain the equality.

Example 7.4.10. In Example 7.1.5, we are given three equations which specify a free group $(S, *)$. Can we reduce the number of equations to one? What is the minimal size of such a single equation? These are examples of questions interested by mathematicians. Here is an answer to the first question:

$$y * i(z * (((u * i(u)) * i(x * z)) * y)) = x$$

which involves four variables. □

Prover9's input file to this problem is simply the single axiom and the goals will be the three equations in Example 7.1.5, with the only exception that we use here $y * i(y)$ for the identity e.

```
formulas(sos).
y * i(z * (((u * i(u)) * i(x * z)) * y)) = x # label(oneAxiom).
end_of_list.

formulas(goals).
(x * y) * z = x * (y * z) # label(associativity).
x * i(x) = y * i(y)       # label(inverse).
x * (y * i(y)) = x        # label(identity).
end_of_list.
```

It took Prover9 less than a half second to find proofs for all the three equations after generating 1587 clauses. The proof of the associativity is the hardest, which consists of 48 steps after the proofs of the other equations.

The equality symbol "=" is built in; so is "!=", which stands for the negation of equality. The major options involving paramodulation are the following:

```
clear(ordered_para).   % default set
set(ordered_par).

assign(para_lit_limit, n). % default n=-1, range [-1 .. INT_MAX]
```

If $n \neq -1$, each parent in paramodulation can have at most n literals. This option may cause incompleteness of the inference system.

```
set(para_units_only).
clear(para_units_only). % default clear
```

This flag says that both parents for paramodulation must be unit clauses. The only effect of this flag is the same as to assign 1 to the parameter `para_lit_limit`.

The major options involving rewriting are the following:

```
assign(demod_step_limit, n). % default n=1000, range [-1 .. INT_MAX]
```

This parameter limits the number of rewrite steps that are applied to a clause during demodulation. If $n = -1$, there is no limit.

```
assign(demod_size_limit, n). % default n=1000, range [-1 .. INT_MAX]
```

This parameter limits the size (measured as symbol count) of terms as they are demodulated. If any term being demodulated has more than n symbols, demodulation of the clause stops. If $n = -1$, there is no limit.

```
set(back_demod).
clear(back_demod). % default clear
```

If this flag is set, a new rewrite rule will try to rewrite equations in the `usable` and `sos` lists. Prover9 also provides options for non-orientable equations to be rewrite rules under certain conditions and rewriting using these rules are also restricted.

7.5 Mace4: Finite Model Finding in FOL

Mace4 is a software tool for finite model finding in a first-order language. Mace4 is distributed together with Prover9 and was created by William McCune. Prover9 and Mace4 share a first-order language and there is a GUI interface for both of them. Mace4 searches for finite models satisfying the formulas in a first-order language based on the DPLL procedure. If the formula is the denial of some conjecture, any model found by Mace4 are counterexamples to the conjecture. Mace4 is a valuable complement to Prover9, looking for counterexamples before (or at the same time as) using Prover9 to search for a proof. It can also be used to help debugging input clauses and formulas for Prover9.

7.5.1 Use of Mace4

For the most part, Mace4 accepts the same input files as Prover9. If the input file contains commands that Mace4 does not understand, then the argument "-c" must be given to tell Mace4 to ignore those commands. For example, we can run the following two jobs in parallel, with Prover9 looking for a proof, and Mace4 looking for a counterexample.

```
prover9 -f x2.in > x2.prover9.out
mace4 -c -f x2.in > x2.mace4.out
```

Most of the options accepted by Mace4 can be given either on the command line or in the input file. The following command lists the command-line options accepted by Mace4.

```
mace4 -help
```

Mace4 searches for unsorted finite models only. That is, a model has one underlying finite set, called the domain (or the universe), and the members are always $0, 1, ..., n-1$ for a set of size n. The models are the structures which define functions and relations over the domain, as an interpretation to the function and predicate symbols in the formula (using the same symbols). By default, Mace4 starts searching for a structure of domain size 2, and then it increments the size until it succeeds or reaches some limit. The size of the initial domain or the incremental size can be specified by the user.

If a formula contains constants that are natural-numbers, $\{0, 1, ...\}$, Mace4 assumes they are members of the domain of some structure, that is, they are distinct objects; in effect, Mace4 operates under the assumptions $0 \neq 1$, $0 \neq 2$, and so on. To Prover9, natural numbers are just ordinary constants. This is a subtle difference between Prover9 and Mace4. Because Mace4 assumes that natural numbers are members of the domain, if a formula contains a natural number that is out of range ($\geq n$, when searching for a structure of size n), Mace4 will terminate with a fatal error.

Mace4 and Prover9 have the same restrictions on the goal formulas it accepts. Mace4 negates the goals and translates them to clauses in the same way as Prover9. The term "goal" is not particularly intuitive for Mace4 users, because Mace4 does not prove things. It makes more sense, however, when one thinks of Mace4 as searching for a counterexample to the goal.

Mace4 uses the following commands to specify the initial domain size, the maximal domain size, and the increment of the domain size.

```
assign(domain_size, n).    % default n=2, range [2 .. 200]
                           % command-line -n n
assign(iterate_up_to, n). % default n=10, range [-1 .. 200]
                           % command-line -N n
assign(increment, n).      % default n=1, range [1 .. 200]
                           % command-line -i n
```

These three parameter work together to determine the domain sizes to be searched. The search starts for structures of size `domain_size`; if that search fails, the size is incremented, and another search starts. This continues up through the value `iterate_up_to` (or until some other limit terminates the process).

Example 7.5.1. In section 4.3.3, we showed how to find a Latin square with special constraints by a SAT solver. We used the propositional variable $p_{x,y,z}$ to represent the meaning that the cell at row x and column y contains value z. The same relation

between x, y and z can be equally specified by the predicate $p(x, y, z)$. Thus, the five sets of propositional clauses can be specified by the five first-order formulas as follows:

1. $\forall x \forall y \exists z \, p(x, y, z)$.
2. $\forall x \forall y \forall z \forall w \, ((z = w) \mid \overline{p(x, y, z)} \mid \overline{p(x, y, w)})$.
3. $\forall x \forall y \forall z \forall w \, ((y = w) \mid \overline{p(x, y, z)} \mid \overline{p(x, w, z)})$.
4. $\forall x \forall y \forall z \forall w \, ((x = w) \mid \overline{p(x, y, z)} \mid \overline{p(w, y, z)})$.
5. $\forall x \forall y \forall z \forall w \, (\overline{p(y, x, z)} \mid \overline{p(z, y, w)} \mid p(w, y, x))$.

These formulas can be easily converted into clauses and can be specified in Mace4 as the following: □

```
assign(domain_size, 5).
formulas(assumptions).
  p(x, y, f(x, y)).
  (z = w) | -p(x, y, z) | -p(x, y, w).
  (y = w) | -p(x, y, z) | -p(x, w, z).
  (x = w) | -p(x, y, z) | -p(w, y, z).
  -p(y, x, z) | -p(z, y, w) | p(w, y, x).
end_of_list.
```

Note that the function f in the first clause is a Skolem function from $\exists z$. Mace4 will find a model of size 5 for this input and it displays the value of `p(x, y, z)` as a Boolean matrix of 25 rows by 5 column, not a very readable format for a Latin square. From the meaning of $p(x, y, z)$, we may reconstruct the Latin square as follows.

```
0 2 1 4 3
4 1 3 2 0
3 4 2 0 1
1 0 4 3 2
2 3 0 1 4
```

If we specify the Latin square using $x * y = z$ instead of $p(x, y, z)$, the input to Mace4 is the following.

```
assign(domain_size, 5).

formulas(assumptions).
  x * z != y * z | x = y.    % Latin square axioms
  x * y != x * z | y = z.
  x * x = x.                 % idempotent law
  ((y * x) * y) * y = x.     % special constraint
end_of_list.
```

Note that $x * z! = y * z \mid x = y$ specifies that if rows x and y contain the same value at column z, then $x = y$; $x * z! = y * z \mid x = y$ specifies that if column y and z contain the same value at row x, then $y = z$. An alternative specification will use \ and / as the Skolem functions for u and v, respectively, in

$$\forall x, y \exists! u, v \, (x * u = y) \wedge (v * y = x)$$

where $\exists!$ means "there exists uniquely". Now Mace4's input will look like

```
assign(domain_size, 5).

formulas(assumptions).
  % quasigroup axioms (equational)
  x * (x \ y) = y.
  x \ (x * y) = y.
  (x / y) * y = x.
  (x * y) / y = x.

  ((y * x) * y) * y = x. % special constraint
end_of_list.
```

Mace4 will find the following model:

```
interpretation( 5, [number=1, seconds=0], [

 function(*(_,_), [     function(/(_,_), [     function(\(_,_), [
   0, 2, 1, 4, 3,         0, 4, 3, 1, 2,         0, 2, 1, 4, 3,
   3, 1, 4, 0, 2,         4, 1, 0, 2, 3,         3, 1, 4, 0, 2,
   4, 3, 2, 1, 0,         3, 0, 2, 4, 1,         4, 3, 2, 1, 0,
   2, 4, 0, 3, 1,         1, 2, 4, 3, 0,         2, 4, 0, 3, 1,
   1, 0, 3, 2, 4 ]),      2, 3, 1, 0, 4 ]),      1, 0, 3, 2, 4 ])
]).
```

The Latin square defined by $*$ is different from the one obtained by $p(x, y, z)$. If we use the command `assign(max_models, -1)`, Mace4 will find all the six models when the domain size is 5; it will find 120 models when the domain size is 7 (you need to turn off the `auto` command).

7.5.2 Finite Model Finding by SAT Solvers

The working principle of Mace4 is based on the fact that a set C of first-order clauses has a finite model if the propositional formula translated from C is satisfiable. Let $D = \{0, 1, ..., n-1\}$ be the domain of the finite model I, we will do the following to obtain a set of propositional clauses from C.

- For each predicate symbol p/k, define a set of propositional variables $q^p_{x_1,...,x_k}$, $x_i \in D$, such that $q^p_{x_1,...,x_k}$ is true iff $p^I(x_1, ..., x_k)$ is true.
- For each function symbol f/k, define a set of propositional variables $q^f_{x_1,...,x_k,y}$, $x_i, y \in D$, such that $q^f_{x_1,...,x_k,y}$ is true iff $f^I(x_1, ..., x_k) = y$, and add the following clause into C:

  $$(f(x_1, ..., x_k) \neq y_1 \mid f(x_1, ..., x_k) \neq y_2 \mid y_1 = y_2)$$

  where $s \neq t$ stands for $\neg(s = t)$.
- For each clause A of C, if $f(t_1, ..., t_k)$ appears in A and not in the format of $f(x_1, ..., x_k) = y$ (or $y = f(x_1, ..., x_k)$), then replace A by $A[f(t_1, ..., t_k) \leftharpoonup y] \mid f(t_1, ..., t_k) \neq y$, where y is a new variable. This step is called *flattening*. Once this step stops, every atom in C will be either $p(x_1, ..., x_k)$ or $f(x_1, ..., x_k) = y$, where $x_1, ..., x_k, y$ are free variables.
- For each flattened clause A of C, for each variable x of A, for each value d of D, create all the ground instances of A. Let the set of all ground clauses be named as G.
- Finally, for each ground clause in G, replace $p(d_1, ..., d_k)$ by $q^p_{d_1,...,d_k}$ and $f(d_1, ..., d_k) = d$ by $q^f_{d_1,...,d_k,d}$. Let the set of all propositional clause be named as $propo(C)$.

Example 7.5.2. For the equational specification of the Latin square problem, we used three function symbols: $*$, $\backslash$, and $/$. The propositional variables will be $p_{x,y,z}$ for $x * y = z$, $q_{x,y,z}$ for $x\backslash y = z$, and $r_{x,y,z}$ for $x/y = z$; if $|D| = 5$, there will be 375

propositional variables. The flattened clauses from the axioms plus the functional constraints are

$$\begin{array}{ll}
(x\backslash y \neq z \mid x * z = y) & // \ x * (x\backslash y) = y. \\
(x * y \neq z \mid x\backslash z = y) & // \ x\backslash(x * y) = y. \\
(x/y \neq z \mid z * y = x) & // \ (x/y) * y = x. \\
(x * y \neq z \mid z/y = x) & // \ (x * y)/y = x. \\
(y * z \neq z \mid z * y \neq w \mid w * y = x) & // \ ((y * x) * y) * y = x. \\
(x * y \neq z) \mid x * y \neq w \mid z = w) & //\text{“}*\text{”} \text{ is a function} \\
(x\backslash y \neq z) \mid x\backslash y \neq w \mid z = w) & //\text{“}\backslash\text{”} \text{ is a function} \\
(x/y \neq z) \mid x/y \neq w \mid z = w) & //\text{“}/\text{”} \text{ is a function}
\end{array}$$

If $|D| = 5$, a clause with three free variables will generate $5^3 = 125$ ground clauses; a clause with four free variable will generate $5^4 = 625$ ground clauses. The ground instances of the above clauses are easy to convert to propositional clauses. For the literal $(z = w)$, its ground instance becomes $\top$ if we have $z\sigma = w\sigma = d$ and $\bot$ if $z\sigma = d_1 \neq d_2 = w\sigma$. □

While the conversion from a first-order formula to a set of propositional clauses uses the equality "=", the equality axioms (introduced in the first section of this chapter) are not needed for the conversion because they produce only trivial propositional clauses. Only the function axiom is needed in the conversion for each function f:

$$(f(x_1, ..., x_k) \neq y_1 \mid f(x_1, ..., x_k) \neq y_2 \mid y_1 = y_2)$$

Proposition 7.5.3. *A first-order formula C in CNF has a finite model iff the propositional formula propo(C) is satisfiable.*

Proof. Let C be a CNF formula of (P, F, X, Op) and $D = \{0, 1, ..., n-1\}$ be the finite domain.

If *propo*(C) is satisfiable, then *propo*(C) has a model σ and σ defines a relation R^p over D^k for each predicate symbol p/k:

$$R^p = \{\langle d_1, ..., d_k\rangle \mid \sigma(q^p_{d_1,...,d_k}) = 1\}$$

and a function $f^\sigma : D^k \longrightarrow D$ for each function f/k:

$$f^\sigma(d_1, ..., d_k) = d \text{ iff } \sigma(q^f_{d_1,...,d_k,d}) = 1.$$

Then it is ready to check that $I = (D, \{R^p \mid p \in P\}, \{f^m \mid f \in F\})$ is a model of C.

On the other hand, if C has a finite model $I = (D, R, G)$, where D is finite, then we can create the interpretation σ_I for *propo*(C) as follows:

$$\sigma_I(q^p_{d_1,...,d_k}) = 1 \text{ iff } p^I(d_1, ..., d_k) = 1$$

and

$$\sigma_I(q^f_{d_1,...,d_k,d}) = 1 \text{ iff } f^I(d_1, ..., d_k) = d$$

It is ready to check that σ is a model of $propo(C)$. □

To use SAT provers to solve a problem of modest difficulty, we need to write an *encoder* which generates propositional clauses and then feed these clauses into a SAT solver. We also need a *decoder* which translates SAT solver's solution into a solution of the problem. Mace4 releases the burden of writing encoders/decoders by converting automatically the first-order clauses into propositional clauses with the free variables being substituted by all the values in the finite domain and then call a SAT solver to find a model; it is then convert the propositional model into a first-order model.

7.6 Exercise Problems

1. Show that $(f(x_1, ..., x_k) \neq y_1 \mid f(x_1, ..., x_k) \neq y_2 \mid y_1 = y_2)$ from the equality axioms.

2. Write out in first order logic the statements that there are at least, at most, and exactly three elements x such that $A(x)$ holds.

3. Given the following set of ground equations,

$$E = \{I = J,\ K = L,\ A[I] = B[K],\ J = A[J],\ M = B[L]\},$$

   run the Knuth-Bendix completion procedure on E with KBO, assuming $B \succ A \succ M \succ L \succ K \succ J \succ I$ and show how each rewrite rule is generated by the procedure.

4. Complete the proof of case 2.2 in Theorem 7.2.18.

5. Given the definition of *add* in Example 7.3.2, prove by structural induction that $add(x, s(y)) = s(add(x, y))$.

6. Given the definition of *add* in Example 7.3.2, prove by structural induction that $add(x, y) = add(y, x)$ and $mul(x, y) = mul(y, x)$.

7. Given the definitions of *add* and *mul* in Example 7.3.2, prove by structural induction that $add(add(x, y), z) = add(x, add(y, z))$ and $mul(mul(x, y), z) = mul(x, mul(y, z))$.

8. Adding the following function to Example 7.3.2:

$$\begin{array}{rrcl} 9 & exp(x,0) & \to & s(0); \\ 10 & exp(x,s(y)) & \to & mul(x,exp(x,y)). \end{array}$$

where $exp(x,y)$ computes x^y, the exponential function. Prove by structural induction that $exp(x,add(y,z)) = mul(exp(x,y),exp(x,z))$.

9. Adding the following functions to Example 7.3.2:

$$\begin{array}{rrcl} 11 & fib(0) & \to & 0; \\ 12 & fib(s(0)) & \to & s(0); \\ 13 & fib(s(s(x))) & \to & add(fib(s(x)),fib(x)). \\ 14 & sum(0) & \to & 0; \\ 15 & sum(s(x)) & \to & add(sum(x),fib(s(x))). \end{array}$$

where $fib(x)$ stands for the Fibonacci number and $sum(x)$ stands for the summation of the first x Fibonacci numbers. Prove by the structural induction that $sum(x) = pre(fib(s(s(x))))$.

10. Adding the following function to Example 7.3.2:

$$\begin{array}{rrcl} 16 & A(0,y) & \to & s(y); \\ 17 & A(s(x),0) & \to & A(x,s(0)); \\ 18 & A(s(x),s(y)) & \to & A(x,A(s(x),y)). \end{array}$$

where $A(x,y)$ stands for the Ackermann function. Prove by the structural induction that $A(s(0),y) = s(s(y))$ and $A(s(s(0)),y) = s(s(s(add(y,y))))$.

11. Given the definition of rev in Example 7.3.8, prove that $sorted(isort(y)) = \top$ is an inductive theorem by structural induction.

12. Use Prover9 to obtain the set of 10 rewrite rules from the three equations in Example 7.2.8.

13. Use Mace4 to decide if there exists Latin squares of size between 4 and 9 for each of the following constraints (called "short conjugate-orthogonal identities"), respectively.

Code	**Constraint**			**Name**
QG3	$(x*y)*(y*x)$	$=$	x	*Schröder quasigroup*
QG4	$(y*x)*(x*y)$	$=$	x	*Stein's third law*
QG5	$((y*x)*y)*y$	$=$	x	
QG6	$(x*y)*y$	$=$	$x*(x*y)$	*Schröder's first law*
QG7	$(y*x)*y$	$=$	$x*(y*x)$	*Stein's second law*
QG8	$x*(x*y)$	$=$	$y*x$	*Stein's first law*
QG9	$((x*y)*y)*y$	$=$	x	*C_3-quasigroup*

14. Repeat the previous problem when an additional constraint, $x * x = x$, is added into the axiom set.

CHAPTER 8
PROLOG: PROGRAMMING IN LOGIC

Prolog is a programming language whose name comes from programming in logic. Unlike many other programming languages, Prolog comes from first-order logic and is intended primarily as a declarative programming language: the program is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations.

Based upon Robinson's resolution principle, Prolog was first developed and implemented in 1972 by Alain Colmerauer with Philippe Roussel, called "Marseilles Prolog". The Japanese Fifth-Generation Computer Project, announced in 1981, adopted Prolog as a development language, and thereby focused considerable attention on the language and its capabilities. Prolog is the major example of a fourth generation programming language supporting the declarative programming paradigm, and is widely regarded as excellent languages for "exploratory" and "prototype programming".

There are several free implementations of Prolog available today, such as GNU's gprolog and SWI's SWI-prolog, and they are the descendants of Marseille Prolog. The language has been used for theorem proving, expert systems, automated planning, natural language processing, and is a popular tool in artificial intelligence and computational linguistics. Prolog is well-suited for specific tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates.

8.1 Prolog's Working Principle

In Prolog, program logic is expressed in terms of relations, and a computation is initiated by running a query over these relations.

8.1.1 Horn Clauses in Prolog

Typically, relations and queries are constructed by Horn clauses. In section 3.4, we have seen that Horn clauses are clauses with at most one positive literal. In Prolog, Horn clauses are further divided into three types.

- **fact**: a unit positive clause;

- **rule**: a clause with one positive literal and one or more negative literals;
- **query**: a negative clause.

A Prolog program consists of facts and rules, which are collectively called *definite clause*. For a rule clause $(A \mid \overline{B_1} \mid ... \mid \overline{B_n})$, where $A, B_1, ..., B_n$, $n \geq 1$, are atomic formulas, it is written in Prolog as $A :- B_1, ..., B_n$. This format looks closely to the equivalent formula

$$B_1 \wedge \cdots \wedge B_n \rightarrow A.$$

In other words, the symbol $:-$ is the inverse of $\rightarrow$ and the commas are $\wedge$ in Prolog. For a rule $H :- B$, H is the *head* and B is the *body* of the rule. It is read as "H is true if B is true". Clauses with an empty body are facts, which do not need the symbol ":-" or we may write it as "H :- true" in Prolog.

Negative clauses like $(\overline{B_1} \mid ... \mid \overline{B_n})$ are Prolog queries and denoted by `?-` $B_1, ..., B_n$, which resembles the equivalent formula $\neg(B_1 \wedge \cdots \wedge B_n)$.

Example 8.1.1. Suppose we want to study the family tree of a big family, we may state the father-child and mother-child relations by facts, and define other relations by rules. Here is a Prolog program. □

```
mom(rose, john).   % Rose is the mother of John
mom(rose, bob).
mom(rose, ted).
papa(joe, john).   % Joe is the father of John
papa(joe, bob).
papa(joe, ted).
papa(ted, patrick).
papa(john, caroline).
papa(bob, kathleen).
papa(bob, joseph).
papa(bob, mary).
parent(X, Y) :- papa(X, Y).  % X is parent of Y if X is papa of Y.
parent(X, Y) :- mom(X, Y).   % X is parent of Y if X is mom of Y.
grandpa(X, Y) :- papa(X, Z), parent(Z, Y).
grandmo(X, Y) :- mom(X, Z), parent(Z, Y).
```

From this example, we can see that variables in Prolog are identifies whose first character is a capital letter. Once the above program is loaded into Prolog, we may issue queries after the symbol `?-`.

To query who are the parents of `ted`, we type `parent(X, ted).` after the symbol `?-`. Prolog will answer with `X = joe`. If you type ";", Prolog will give you another answer: `X = rose`. If you hit "return", you will terminate the query.

To query the sibling relation, we may define the sibling relation as a rule,

```
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X =\= Y.
                 % X and Y share parent Z.
```

where $\backslash =$ stands for "not equal", and issue the query `?- sibling(X, Y).` On the other hand, we may use the query

```
?- parent(Z, X), parent(Z, Y), X =\= Y.
```

This query also prints out the common parent Z of X and Y. For instance the first answer found by Prolog is

```
X = john, Y = bob, Z = joe.
```

8.1.2 Resolution Proofs in Prolog

In Prolog, associated with every successful query, there exists a resolution proof. For instance, for the query `?- parent(X, ted)`, we have an answer `X = joe`. The corresponding resolution proof is given below.

	Prolog format	clausal format	
1	`papa(joe, ted).`	$(f(joe, ted))$	// input
2	`parent(X, Y) :- papa(X, Y).`	$(p(X,Y) \mid \overline{f(X,Y)})$	// input
3	`?- parent(X, ted).`	$(\overline{p(X, ted)})$	// query
4	`?- papa(X, ted).`	$(\overline{f(X, ted)})$	// resolvent of 2 and 3
5	$\perp$.	$\perp$	// resolvent of 1 and 4.

In the last resolution, the mgu is `X` $\mapsto$ `joe`, which is the answer of the query. The query comes from the negation of the formula $\exists X\, parent(X, ted)$, which is a logical consequence of the input clauses.

As another example, for the query `parent(Z, X), parent(Z, Y), X` $\bar{\texttt{Y}}$, the corresponding resolution proof is the following.

```
1 papa(joe, john).                          // input
2 papa(joe, bob).                           // input
3 parent(X, Y) :- papa(X, Y).               // input
4 ?- parent(Z, X), parent(Z, Y), X \= Y.    // query
```

```
5 ?- papa(Z, X), parent(Z, Y), X \= Y.   // resolvent of 3 and 4
6 ?- parent(joe, Y), john \= Y.          // resolvent of 1 and 5
7 ?- papa(joe, Y), john \= Y.            // resolvent of 3 and 6
8 ?- john \= bob.                        // resolvent of 2 and 7
9 ?- ⊤.                                  // ?- ⊤ is ⊥, the empty clause
```

The mgus used in the resolutions provide the solution of the query: Resolution between 1 and 5 provides `X = john, Z = joe` and resolution between 2 and 7 provides `Y = bob`.

Both above resolution proofs are a negative, input and linear resolution proof. That is, every resolution includes a negative clause and an input clause as parents, and the latest resolvent (which is negative) is used in the next resolution. This is true for every resolution proof found by Prolog. Corollary 3.4.11 claims that, for propositional Horn clauses, if there exists only one negative clause, then the negative, input and linear resolution is complete for Horn clauses. This result can be extended to first-order clauses. In other words, the negative, input and linear resolution strategy is complete for first-order Horn clauses.

Thus, given a query, the Prolog engine attempts to find a resolution refutation of the query. If the empty clause can be derived, i.e., an instantiation for all free variables is found that makes the input clauses plus the query false, it follows that the negation of the query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog particularly useful for symbolic computation and language parsing applications.

Example 8.1.2. Suppose we are asked to write a program to convert a propositional formula into negation normal form. This would be a week-long exercise using a conventional programming language. Using Prolog, it is an hour-long exercise. Suppose we use only three propositional variables, say `p, q, r`, and the logical operators are "n" for ¬, "a" for ∧, "o" for ∨, "i" for →, and "e" for ↔, the Prolog program is given below: □

```
pvar(p).   % definition of propositional variables
pvar(q).
pvar(r).

literal(n(L)) :- pvar(L). % definition of literals
literal(L) :- pvar(L).

nnf(n(n(X)), NNF) :- nnf(X, NNF).  % double negation
```

```
nnf(a(A, B), a(N1, N2)) :- nnf(A, N1), nnf(B, N2).  % and
nnf(n(a(A, B)), o(N1, N2)) :-  nnf(n(A), N1), nnf(n(B), N2).

nnf(o(A, B), o(N1, N2)) :- nnf(A, N1), nnf(B, N2).  % or
nnf(n(o(A, B)), a(N1, N2)) :- nnf(n(A), N1), nnf(n(B), N2).

nnf(i(A, B), o(N1, N2)) :- nnf(n(A), N1), nnf(B, N2).  % implication
nnf(n(i(A, B)), a(N1, N2)) :- nnf(A, N1), nnf(n(B), N2).

nnf(e(A, B), o(a(NA, NB), a(NnA, NnB))) :-     % equivalence
       nnf(A, NA), nnf(n(A), NnA), nnf(B, NB), nnf(n(B), NnB).
nnf(n(e(A, B)), o(a(NA, NnB), a(NnA, NB))) :-
       nnf(A, NA), nnf(n(A), NnA), nnf(B, NB), nnf(n(B), NnB).

nnf(X, X) :- literal(X).      % NNF of literal is itself
```

The following is a query to the above program:

```
?- nnf(n(o(n(a(p,q)),e(p,q))), X).
X = a(a(p,q),o(a(p,n(q)),a(n(p),q))) ?
```

8.1.3 A Goal-Reduction Procedure

The Prolog engine can be regarded as a goal-reduction process: For each G in the original query, the engine will search for a rule in the Prolog program whose head unifies with G and recursively apply the engine to the subgoals in the body of the rule. When all subgoals are solved, or when the body is empty, goal G is considered as *solved* and a substitution for the variables of G is returned; otherwise, the label "fail" is returned.

Procedure 8.1.3. Assuming P is a Prolog program, procedure *engine*1 takes a single goal G as input and returns a substitution as output if one rule of P can solve G; otherwise, $fail$ is returned. Procedure *engine* takes a list Gs of goals as input and returns a substitution as output if every goal of Gs is solved by P.

proc *engine*1(G)
1 **for** $(H :- B) \in P$ **do**
2 **if** (H and G are unifiable with mgu σ)
3 $\theta := engine(B\sigma)$;

```
4          if θ ≠ fail return σθ;
5     return fail;

proc engine(Gs)
1     σ = θ := ∅;
2     for A ∈ Gs do
3          σ := engine1(Aθ);
4          if σ = fail break; else θ := θσ;
5     if σ = fail return fail; else return θ;
```

We assume that B is empty if $H :- B$ is a fact. The first call is *engine*(Qs), where Qs are the goals in a query.

Obviously, *engine*1 and *engine* are mutually recursive procedures. If we examine the recursion tree of *engine*1 and *engine*, the links going out of the *engine*1 node are labeled by rules of P; and the links going out of the *engine* node are labeled by a goal in Gs. The *engine*1 node is an $\vee$-node: if one of its children succeeds, it will succeed. The *engine* node is an $\wedge$-node: if one of its children fails, it will fail. In other words, the recursion tree of *engine* and *engine*1 is an $\wedge$-$\vee$ tree.

Procedure *engine*1 as given intends to find one answer; if more answers are desired, then line 4 of *engine*1 will remember which rule is used to get σ and will use the next rule when more solutions are desired. In this case, the *engine*1 node may have more than one successful child.

A leaf node in the $\wedge$-$\vee$ tree is said to be *ok* if (a) it is an *engine* node and Gs is empty or (b) it is an *engine*1 node and G is a built-in atom evaluated to be true. An internal node in the $\wedge$-$\vee$ tree is said to be *ok* if (c) it is an *engine* node and every child node is ok, or (d) it is an *engine*1 node and one of its children is ok.

In this case, the $\wedge$-$\vee$ tree has a solution iff its root node is ok and it represents all the solutions dictated by the $\vee$ nodes.

Example 8.1.4. The following program implements a depth-first search procedure for a directed graph:

```
% s(u, v): directed edge (u, v)
1: s(a,b).
2: s(a,c).
3: s(b,d).
4: s(c,g).
5: s(d,g).
6: goal(e).  % goal(x): x is a goal
```

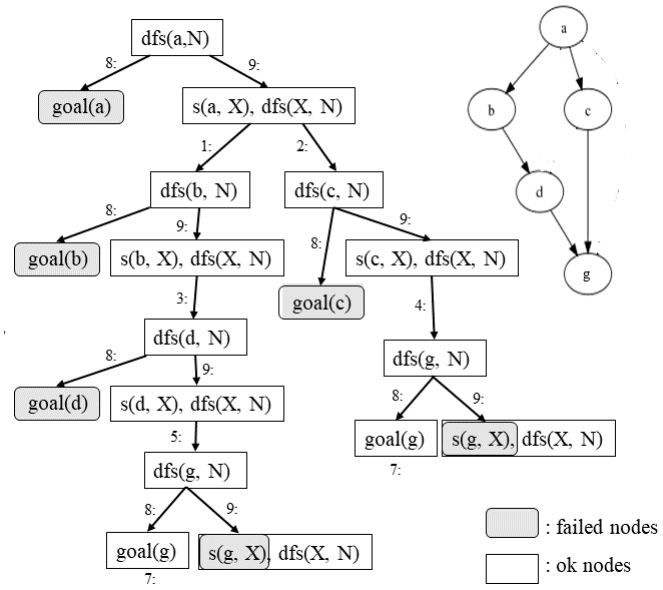


Figure 8.1.1: $\vee$-nodes in the $\wedge$-$\vee$ tree for the query `dfs(a, N)`.

```
7: goal(g).
8: dfs(N, N) :- goal(N).  % dfs(startNode, solution)
9: dfs(N, l(N, R)) :-  s(N, X),  dfs(X, R).
```

There are two solutions to the query `?- dfs(a, N)`. That is, `N = l(a, l(b, l(d, g)))` and `N = l(a, l(c, g))`, where a term like $l(a, l(c, g))$ represents a path `(a, c, g)` in the graph. Later, after introducing the list, the same path can be represented by `[a, c, g]`. Figure 8.1.1 shows the $\vee$-nodes in the search tree for the query `dfs(a, N)`; the $\wedge$-nodes are implicit. The shaded nodes are the *failed* nodes; the other nodes are *ok* nodes. □

Note that the two recursive procedures *engine* and *engine*1 just illustrate the goal-reduction process; the actual Prolog engine is an interactive procedure and uses a stack to save all the information in a search path of the $\wedge$-$\vee$ tree.

Procedure *engine*1 shows that the order of rules in P is important. Procedure *engine* shows that the order of atomic formulas in the body of a rule is important. In other words, the links in the $\wedge$-$\vee$ tree are ordered and it implements a special case of the negative, input and linear resolution strategy: the literals in a rule are handled from left to right, and the commas, which represent $\wedge$ in logic, are not commutative. The input clauses are tried to unify with a given goal from first to last. From the viewpoint of search strategies, this is the depth-first search in the $\wedge$-$\vee$ tree.

Example 8.1.5. Adding the following rules to the Prolog program in Example 8.1.1:

```
descen(X, Y) :- parent(Y, X). % X is a descendant of Y
descen(X, Z) :- parent(Y, X), descen(Y, Z).
```

The last rule is different from the previous rules in that `descen` is recursively defined. If we ask "who are the descendants of Joe", the query is `?- descen(X, joe).` Prolog will find that `X = john, bob, ted, patrick, caroline, kathleen, joseph,` and `mary`, in the given order. For the first answer, i.e., `X = john`, the recursive calls of *engine* and *engine*1 go as follows:

engine([`descen(X, joe)`])
    *engine*1(`descen(X, joe)`)
        engine([`parent(joe, X)`])
            *engine*1(`parent(joe, X)`])
                engine([`papa(joe, X)`])
                    *engine*1(`papa(joe, X)`) **return** `X = john`
...
engine([`descen(X, joe)`]) **return** `X = john`

For the last answer, i.e., `X = mary`, the recursive calls go as follows (the calls to *engine* are ignored if the list contains a single goal).

*engine*1(`descen(X, joe)`)
    engine([`parent(Y, X), descen(Y, joe)`])
        *engine*1(`parent(Y, X)`)
            *engine*1(`papa(Y, X)`) **return** {`X = mary, Y = bob`}
        *engine*1(`descen(bob, joe)`)
            *engine*1(`parent(bob, joe)`)
                *engine*1(`papa(bob, joe)`) **return** { }
...
    engine([`parent(Y, X), descen(Y, joe)`]) **return** {`X = mary, Y = bob`}
*engine*1(`descen(X, joe)`) **return** {`X = mary, Y = bob`}

However, if we change the last rule to a logically equivalent one:

```
descen(X, Z) :- descen(Y, Z), parent(Y, X).
```

The same query, `?- descen(X, joe)`, will produce `X = john, bob, ted, caroline, kathleen, joseph, mary`, and `patrick`, the same set of eight answers in different orders, and will cause Prolog to crash if you ask for more answers. If we switch the order of the last two clauses, i.e., the last two clauses are

```
descen(X, Z) :- descen(Y, Z), parent(Y, X).
descen(X, Y) :- parent(Y, X).
```

Then the same query will cause Prolog to crash without generating any answer. The first-order logic cannot explain the difference because these two clauses are logically equivalent. The procedures *engine* and *engine*1 can explain the difference perfectly:

The initial goal list is `[descen(X, joe)]` in all the three cases. In the last case, the rule `descen(X, Z) :- descen(Y, Z), parent(Y, X)` is applied first, and the recursive calls of *engine* and *engine*1 go as follows.

*engine*1(`descen(X, joe)`)
  engine([`descen(Y, joe), parent(Y, X)`])
    *engine*1(`descen(Y, joe)`)
      engine([`descen(Y1, joe), parent(Y1, Y)`])
        *engine*1(`descen(Y1, joe)`)
          engine([`descen(Y2, joe), parent(Y2, Y1)`])
...

The unsolved goal list contains a long sequence of goals like

```
..., parent(Y3, Y2), parent(Y2, Y1), parent(Y1, Y), parent(Y, X)
```

until Prolog runs out of memory. Here we changed the variable names because each clause is assumed to have different variables from others. □

The above example shows that Prolog uses a depth-first strategy, which is not a fair strategy, to implement resolution, and becomes an incomplete theorem prover for Horn clauses. As a Prolog programmer, we need to be careful to avoid the pitfall of infinite loops associated with the depth-first strategy. We will address this issue in the next section.

Example 8.1.6. Hanoi's tower asks to move n disks from peg A to peg B using peg C as an auxiliary holding peg. At no time can a larger disk be placed upon a smaller disk. The Prolog code contains two predicate definitions: `move` writes out the move information; `hanoi(N,A,B,C)` checks if $N = 1$, then moves disk 1 from `A` to `B`; otherwise, it tries to move the top `N-1` disks from `X` to `C`; move bottom disk `N` from `X` to `B`; and then `N-1` disks from `C` to `B`.

```
move(N,A,B) :-       % print a move info
   write('Move disk '), write(N), write(' from '),
   write(A), write(' to '), write(B), nl.

hanoi(1,A,B,C) :-    % disk 1 can move freely from A to B
   move(1,A,B).
hanoi(N,A,B,C) :-    % To move N disks from A to B,
   N>1, M is N-1,    % where N > 1,
   hanoi(M,A,C,B),   % we move the top N-1 disks from A to C
   move(N,A,B),      % then move disk N, the largest, from A to B
   hanoi(M,C,B,A).   % then move the N-1 disks from C to B
```

```
| ?-  hanoi(3, a, b, c).
Move disk 1 from a to b
Move disk 2 from a to c
Move disk 1 from b to c
Move disk 3 from a to b
Move disk 1 from c to a
Move disk 2 from c to b
Move disk 1 from a to b
```

The code comes directly from the description of the solution. It involves two recursive calls of `hanoi`. The recursive calls will terminate because the first argument of `hanoi` is reduced by one in each successive call. □

8.2 Prolog's Data Types

As a programming language, Prolog has several built-in data types, such as numbers (floats or integers), strings, and lists, and associated functions. These data types are subtypes of a supertype, called *terms*. In this view, *term* is the only data type language in Prolog. Terms are either atoms, numbers, variables or compound terms. Atomic formulas in Prolog are called *callable terms*.

8.2.1 Atoms, Numbers, and Variables

We have been using "atom" for "atomic formula"; in this section, we will use "atomic formula" explicitly because "atom" has special meaning in Prolog: an *atom* is either an identifier other than variables or a single-quoted string of characters.

Examples of atoms include `x, red, 'Taco'`, and `'some atom'`. An atom is a general-purpose name with no inherent meaning.

Prolog provides the predicate `atom/1`, where `atomX` is true iff `X` is an atom. For example, `atom('some atom')` is true; `atom(123)` is false.

Numbers can be floats or integers. The range of numbers that can be used is likely to be dependent on the number of bits (amount of computer memory) used to represent the number. Most of the major Prolog systems support arbitrary length integer numbers. The treatment of float numbers in Prolog is likely to vary from implementation to implementation; float numbers are not all that heavily used in Prolog programs, as the emphasis in Prolog is on symbol manipulation.

Prolog provides all the usually predicates on numbers, such as `>/2, </2, =</2, >=/2`, etc., and arithmetic operations, such as `-/1, +/1, +/2, -/2, */2, //2, mod/2, rem/2, gcd/2, lcm/2, abs/1, sign/1, max/2, min/2, truncate/1, floor/1, ceiling/1, sqrt/1, sin/1, cos/1`, etc. You use `=:=/2` to check if two numbers are equal; and `=\=/2` for not equal. The symbol `=/2` is reserved for unification: `s = t` returns true if `s` and `t` are unifiable.

Prolog also provides predicate `number/1` to check if a term is a number or not; `integer/1` for being an integer; `float/1` for being a float number. For example, `float(1.0)` returns true; `float(1)` returns false. Note that `integer(1+1)` returns false as `1+1` is a compound term. If we check the value of `1+1<3`, Prolog will return true because $1 + 1$ will be evaluated to 2 before $<$ is called on $(2, 3)$.

Variables are denoted by an identifier consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. The underscore alone, "`_`", denotes a nameless distinct variable. A nameless variable is used when we do not care about the value of the variable. For example, if we define the multiplication relation `mul(X, Y, Z)`, where `Z = X*Y`, the first rule can be `mul(0, _, 0)`, where the second position is a nameless variable.

Prolog variables closely resemble variables in logic in that it can be instantiated by arbitrary terms through unification. If a variable is instantiated, we say "the variable is bounded". Some operations require the involved variables are bounded. For instance, to show

```
descen(X, Z) :- parent(Y, X), descen(Y, Z).
```

is a terminating rule in Example 8.1.5, we need to check that the bindings of `X, Y` and `Z`. The predicate `parent(Y, X)` ensures that `X` and `Y` will be bounded to atoms when we reach `descen(Y, Z)`. When these variables are bounded by atoms, the atom bounded to `Y` is one step closer than that of `X` to `Z` in the descendant relation, this meaning defines a well-founded ordering $>$ such that `X` $>$ `Y` when we compare `descen(X, Z)` and `descen(Y, Z)`.

In Prolog, there is a special infix predicate called `is/2`, which was used in the Hanoi example. It resembles the assignment statement of a variable by a numeric expression in a conventional programming language. For example, the value of `X is 1+1` is true with `X` is instantiated with `2`. That is, Prolog will evaluate the right side of `is`, if it is a number, that number is bounded to the variable on the left side of `is`; otherwise it will abort with error. In contrast, `X = 1+1` will also return true, but it will bound the variable `X` to the compound term `1+1`. As another example, "`X = a`" will return true and "`X is a`" will be aborted, because `a` is not a number.

8.2.2 Compound Terms and Lists

A *compound term* is composed of a predicate or function symbol, called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An identifier can be regarded as a compound term with arity zero.

There is a built-in predicate called `functor/3` which allows us to extract the functor and its arity from a compound term:

```
?- functor(f(a, b), F, N).
F = f, N = 2
```

Compound terms are indispensable for representing various data structures in conventional languages. For example, to represent a binary tree where each non-empty tree node contains a key, we may use the compound term `node(key, leftChild, rightChild)`. We can define easily predicate `tree/1`, which checks if its argument is a tree, and predicate `dfs/2`, which tries to find a key by depth-first search in a tree.

```
tree(nil).                      % nil denotes the empty tree
tree(node(_, Left, Right)) :- tree(Left), tree(Right).

dfs(node(K, _, _), K) :- write('found '), write(K).
dfs(node(_, L, _), K) :- dfs(L, K).          % go to Left branch
dfs(node(_, _, R), K) :- dfs(R, K).          % go to Right branch
```

In the definition of `dfs`, we used "_" twice in each rule; each "_" represents a unique variable.

One special member of compound terms is *list*, which is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas, or

in the case of the empty list, by `[]`. Here are some lists of three elements, `[1,2,3]`, `[red,green,blue]`, or `[34,tom,[2,3]]`. Lists are the replacement of arrays in conventional programming languages.

For a non-empty list, the first member of a list is separated from the rest members by the symbol `|`. For example, the result of `[X | Y] = [1,2,3]` is `X = 1` and `Y = [2, 3]` and the result of `[X | Y] = [1]` is `X = 1` and `Y = []`.

Two lists unify if they are the same length and all their elements unify pairwisely.

```
|?-[a,B,c,D]=[A,b,C,d].
A = a,
B = b,
C = c,
D = d ?
yes
```

```
|?-[(a+X),(Y+b)]=[(W+c),(d+b)].
W = a,
X = c,
Y = d?
yes
```

```
|?-[a,b,c,d]=[Head|Tail].
Head = a,
Tail = [b,c,d]?
yes
```

```
|?-[a,b,c,d]=[X,Y|Z].
X = a,
Y = b,
Z = [c,d];
yes
```

8.2.3 Popular Predicates over Lists

Most Prolog implementations provide efficient implementations of the following predicates over lists. We also are providing hypothetical Prolog codes of these predicates, even though they are not identical to actual implementations.

- `list/1`: Check if its argument is a list.

```
list([]).
list([_ | X]) :- list(X).
```

- `member/2`: Check if the first argument is a member of the second argument.

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

- `append/3`: Check if the third list is the concatenation of the first two lists.

```
append([], Y, Y).
append([F|X], Y, [F|Z]) :- append(X, Y, Z).
```

- `select/3`: `select(X, List1, List2)` is true when `List1`, with `X` removed, results in `List2`.

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

  For instance, `select(X, [a,b], Z)` has two solutions: (1) `X = a` and `Z = [b]`; (2) `X = b` and `Z = [a]`. An alternative implementation of `select` is to use `append`:

  ```
  select(E, L, R) :- append(X, [E|Y], L), append(X, Y, R).
  ```

- `delete/3`: `delete(X, List1, List2)` is true when `List1`, with all members matching to `X` are removed, results in `List2`.

  ```
  delete(_, [], []).
  delete(X, [X|Y], Z) :- delete(X, Y, Z).
  delete(X, [F|Y], [F|Z]) :- X \= F, delete(X, Y, Z).
  ```

  where `\=` means "not unifiable". For instance, `delete(X, [a,b], Z)` has only one solution: `X = a, Z = [b]`.[1] This shows the difference of `select` and `delete`.

- `reverse/2`: `reverse(List1, List2)` is true when the elements of `List2` are in reverse order compared to `List1`.

  ```
  reverse([], []).
  reverse([F|X], Z) :- reverse(X, Y), append(Y, [F], Z).
  ```

- `permutation/2`: `permutation(List1, List2)` is true when `List2` is a permutation of `List1`.

  ```
  permutation([], []).
  permutation([F|X], Y) :- permutation(X, Z), select(F, Y, Z).
  ```

[1] For the same query, the built-in `delete` in `gprolog` gives an infinite number of solutions.

- `length/2`: `length(List, N)` is true when the length of `List` is `N`.

```
length([], 0).
length([_|X], N) :- length(X, M), N is M+1.
```

One common feature of these predicates is that they are all defined recursively. The termination of these definitions is easy to establish as one of the arguments to the recursive calls goes smaller, or we may show that the head of each rule is strictly greater than its body under a simplification ordering.

Due to the relational nature of many built-in predicates, they can typically be used in several directions. For example, `append/3` can be used both to append two lists (`append(ListA, ListB, X)` given lists `ListA` and `ListB`) as well as to split a given list into parts (`append(X, Y, List)`, given `List`). For instance, the query `append(X, Y, [1,2,3])` will produce the following four answers:

```
X = [], Y = [1,2,3]
X = [1], Y = [2,3]
X = [1,2], Y = [3]
X = [1,2,3], Y = []
```

For this reason, a comparatively small set of library predicates suffices for many Prolog programs. Note that the suggested implementation for `length` can compute the answer `N = 3` from `length([a,b,c], N)`, but cannot generate a list of 3 placeholders, `L = [_1, _2, _3]`, from `length(L, 3)`. Some version of Prolog, e.g., SWI Prolog, can do so.

For some applications, we need to obtain a set $S = \{x \mid A(x)\}$, that is, S is a set of elements satisfying the condition $A(x)$. It is not easy to write a Prolog program which generates such a set. Fortunately, in most implementations of Prolog, such a set can be obtained by calling `bagof(X, Goal(X), S)`. For example, in Example 8.1.1, we may define `children(F) = { C | papa(F, C) }`. A query like the following will give us a satisfactory answer:

```
?- bagof(C, papa(Father, C), Children).
Father = bob, Children = [kathleen,joseph,mary] ? ;
Father = joe, Children = [john,bob,ted] ? ;
Father = john, Children = [caroline] ? ;
Father = ted, Children = [patrick]
```

Some Prolog also provides the set operations, such as `intersection/3, union/3, subtract/3, subset/2`, when a list is viewed as a set.

In Prolog, a string is a sequence of characters surrounded by double quotes and is equivalent to a list of numeric character codes. One common mistake is to use strings in the `write` predicate instead of a single-quoted sequence of characters (an atom).

8.2.4 Sorting Algorithms in Prolog

To illustrate how an algorithm is implemented in Prolog, we show below the Prolog code of insertion sort and quick-sort.

Example 8.2.1. The following program defines four predicates for the implementation of insertion sort and quick-sort: `isort/2, insert/3, split/4`, and `qsort/2`.

```
% isort(L, R): R is the output of L by insert sort
isort([], []).
isort([X|L], R) :-
  isort(L, T),            % sort L into T
  insert(X, T, R).        % insert X into T to obtain R.

% insert(X, L, R): insert X into list L and R is the result.
insert(X, [], [X]).
insert(X, [Y|L], [X, Y|L]) :- X =< Y.
insert(X, [Y|L], [Y|R]) :- X > Y, insert(X, L, R).

% split(P, I, S, B): split the input list I into S and B by pivot P.
split(_, [], [], []).
split(P, [H|I], [H|S], B) :- H =< P, split(P, I, S, B).
split(P, [H|I], S, [H|B]) :- H > P, split(P, I, S, B).

% qsort(L, R): L is the input list and R is the output by quicksort
qsort([], []).
qsort([X|Y], Z) :- split(X, Y, S, B),
     qsort(S, S1), qsort(B, B1), append(S1, [X|B1], Z).
```

□

The above code is lucid and neat, easy to understand and maintain. This is typical feature of a declarative programming language. Of course, its execution is not as efficient as conventional programming languages. That is why Prolog is used mostly for prototype programming, and for popular functions, we have efficient built-in implementations.

Another feature of declarative programming is that the correctness of the algorithm can be verified. For sorting, the correctness is ensured by the two properties: (1) the output is sorted; (2) the output is a permutation of the input list. In Prolog, (2) is a built-in function and we can define (1) with easy.

Example 8.2.2. The following Prolog program defines the predicate `sorted`(L), which returns true iff L is a sorted list, and some utility predicates for the verification purpose.

```
sorted([]).
sorted([_]).
sorted([X,Y|L]) :- X=<Y, sorted([Y|L]).

verifyIsort(X, Y) :- isort(X, Y), permutation(X, Y), sorted(Y).
verifyQsort(X, Y) :- qsort(X, Y), permutation(X, Y), sorted(Y).
slowsort(X, Y) :- permutation(X, Y), sorted(Y).
```

In the above code, the first three clauses define `sorted/1`. The next two clauses illustrate the verification of `isort` and `qsort`. The last clause shows how the requirement can be the code itself. The code of `permutation/2` can be found in section 8.2.3. □

Most Prolog implementations provide two versions of predicates for sorting: `sort/2`, which removes duplicate elements, and `msort/2`, which does not remove duplicates. For instance, the query `sort([b,a,c,a], X)` will produce `X = [a,b,c]`, while `msort([b,a,c,a], X)` produces `X = [a,a,b,c]`. These built-in predicates run faster than the user defined sorting algorithms.

8.3 Recursion in Prolog

As is commonly the case in most programming tasks, we often wish to repeatedly perform some operation either over a whole collection of data, or until a certain point is reached. Conventional programming languages provide various loop controls for performing such tasks. The only way we can do this in Prolog is by recursion. In section 8.2.3, we have seen many Prolog programs which contain recursive definitions. Recursion allows us to write clear and elegant code. Data such formulas or lists are often recursively and best processed recursively. Recursion allows Prolog to perform complex search of a problem space without any dedicated algorithms. Without recursion, Prolog goes nowhere. To master Prolog, we have to learn how to use recursion correctly and efficiently.

Any recursive definition, whether in Prolog or some other language needs two things: *base cases* and *recursive cases*. The base cases specify a condition of when the recursion terminates. Without this the recursion would never stop! For example, the base case of `append(X, Y, Z)` is `append([], Y, Y)` when `X=[]`. Base cases almost always come before recursive cases. Recursive cases ask a program to handle a similar problem of smaller size. For example, `append([U|X], Y, [U|Z]) :- append(X, Y, Z).`

8.3.1 Program Termination

The danger of recursion is that the program may loop forever and never terminate. The termination problem is more complicated in Prolog than in conventional programs because the input to a Prolog program may contain variables and we may ask for multiple solutions from a single query.

Definition 8.3.1. *A Prolog program P is said to have* first termination *on an input x if P terminates with either no solution or the first solution on x. P is said have* last termination *on x if P terminates after all solutions of x are generated. If P has first (last) termination on any ground input x, then P is said to have* first (last) ground termination.

Example 8.3.2. Given the following program P,

```
ancestor(tom, jack).
ancestor(X, Y) :- ancestor(X, Z), ancestor(Z, Y).
```

P has first termination on the query `?- ancestor(A, B)` with the solution `A=tom, B=jack`, but not on `?- ancestor(jerry, tom)`, as the second clause will be used forever during the search. Since the second input is ground, P has no first ground termination. Since last termination implies first termination, P has no last ground termination. □

Prolog is a relational programming language, and relations can hold between multiple entities that are reported on backtracking. Therefore, first termination does not fully characterize the procedural behavior of a Prolog query and we need the concept of last termination. We may use the termination of the query `?- Q, false.` to check if `Q` has last termination, because all the solutions of `Q` must be generated before the query terminates.

Consider the `member/2` program in section 8.2.3, the program has ground termination when the second argument of *member* is ground. However, the query `?- member(X, Y), false.` will fail with a stack overflow error, thus showing that `member(X, Y)` has no last termination.

Example 8.3.3. Given a propositional formula A in negation normal form (NNF), we may use the following program to decide if A is satisfiable, assuming the formula uses Prolog variables for propositional variables, and `and`, `or`, `not` for $\wedge$, $\vee$, $\neg$, respectively.

```
sat(true).                % positive literal
sat(not(false)).          % negative literal
sat(or(X, _)) :- sat(X).
sat(or(_, Y)) :- sat(Y).
sat(and(X, Y)) :- sat(X), sat(Y).

test1(X, Y) :- sat(and(not(X), X)).    % test examples
test2(X, Y) :- sat(and(X, not(Y))).
test3(X, Y) :- sat(and(X, or(not(X), Y))).
```

The *NNF* formulas can be defined recursively by the BNF as follows:

$$\langle NNF \rangle ::= \langle Literal \rangle \mid \langle NNF \rangle \texttt{ or } \langle NNF \rangle \mid \langle NNF \rangle \texttt{ and } \langle NNF \rangle$$

The first two clauses are the base cases of `sat` which handle literals, and the next three clauses are the recursive cases which handle `or` and `and`.

It is easy to see that the argument to the recursive call is strictly less than the argument in the head of each rule. Thus, we expect that the program will terminate with the test examples. For the three test examples, the query `?- test1(X, Y)` will say `no`, thus `sat` has last termination on `and(not(X), X)`. For the second query, `?- test2(X, Y)`, Prolog will give the first solution `X=true, Y=false`. The third query `?- test3(X, Y)` will produce the first solution `X=true, Y=true`. Thus, `sat` has first termination on both inputs.

However, if we look for more answers from `?- test2(X, Y)`, we will see the following answers:

```
X = not(false), Y = false
X = or(true,_), Y = false
X = or(not(false),_), Y = false
X = or(or(true,_),_), Y = false
X = or(or(not(false),_),_), Y = false
X = or(or(or(true,_),_),_), Y = false
X = or(or(or(not(false),_),_),_), Y = false
...
```

These answers are correct in Prolog, but not correct for propositional satisfiability (which asks for the models of a formula). If you ask for more answers from `?- test3(X, Y)`, you will see an infinite number of answers, too. Thus, `sat` has last termination neither on `and(X, not(Y))` nor on `and(X, or(not(X), Y))`. Ground termination does not make sense for `sat` because propositional variables are represented by variables. □

The problem illustrated by the above example is that variables appearing in the arguments of a recursive program are not bounded and they can be substituted by terms containing other unbound variables, thus creating an infinite number of possibilities. This is the same reason why `member(X, Y)` has no last termination.

8.3.2 Focused Recursion

A poorly definition of a recursive predicate may cause the program to loop. In the next chapter, it is shown that checking if a Prolog program terminates on a given input is a undecidable problem. Thus, people are interested in sufficient conditions which ensure the termination of a Prolog program. Focused recursion is one of such conditions.

Definition 8.3.4. *A recursive predicate is said to be* focused *on an input t_0 if there exists a well-founded order $\succ$ such that $t_1 \succ t2 \succ \cdots \succ t_n$, where t_i are the values of the same argument appearing in the predicate's consecutive recursive calls.*

The argument in the above definition is called the *focused position* of the predicate. Given the definition of `append/3` in section 8.2.3, the predicate `append` is focused on the first argument when the first argument is bound to a list of items (which can be variables) of finite length. Each recursive call reduces the number of items in the list by one. On the other hand, if the first argument is bounded to a variable, the call may go on forever. The same can be said about `member/2` or `sat` in Example 8.3.3, where one sequence of calls of `sat` in the query `test2(X, Y)` may contain the following values as the argument of `sat`:

```
and(X, not(Y)), X, or(X1, _), X1, or(X2, _), X2, or(X3, _), X3, ...
```

Example 8.3.5. In Example 8.3.2, the transitivity of the `ancestor` relation is expressed by the rule `ancestor(X, Y) :- ancestor(X, Z), ancestor(Z, Y).` This rule cannot be used in Prolog as it causes an infinite loop. To define `ancestor`, we may use the `parent/2` relation, which is defined by a set of ground facts, and can be represented by a directed acyclic graph (DAG) among individuals. Then we can show that `ancestor/2` defined by the following rules is a focused recursion on the first argument.

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Let $X_0, X_1, ..., X_n$ be the values of the first argument in the recursive calls of `ancestor`. Under the assumption that `parent` defines a DAG among the individuals, we can see that the distance from X_i to Y in the DAG is reduced by one when i increases by one. Thus, the length of longest paths of the DAG provides an upper bound for the number of recursive calls of `ancestor`. It is easy to see that `ancestor` is the transitive closure of `parent`. □

Proposition 8.3.6. *If a recursive predicate P is focused on an input t_0, then then P has last termination on t_0.*

8.3.3 Tail Recursions

In Prolog, due to the power of logic variables, many predicates can be naturally written in a tail recursive way, where the recursion happens in the last position of the body. For example, the predicates `list, member, append, select`, and `delete`, are tail recursive as given in section 8.2.3. On the other hand, `reverse, permutation` and *length* are not tail recursive.

In many cases, tail recursion is good for performance. For conventional programs, a smart compiler can change tail recursion to an interactive one, thus saving the time and space of using a stack to store the environments of a recursive call. In Prolog, the same technique can be used so that a tail call means that the Prolog system can automatically reuse the allocated space of the environment on the local stack. In typical cases, this measurably reduces memory consumption of your programs, from $O(N)$ in the number of recursive calls to $O(1)$. Since decreased memory consumption also reduces the stress on memory allocation and garbage collection, writing tail recursive predicates often improves both space and time efficiency of your Prolog programs.

Example 8.3.7. The `reverse/2` predicate defined in section 8.2.3 is copied here:

```
reverse([], []).
reverse([F|X], Z) :- reverse(X, Y), append(Y, [F], Z).
```

Suppose the query is `?- reverse([a, b, c, d], L)`. The subgoal list can become `?- reverse([], []), append([], [d], L3), append(L3, [c], L2), append(L2, [b], L1), append(L1, [a], L)`, before it is shrinking. To avoid this long list of subgoals, we may introduce a tail recursive `rev/3`:

```
reverse(X, Y) :- rev(X, [], Y).
rev([], R, R).
rev([U|X], Y, R) :- rev(X, [U|Y], R).
```

Now, the subgoal list for `?- reverse([a, b, c, d], L).` never contains more than one item at any time:

```
?- reverse([a, b, c, d], L)
?- rev([a, b, c, d], [], L)
?- rev([b, c, d], [a], L)
?- rev([c, d], [b, a], L)
?- rev([d], [c, b, a], L)
?- rev([], [d, c, b, a], L)
```

□

This example illustrates that tail recursion can save space and time. However, you should not overemphasize it, as for beginners, it is more important to understand termination, and to focus on clear declarative descriptions.

8.3.4 A Prolog program for N-Queen Puzzle

Example 8.3.8. Below is a simple Prolog program for generating all legal positions for the N-queens problem. The predicate `queen`(N, L) takes N as the number of queens and generate a list L for the row numbers of N queens in the N columns.

□

```
% queen(N, R) generates at first a list R of N variables.
% Assuming one variable per column, then generate values for R,
% which gives the row number of the queen in each column.
queen( N, Res) :-
    length( Res, N),            % Res = N variables for columns
    gen_list( 1, N, L),         % L = [1, 2, ..., N]
    solution( N, Res, L).       % Res is a permutation of L

% gen_list( X, N, R) is true iff R = [X, X+1, X+2, ..., N].
gen_list( X, N, []) :- X>N, !.
gen_list( X, N, [X | Res]) :- Y is X+1, gen_list( Y, N, Res).

% solution( N, RowValues, List) if
%     RowValues represent a list of N non-attacking queens
```

```
solution( 0, [], _).                % run out of columns
solution( C, [R|Others], L)  :-     % 1st queen at position [R,C]
  C1 is C-1,
  select( R, L, L2),                % pick row R for column C
  solution( C1, Others, L2),        % find a partial solution
  noattack( R/C, C1, Others).       % 1st queen doesn't attack Others

noattack( _, _, []).                % Nothing to attack
noattack( R/C, C1, [R1 | Others])  :-
  C1-C =\= R1-R,                    % Different diagonals
  C1-C =\= R-R1,
  C2 is C1-1,
  noattack( R/C, C2, Others).       % no attacks for Others
```

Given the query is `?- queen(4, R).` the first answer is `L = [2,4,1,3]`, which says that the queens are placed at row 2 in the first column, row 4 in the second column, etc., and the second answer is `L = [3,1,4,2]`. In the above program, each position is generated and checked against the positions generated previously, so that no two positions can be attacked each other by the move of queens. Using Prolog's depth-first search, the program is simple to write but highly inefficient. When $N > 12$, it will take quite a while to find one solution, must slower than a SAT solver. Sudoku puzzles can also be solved by Prolog using the same logic. However, Sudoku cannot be solved efficiently using this generate-and-test approach, as the search space is huge. SAT solvers are a much better tool for the Sudoku puzzles.

8.4 Beyond Clauses and Logic

As a general-purpose language, Prolog provides various built-in predicates to perform routine activities like input/output, using graphics and otherwise communicating with the operating system. These predicates are not given a relational meaning and are only useful for the side-effects they exhibit on the system. For example, the predicate `write/1` displays a term on the screen, and returns true when it succeed. Besides this kind of extensions, various kinds of extensions are proposed, some intended to improve the expressive power, some speed up the execution. Some of these extensions follow the first-order logic; some go beyond the first-order logic.

For instance, since the occur-check in the unification algorithm is expensive, most prolog implementations choose to ignore the occur-check. This does not cause any problem in most applications. If you run into this problem and need a sound unification algorithm, you may call the built-in function `unify_with_occurs_check`.

```
?- X = f(Y,Y).
X = f(Y,Y)
yes

?- X = f(X,Y).
cannot display cyclic term for X

?- unify_with_occurs_check(X, f(X,Y)).
no
```

8.4.1 The Cut Operator !

The cut, in Prolog, is a predicate of arity 0, written as `!`, which always succeeds, but cannot be backtracked. It is best used to prevent unwanted backtracking, including the finding of extra solutions by Prolog and to avoid unnecessary computations.

Example 8.4.1. The figure shows how the cut operator affects the search procedure *engine* and *engine*1 by a contrived example, where `p` is reduced to a list of subgoals by rule 0, which contains !. When ! is reached by *engine*1, alternative rules, such as rule 1 for `p`, are eliminated; alternative rules for all the left siblings of !, such as rule 4 for `a`, are also eliminated. The right siblings of ! are not affected. □

Some programmers call the cut a controversial control facility because it was added for efficiency reasons only and is a function of logic.

A *green cut* is a use of cut which only improves efficiency. Green cuts are used to make programs more efficient without changing program output. For example, in the definition of `split` predicate of `qsort`, we may insert a cut after `H =< P` of the second rule, that is, the three clauses become the following.

```
split(_, [], [], []).
split(P, [H|I], [H|S], B) :- H =< P, !, split(P, I, S, B).
split(P, [H|I], S, [H|B]) :- H > P, split(P, I, S, B).
```

If the current goal G is `split(3, [2, 4, 1], L, R)`, then G unifies with the heads of the last two rules. Once the subgoal `H =< P`, i.e., `2 =< 3`, succeeds, the last clause will never be tried due to the cut operator, even if G unifies with the head of this clause and we want to see more answers. This cut is a green cut, because we know the subgoal `H>P`, i.e., `2 > 3`, will fail and the last clause cannot

be applied. The cut operator saves us the time for trying unifying G with the head of the last clause. With or without the cut, the program performs the same way as desired.

A *red cut* is a use of cut which changes the meaning of the program if the cut is missing. For the above `split` example, if `H =< P` fails, then `H > P` will succeeds. To save the cost of `H > P`, we can write the `split` program as the following:

```
split(_, [], [], []).
split(P, [H|I], [H|S], B) :- H =< P, !, split(P, I, S, B).
split(P, [H|I], S, [H|B]) :- split(P, I, S, B).
```

This program will give us the same result as the previous one for any input. However, this program will perform differently from the following program, where the cut is gone:

```
split(_, [], [], []).
split(P, [H|I], [H|S], B) :- H =< P, split(P, I, S, B).
split(P, [H|I], S, [H|B]) :- split(P, I, S, B).
```

So, the above cut is a red cut. Red cuts are potential pitfalls for bugs. For example, if the order of the two rules is reversed, the green cut will work correctly and the red cut will produce wrong results. If the saving is significant when using a red cut, a detailed documentation should be provided along the code. The cut should be used sparingly. Proper placement of the cut operator and the order of the rules is required to determine their logical meaning.

8.4.2 Negation as Failure

A Non-Horn clause like $A \vee B \vee \neg C$ can be represented as $\neg B \wedge C \rightarrow A$. Can we write it in Prolog as $A :- \neg B, C$? The answer lies on how to solve the subgoal $\neg B$ by the goal-reduction process. One approach is called "negation as failure": Try to solve B. If B succeeds, then $\neg B$ fails; if B fails, then $\neg B$ succeeds.

Using cut together with the built-in predicate `fail`, we may define this kind of negation, so that properties that cannot be specified by Horn clauses can be specified.

For example, to express "Mary likes any animals except reptiles" in Prolog,

```
likes(mary, X) :- reptile(X), !, fail.
likes(mary, X) :- animal(X).
```

We can use the same idea of "cut fail" to define the predicate `not`, which takes a term as an argument, and uses the built-in predicate "call". `not(G)` will "call" the term `G`, evaluates `G` as though `G` as a goal. If `G` succeeds, so is `call(G)`, and `not(G)` will fail. Otherwise, `not(G)` succeeds.

```
not(G) :- call(G), !, fail.
not(_).
```

Most Prolog systems have a built-in predicate like `not`. For instance, SWI-Prolog calls it `\+`. Remember, "not" is not the same as ¬, because it is based on the success/failure of goals. It can, however, be useful:

```
likes(mary, X) :- not(reptile(X)).
different(X, Y) :- not(X = Y).
```

Negation as failure can be misleading. Suppose the database held the names of members of the public, marked by whether they are innocent or guilty of some offense, and expressed in Prolog:

```
innocent(peter_pan).
innocent(winnie_the_pooh).
innocent(X) :- occupation(X, nun).
guilty(joe_bloggs).
guilty(X) :- occupation(X, thief).

?- innocent(einstein).
no.
```

If we add one more rule into the above program:

```
guilty(X) :- not(innocent(X)).

?- guilty(einstein).
yes.
```

It is one thing to show that "Einstein cannot be demonstrated to be innocent"; but it is quite another thing to incorrectly show that he is guilty. To justify the behavior of "negation as failure", people suggest the *closed world assumption*: Every ground atomic formula is assumed to be false unless it can be shown to be true by the program. Under this assumption, since we cannot show that `innocent(einstein)`

is true in the program, so `innocent(einstein)` is assumed to be false, thus it justifies the answer `guilty(einstein)`.

Some disturbing behavior is even more subtle than the innocent/guilty problem, and can lead to some extremely obscure programming errors. Here is a Prolog program about restaurants:

```
good_standard(goedels).
good_standard(hilberts).
expensive(goedels).
reasonable(R) :- not(expensive(R)).

?- good_standard(X), reasonable(X).
X = hilberts
yes

?- reasonable(X), good_standard(X).
no.
```

Logically, the query `good_standard(X), reasonable(X)` is equivalent to the query `reasonable(X), good_standard(X)`. Why do we get different answers for what seem to be logically equivalent queries? The difference between the questions is as follows. In the first query, the variable `X` is always instantiated when `reasonable(X)` is executed. In the second question, `X` is not instantiated when `reasonable(X)` is executed. The semantics of `reasonable(X)` differ depending on whether its argument is instantiated. To avoid this kind of problems, some implementations of Prolog ask that G is ground in $not(G)$.

Because of the problems associated with "negation as failure", extra care is needed when using it.

8.4.3 Beyond Clauses

Typically, a rule's body consists of atomic formulas, separated by the commas, and denotes the conjunction of subgoals. If the commas are replaced by the semicolons (;/2), then it will denote the disjunction of subgoals. A rule like `H :- (C; D), E` is no longer a clause: It is equivalent to two clauses: `H :- C, E` and `H :- D, E`, because ";" is considered to be $\vee$. Conjunctions "," and disjunctions ";" can only appear in the body, not in the head of a rule.

Prolog also supports the if-then relation by using the infix symbol "-¿/2". A rule like `H :- C -> G` has the same effect as `H :- C, G`: if `C` returns true, `G` will be

called; if `C` returns false, `C -> G` returns false. When `->` is combined with ";", we then have the if-then-else relation implemented as `H :- C -> G1; G2`: If `C` returns true, `G1` will be called; otherwise, `G2` will be called. Logically, `H :- C -> G1; G2`, which is not a clause, is equivalent to two clauses: `H :- C, G1` and `H :-` $\neg$ `C, G2` (which is not a Horn clause).

Once a predicate is declared as *dynamic*, Prolog allows the clauses of that predicate can be inserted or removed during the execution. For Example 8.1.1, if we have a file which contains the pair of father-children names, we can declare `papa` as dynamic: `dynamic(papa).` During the execution, we read names into `Father` and `Child`, and call `assertz(papa(Father, Child))`, which adds the clause `papa(Father, Child)` at the end of all existing clauses of the predicate `papa`. If we want to add the new clause at the beginning of the existing clauses, the command is `asserta/1`. To remove a clause during the execution, the command is `extract/1`. This set of commands allows the user to dynamically change the program that occurs as the result of executing `assertz/1` or `retract/1`. The change does not affect any activation that is currently being executed. Thus the database is frozen during the execution of a goal, and the list of clauses defining a predicate is fixed at the moment of its execution.

8.5 Exercise Problems

1. Given the following Prolog program,

   ```
   append([], Y, Y).
   append([H | X], Y, [H | Z]) :- append(X, Y, Z).
   ```

   Find the four solutions for the query `?- append(X, Y, [a, b, c]).` For each of the four solutions, find a resolution proof.

2. For the program in Example 8.3.3, place the clause for handling `and` before those for `or`, and run the query `?- test2(X, Y)` for three answers. Please produce the resolution proof for each of the three answers.

3. Implement in Prolog the selection sort, `ssort`(X, Y), where X is a list of integers and Y is the result of sorting X.

4. Implement in Prolog the merge sort, `msort`(X, Y), where X is a list of integers and Y is the result of sorting X.

5. The predicate `slowsort` is defined in Example 8.2.2. Construct the resolution proof for the query `?- slowsort([3,2,1], Y).`

6. Assume `xor/2` is the logical operation $\oplus$ (exclusive-or). Add the rules for converting the formula containing `xor` into NNF in the Prolog program in Example 8.1.2.

7. Add a predicate `cnf/2` into the Prolog program in Example 8.1.2, where `cnf(`F, X`)` takes a propositional formula F as input and converts F into an equivalent formula X in CNF.

8. Write a Prolog program which converts a first-order formula A into NNF (negation normal form), with the following assumptions: The predicate symbols are `p/2`, `q/1`, and `r/0`. The quantifiers are `all(`X, A`)` and `ex(`X, A`)`, where X takes a value in {`x, y, z`}.

9. Decide with evidence if the following Prolog programs are terminating or not. Do they have first termination? last termination? ground termination? on what inputs? under what conditions?

   (a)
   ```
   goHome(no_12).
   goHome(X) :- get_next_house(X,Y), goHome(Y).
   ```

   (b)
   ```
   hasFlu(rebecca).
   hasFlu(john).
   hasFlu(X) :- kisses(X, Y), hasFlu(Y).
   kisses(janet,john).
   ```

   (c)
   ```
   search(end).
   search(X) :- path(X,Y), search(Y).
   ```

10. Introduce a tail recursive predicate for each of the following predicate: (a) `length/2` and (b) `permutation/2`, so that the same result can be obtained through the tail recursive calls.

11. Given the following Prolog program which gives reviews on restaurants:

    ```
    not(P) :- call(P), !, fail.
    not(_).
    good_standard(goedels).
    good_standard(hilberts).
    expensive(goedels).
    reasonable(R) :- not(expensive(R)).
    ```

    Do the following two queries provide the same result? Why?

```
?- good_standard(X), reasonable(X).
?- reasonable(X), good_standard(X).
```

CHAPTER 9
UNDECIDABLE PROBLEMS IN FIRST-ORDER LOGIC

Undecidability concerns about the problems which are beyond the power of computers. To talk about "undecidable problems", we have to start with Turing machines. Turing machines are important in computability theory due to the *Church-Turing thesis*, which states that every computing model, invented in the past or to be invented in the future, can be simulated by a Turing machine.

Let us quote an account of historic events from the Wikipedia on the "Church-Turing thesis": In the 1930s, several independent attempts were made to formalize the notion of computability:

- In 1933, Kurt Gödel, with Jacques Herbrand, created a formal definition of a class called *general recursive functions*. The class of general recursive functions is the smallest class of functions (possibly with more than one argument) which includes all constant functions, projections, the successor function, and which is closed under function composition, recursion, and minimization.

- In 1936, Alonzo Church created a method for defining functions called λ-calculus. Within λ-calculus, he defined an encoding of the natural numbers called *Church numerals*. A function on the natural numbers is called λ-computable if the corresponding function on the Church numerals can be represented by a term of the λ-calculus.

- Also in 1936, before learning of Church's work, Alan Turing created a theoretical model for machines, now called *Turing machines*, that could carry out calculations from inputs by manipulating symbols on a tape. Given a suitable encoding of the natural numbers as sequences of symbols, a function on the natural numbers is called *Turing computable* if some Turing machine computes the corresponding function on the encoded natural numbers.

Church and Turing proved that these three formally defined classes of computable functions coincide: a function is λ-computable iff it is Turing computable, and iff it is general recursive. This has led mathematicians and computer scientists to believe that the concept of computability is accurately characterized by these three equivalent computing models. Other formal attempts to characterize computability have subsequently strengthened this belief. Since we cannot exhaust

all possible computing models, the Church–Turing thesis, although it has near-universal acceptance, cannot be formally proven.

In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible pairs $\langle M, w \rangle$, where M is a Turing machine and w is an input to M, cannot exist. Turing's proof is one of the first cases of undecidable decision problems to be concluded. The theoretical conclusion that it is not undecidable is significant to practical computing efforts, defining a class of problems which no computing devices can possibly perform perfectly.

The halting problem is not just a theoretic problem. This is a practical problem for many researchers. For example, the unsolved Collatz conjecture is related to whether the program below terminates for all positive integer x:

while $x > 1$ **do**
    if $even(x)$ $x := x/2$ **else** $x := 3 * x + 1$

The same code can be expressed as a recursive function

proc $f(x)$
    if $x = 1$ **return** 1
    else if $even(x)$ **return** $f(x/2)$
    else return $f(3 * x + 1)$

or as a rewrite system:

$$\begin{aligned} f(1) &\rightarrow 1 \\ f(2x) &\rightarrow f(x) \\ f(2x+1) &\rightarrow f(3x+2) \end{aligned}$$

Does $f(x) = 1$ for any positive integer x? Or will the above rewrite system terminate? The renowned mathematician Paul Erdos (1913-1996) once said about the Collatz conjecture: "Mathematics is not yet ready for such problems." He offered $500 for its solution.

9.1 Turing Machines

A *Turing machine* (TM) is a mathematical model of computation first proposed by Alan Turing in 1936. A Turing machine manipulates symbols on a tape of symbols according to a set of rules. According to the Church-Turing Thesis, any function on the natural numbers can be calculated by an effective method iff

it is computable by a TM. In other words, despite the model's simplicity, Turing machines are capable of simulating any algorithm on the natural numbers.

A Turing machine operates on a tape containing infinite cells and each cell contains one symbol. The machine positions its "head" over a cell and "reads" the symbol there. Then, as per the symbol and the machine's own present state in a user-specified set of instructions, the machine (i) writes a symbol in the cell, (ii) either moves the tape head one cell left or right, (iii) possibly changes the current state, and (iv) goes to the next round, or halts the computation with either "success" or "failure".

Initially, the beginning of the tape is filled with an input string of symbols and is blank everywhere else. The machine positions its head to the first symbol with the designated state called the "initial state". The machine moves as described above through the steps $(i) - (iv)$. There are three possible outcomes: (1) the machine stops with "success"; (2) it stops with "failure"; or (3) it loops forever.

Turing was able to answer a fundamental question in the negative: Does an algorithm exist that can determine whether any arbitrary TM stops on a given input string? This is so-called the "halting problem" of Turing machines.

A Turing machine can do everything that a real computer can do. However, their minimalist design makes them unsuitable for computation in practice: real-world computers are based on different designs that, unlike Turing machines, use random-access memory.

While a Turing machine can express arbitrary computations, nonetheless, even a Turing machine cannot solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation. Turing machines proved the existence of fundamental limitations on the power of mechanical computation.

9.1.1 Formal Definition of Turing Machines

The TM model uses an infinite tape as its unlimited memory. The symbols appearing in the tape is denoted by Γ, called the *tape alphabet*. By default, Γ is a super set of Σ, which are the symbols used in all the input strings. By default, the blank symbol, $\sqcup$, is in $\Gamma - \Sigma$. Σ^* denotes the set of all input strings (see Example 1.3.12) and Γ^* denotes the set of all tape contents excluding the trailing blanks.

For any string $w \in \Sigma^*$, let $w^0 = \epsilon$ if $i = 0$ and $w^{i+1} = ww^i$, where $i \geq 0$. For example, $a^3 = aaa$ and $(ab)^2 = abab$.

Example 9.1.1. Suppose $\Sigma = \{a, b\}$ and we would like to design a TM which checks whether input string $w \in L_1 = \{a^j b^k \mid 0 \leq j \leq k\}$. To design a Turing

machine is the same as to design an algorithm working on the array of symbols. If the machine reads 'a', we may cross it off, which corresponds to $j := j - 1$, and then look for the symbol 'b' and cross it off, which corresponds to $k := k - 1$. The pseudo code of the algorithm may read as:

while $(j > 0 \wedge k > 0)$ **do**
    $j := j - 1; k := k - 1;$
if $(j = 0)$ **return** "success" **else return** "failure";

□

The machine has a tape head that can read and write symbols and move around on the tape. Initially the tape contains only the input string in the beginning portion of the tape and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs "success" (or "accept") and "failure" (or "reject") are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.

To report "success", the Turing machine uses the state q_a (or q_{accept}); to report "failure", the machine either uses the state q_r (or q_{reject}) or provides no move information in that situation ("reject" by blocking the next move). Since each move is decided by the current symbol and the current state, we may define each move as a function δ which takes a state and a symbol as input and outputs (i) the next state, (ii) the symbol to be written on the tape, and (iii) the move direction of the tape head.

Definition 9.1.2. *A Turing machine is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$, where Q is a finite set of states; Σ is the alphabet of input strings, Γ is the set of tape symbols, $\Sigma \subset \Gamma$, $\sqcup \in \Gamma - \Sigma$, the initial state $q_0 \in Q$, and the final state (or the accepting state) $q_a \in Q$ and the (partial) function δ,*

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$$

defines each move of M, where L means the tape head moves left and R means the tape head moves right. If the tape head points to the first symbol on the tape, it cannot move left ("move off the tape" is blocked).

Example 9.1.3. Continuing from Example 9.1.1, We would like to construct $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$ to accept the stings in $L_1 = \{a^j b^k \mid 0 \leq j \leq k\}$, where $\Sigma = \{a, b\}$, $\Gamma = \{a, b, x, y, \sqcup\}$, $Q = \{q_0, q_1, q_2, q_3, q_a\}$ and the functions of each state are:

- q_0: at the beginning of the while loop for checking $j > 0$ or $k > 0$.
- q_1: done $j := j - 1$, look for b.
- q_2: done $k := k - 1$, rewind to the last x.
- q_3: found $j = 0$, check $j \geq 0$ and no a's after b.

and δ is defined as follows. □

$$
\begin{array}{rrcll}
1 & \delta(q_0, \sqcup) & = & (q_a, \sqcup, R) & // \; j = k = 0, \epsilon \in L_1 \\
2 & \delta(q_0, a) & = & (q_1, x, R) & // \; j > 0, j := j - 1, \text{goto } q_1 \\
3 & \delta(q_0, b) & = & (q_3, b, R) & // \; j = 0, k > 0, \text{goto } q_3 \\
4 & \delta(q_0, y) & = & (q_3, y, R) & // \text{ skip } y, \text{ goto } q_3 \\
5 & \delta(q_1, a) & = & (q_1, a, R) & // \text{ skip } a, \text{ continue right} \\
6 & \delta(q_1, y) & = & (q_1, y, R) & // \text{ skip } y, \text{ continue right} \\
7 & \delta(q_1, b) & = & (q_2, y, L) & // \; k := k - 1, \text{goto } q_2 \\
8 & \delta(q_2, y) & = & (q_2, y, L) & // \text{ skip } y, \text{ continue left} \\
9 & \delta(q_2, a) & = & (q_2, a, L) & // \text{ skip } a, \text{ continue left} \\
10 & \delta(q_2, x) & = & (q_0, x, R) & // \text{ found } x, \text{ goto } q_0 \\
11 & \delta(q_3, y) & = & (q_3, y, R) & // \text{ skip } y, \text{ continue right} \\
12 & \delta(q_3, b) & = & (q_3, b, R) & // \text{ skip } b, \text{ continue right} \\
13 & \delta(q_3, \sqcup) & = & (q_a, \sqcup, L) & // \; k \geq 0, b^k \in L_1
\end{array}
$$

Definition 9.1.4. *Given an input string w to a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$, let $q \in Q$ be the current state, α and β be the strings of tape symbols before and after the tape head, respectively. The triple $\langle \alpha, q, \beta \rangle$, or simply $\alpha q \beta$, is called a* configuration *of M. The initial configuration is $q_0 w$. A legal move of M is a pair (C_1, C_2) of configurations, written as $C_1 \vdash C_2$, such that either $C_1 = \alpha q a \beta$, $\delta(q, a) = (p, b, R)$, and $C_2 = \alpha b p \beta$, or $C_1 = \alpha c q a \beta$, $\delta(q, a) = (p, b, L)$, and $C_2 = \alpha p c b \beta$.*

Example 9.1.5. Continuing from the previous example, $q_0aabb \vdash xq_1abb \vdash xaq_1bb \vdash xq_2ayb \vdash q_2xayb \vdash xq_0ayb \vdash xxq_1yb \vdash xxyq_1b \vdash xxq_2yy \vdash xq_2xyy \vdash xxq_0yy \vdash xxyq_3y \vdash xxyyq_3\sqcup \vdash xxyq_ay$. So $aabb$ is accepted by M_1.

On the other hand, $q_0aba \vdash xq_1ba \vdash q_2xya \vdash xq_0ya \vdash xyq_3a$. Since $\delta(q_3, a)$ is not defined, the move is blocked and the input aba is rejected by M_1. □

Definition 9.1.6. *Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$ be a TM, and $w \in \Sigma^*$, w is said to be* accepted *by M if*

$$q_0 w \vdash^* \alpha q_a \beta$$

where $\vdash^*$ denotes the reflexive and transitive closure of $\vdash$, and $\alpha, \beta \in \Gamma^*$.

The language recognized by M is the set $L(M)$, where

$$L(M) = \{w \mid w \in \Sigma^*, w \text{ is accepted by} M\}.$$

It is easy to see that $a^2b^2 \in L(M_1)$ and $aba \notin L(M_1)$, as shown by Example 9.1.5. It is not an easy task to show that M_1 recognizes $L_1 = \{a^jb^k \mid 0 \leq j \leq k\}$, i.e., $L(M_1) = L_1$. We would like to point out that M_1 can be also regarded performing $k - j$, as the number of b's left on the tape when M_1 accepts is exactly $k - j$.

9.1.2 High-level Description of Turing Machines

In general, a detailed description of a TM is often tedious. We often prefer to provide a high level description such that every line in the description can be implemented by a finite number of moves of the machine.

Example 9.1.7. If we are asked to design a TM M_2 to recognize $L_2 = \{b^ia^jb^k \mid i * j = k, i, j, k \geq 0\}$, the basic algorithm is the following.

Check if $i = 0$ or $j = 0$, then k must be 0;
while $(i > 0)$ **do**
    $i := i - 1; k := k - j$;
if $(k = 0)$ **return** "success" **else return** "failure";

From Example 9.1.1, we know how to check $i = 0$, how to do $i := i-1$ and $k := k-j$. There is no technical difficulty to design M_2 for L_2.

M_2 = "On input $w \in \{a, b\}^*$

1. Scan the tape to check if $i = 0$ or $j = 0$. If yes, *accept* if $k = 0$, else *reject*;
2. Check if w is of form $b^ia^jb^k$; if not, *reject*, else go to the first symbol;
3. Decrease i by one by crossing off the first b;
4. Zigzap between a^j and b^k to check if $k < j$, if yes, "reject", else $k := k - j$;
5. Go back to the first b and restore a;
6. If $i > 0$, goto 3; otherwise check if $k = 0$; if yes, *accept*, else *reject*."

□

Once we know how to check $i * j = k$, we can modify the machine code to compute $i * j$, k/i (an exercise), or j^k. We can also compare two numbers, or two strings are identical, or copy one string from one place to another. This kind of

operations can be assumed in a high-level description of a TM. In many cases, a high-level description is the only way to design a TM. However, a basic algorithm or procedure, which can be expressed by a high-level description, is a must in the design of some Turing machines.

9.1.3 Recognizable vs Decidable

Given any TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$, if we feed $w \in \Sigma^*$ to M, there will be three possible outcomes: (1) M halts with "accept"; (2) M halts with "reject"; and (3) M loops on w. In other words, M partitions Σ^* into three disjoint sets:

- $L(M) = \{w \mid w \in \Sigma^*, M \text{ halts and accepts } w\}$,
- $reject(M) = \{w \mid w \in \Sigma^*, M \text{ halts and rejects } w\}$, and
- $loop(M) = \{w \mid w \in \Sigma^*, M \text{ loops on } w\}$.

Obviously, $L(M) \cup reject(M) \cup loop(M) = \Sigma^*$. By convention, $\overline{L(M)} = \Sigma^* - L(M) = reject(M) \cup loop(M)$.

In computability theory, any subset L of Σ^* is said to be a *formal language*. L can be $\emptyset$ or Σ^*. Turing machines can be used to classify formal languages.

Definition 9.1.8. (**recognizable and decidable**) *A formal language $L \subseteq \Sigma^*$ is said to be* recognizable*, if there exists a TM M such that $L = L(M)$, and we say M is a recognizer of L.*

L is said be decidable *if there exists a TM M such that $L = L(M)$ and $loop(M) = \emptyset$, and we say M is a* decider *of L.*

If L is decidable and $L(M) = L$ for some TM M, it does not mean that $loop(M) = \emptyset$. The definition just ensures that there exists a decider M' such that $L = L(M')$ and $loop(M') = \emptyset$.

9.2 Decidability of Problems

In computability theory, a problem is typically a decision problem which has a simple yes/no answer. Optimization problems like "finding the longest simple cycle in graph G" is always converted to a question like "does G have a simple cycle of length at least m?" by introducing a number m, so that it becomes a decision problem.

9.2.1 Encoding of Decision Problems

Another common practice in computability theory is to encode every decision problem by a set of strings.

For instance, the problem that "does G have a simple cycle of length at least m?" can be encoded as a set L_1:

$$L_1 = \{\langle G, m\rangle \mid G \text{ has a simple cycle of length at least } m\},$$

where $\langle G, m\rangle$ is a string which represents G and m; we may think $\langle G, m\rangle = \langle G\rangle\langle m\rangle$. If G contains n vertices and $m \leq n$, then the alphabet for $\langle G, m\rangle$ can be $\{1, 2, ..., n, (,), \cdot\}$, where each vertex is represented by a number i, $1 \leq i \leq n$, and each edge is represented by $(i \cdot j)$, $1 \leq i, j \leq n$. The alphabet can also be $\{0, 1\}$, if every symbol in $\{1, 2, ..., n, (,), \cdot\}$ is encoded by a binary string of equal length, as we use the ASCII code in a computer. Now, a graph G has a simple cycle of length at least m iff $\langle G, m\rangle \in L_1$. L_1 catches the essence of this decision problem.

"Does there exist an algorithm A which finds a simple cycle in G of length at least m?" This is also a decision problem and can be encoded as

$$L_2 = \{\langle A, G, m\rangle \mid \text{Algorithm } A \text{ finds in } G \text{ a simple cycle of length at least } m\}$$

where $\langle A\rangle$ can be the code file for implementing A in a programming language.

Turing machines can also be encoded as a string. For instance, the definition of M_1 in Example 9.1.1 takes less than 20 lines of this book and each line has less than 100 characters, so a string of 2000 characters will be more than enough to encode M_1.

There are many decision problems about Turing machines and the following are some of them.

- $A_{TM} = \{\langle M, w\rangle \mid \text{TM } M \text{ accepts input } w\}$.
- $H_{TM} = \{\langle M, w\rangle \mid \text{TM } M \text{ halts on input } w\}$.
- $E_{TM} = \{\langle M\rangle \mid L(M) \text{ is empty}\}$.
- $One_{TM} = \{\langle M\rangle \mid |L(M)| = 1\}$.
- $Fin_{TM} = \{\langle M\rangle \mid L(M) \text{ is finite}\}$.
- $S10_{TM} = \{\langle M\rangle \mid M \text{ has at least 10 states}\}$.
- $A10_{TM} = \{\langle M\rangle \mid M \text{ accepts at least 10 strings}\}$.

- $B10_{TM} = \{\langle M\rangle \mid M \text{ moves at least 10 steps on any input}\}$.

The language A_{TM} encodes the *acceptance problem* of Turing machines: Given any TM M and any input w, does M accept w? This is one of the first problems shown to be undecidable, that is, there exists no algorithm to solve the acceptance problem.

The language H_{TM} encodes the well-known halting problem of Turing machines. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible $\langle M, w\rangle$ pairs cannot exist. Later, after giving formally the definition of "undeciable", we will show that both A_{TM} and H_{TM} are Turing recognizable, but not decidable.

The language E_{TM} encodes the *emptyness problem* of Turing machines. While both A_{TM} and H_{TM} are undecidable but Turing recognizable, E_{TM} is even not Turing recognizable.

The rest languages in the above list encode some problems less popular. You may create your own languages for any decision problem.

Let the encodings of all TMs use a common alphabet, say Σ, and $L_{TM} = \{\langle M\rangle \mid M \text{ is a TM}\}$.

Proposition 9.2.1. *L_{TM} is countable.*

Proof. Since $\langle M\rangle \in \Sigma^*$ for every TM M, $L_{TM} \subseteq \Sigma^*$. According to Lemma 1.3.4, Σ^* is countable and any subset of a countable set is also countable. □

The above proposition claims that the set of possible TMs is countable.

9.2.2 Decidable Problems

Since every decision problem can be encoded as a formal language $L \subseteq \Sigma^*$, we wish to design a TM M which can answer the question of $w \in L$ or not, for any $w \in \Sigma^*$. This is only possible when L is decidable; if L is not recognizable, then no such TMs can exist.

Definition 9.2.2. (**decidable and computable problem**) *A decision problem is said to be* decidable *if the language of its encoding is decidable. A decision problem is said to be* computable *if its encoding is recognizable.*

All the decision problems we have met in the courses on data structures and algorithms, where their complexity is obtained by a big O function, belong to the class of decidable problems. In computability theory, a decider for a decision problem is also call a *decision procedure*, or an *algorithm*, which coincides with the

conventional definition of *algorithm*: a set of instructions which will be executed in a finite number of steps for any input.

Some problems are computable but undecidable; some are not computable. Any Boolean function $f : N \to \{0, 1\}$, over the natural numbers is a decision problem about the natural numbers: Does $f(n) = 1$ for any $n \in N$? According to Lemma 1.3.5, the set of all Boolean functions over the natural numbers is not countable. That is, there are more Boolean functions than the available TMs, which are countable. For any function $f : N \to \{0, 1\}$, it is not always possible to have a TM M_f such that M accepts i iff $f(i) = 1$ for any natural number i. There exist many, many functions which cannot be computed by any computing device in view of the Church-Turing thesis.

Here, we are in particular interested in decision problems about TMs.

Example 9.2.3. Let $S10_{TM} = \{\langle M\rangle \mid M \text{ has at least 10 states}\}$. Then $S10_{TM}$ is decidable because we can design a TM which counts how many states in $\langle M\rangle$. It takes no more than the linear time to count and if the count goes beyond 10, stop with "accept", otherwise, stop with "reject". □

Example 9.2.4. Let $B10_{TM} = \{\langle M\rangle \mid M \text{ moves at least 10 steps on any input}\}$. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a)$ and M_B be the decider that we will construct for $B10_{TM}$.

The basic algorithm of M_B works as follows: The tape of M_B is divided into three sections separated by the special symbol # and \$. Section 1 stores the input string $\langle M\rangle$ and will never change.

Section 2 has 11 cells and takes turns to store all the input strings $w \in \Sigma^*$ for M, $|w| \leq 10$. The strings in Σ^* are generated by the following order: (i) shorter strings first; (ii) for the strings of the same length, smaller string under the lexicographic order first. Thus, if $\Sigma = \{0, 1\}$, then the strings are in this order: $\epsilon, 0, 1, 00, 01, 10, 11, 000, ...$ We say the *successor* of ϵ is 0, the *successor* of 0 is 1, and so on.

Section 3 will be the space for simulating the moves of M.

$M_B =$ "On input $\langle M\rangle$, M is a TM

1. Place # at the end of $\langle M\rangle$; move right 11 steps and place \$.
   Treat the string between # and \$ as w, which is empty initially.
2. Write q_0 of M after \$ in section 3.
3. Copy string w in section 2 to section 3, after q_0.
   Now section 3 contains q_0w, an initial configuration of M.
4. Zigzap between section 1 and section 3, do the following:
   4.1 Using the states of M_B to remember the pair (q, a) in section 3;
   4.2 Find the delta definition for (q, a) in section 1;

4.3 If no $\delta(q, a)$ is found, stop with "reject";
4.4 If found, use the states to remember $\delta(q, a) = (p, b, D)$, $D \in \{L, R\}$;
4.5 Move to section 3, and update the configuration of M according to $\delta(q, a) = (p, b, D)$;
4.6 If 4.5 has been done less than 10 times, go to 4.1;
4.7 else erase the content of section 3 and go to 5;
5. Go to section 2 and generate the successor of w, replacing w;
6. If w exceeds 10 symbols, stop with "accept", else go to 2. "

Lines 4.1-4.7 simulates the moves of M for 10 steps. If this is possible, M_B continues for the successor of w (4.7); if the move of M is blocked before moving 10 steps, it stops with "reject" (4.3), because we have found an input string for M to move less than 10 steps. In line 5, we generate the successor of the current w and replace w by it. If all the strings of length 10 or less are generated and M moves over 10 steps on each of them, we stop with "accept", because to read the 11^{th} symbol, M has to move at least 10 steps. Thus, we do not need to consider the strings of length more than 10 for w. □

Lines 4.1-4.5 describe how a move of M is simulated by M_B. Later, when we say "M' simulates M", we refer to this example for the details of the simulation. Note that the states of a TM can remember only a finite amount of information. For M_B, one definition of $\delta(q, a) = (p, b, D)$ contains only five symbols. You cannot use the states to restore the information like a^i, where i is any number.

9.2.3 Undecidable Problems

Theorem 9.2.5. *The acceptance problem of Turing machines is undecidable. That is,* $A_{TM} = \{\langle M, w\rangle \mid \text{TM } M \text{ accepts input } w\}$ *is undecidable.*

Proof. Assume that A_{TM} is decidable, then there exists a hypothetical decider H for A_{TM}. Since H is a decider, H either accepts x or rejects x for any input x, and H accepts $\langle M, w\rangle$ iff M accepts w.

Using H, we construct another decider D, which checks if a TM M does not accept its own code:

$D =$ "On input $\langle M\rangle$, M is a TM,
1. Simulate H on $\langle M, \langle M\rangle\rangle$ until H halts.
2. If H accepts, *reject*; if H rejects, *accept*."

Obviously, D is a TM and $L(D) = \{\langle M\rangle \mid M \text{ is a TM, } M \text{ rejects } \langle M\rangle\}$ is decidable, because D halts on any input.

Let us check if $\langle D\rangle \in L(D)$ and there are only two cases:

- $\langle D \rangle \in L(D)$: Every member $\langle M \rangle$ of $L(D)$, including $\langle D \rangle$, satisfies the definition of $L(D)$: "M rejects $\langle M \rangle$". Replace M by D, we have "D rejects $\langle D \rangle$", which means $\langle D \rangle \notin L(D)$.
- $\langle D \rangle \notin L(D)$: That means D rejects $\langle D \rangle$. By the definition of $L(D)$, $\langle D \rangle \in L(D)$ should be true.

In both cases, we have a contradiction, so D cannot exist. D is constructed from H, hence, H cannot exist. H comes from the assumption that A_{TM} is decidable. Thus A_{TM} is undecidable. □

The above proof is analogous to that of "Barber's paradox" (see Paradox 1.3.8). The TM D, which accepts the encodings of those TMs that do not accept their own encoding, corresponds to "the Barber who shaves all those who do not shave themselves". Both proofs originated from Cantor's diagonalization method. To see this connection, the reader is recommended to Michael Sipser's textbook on theory of computation.

9.2.3.1 Reduction

Once we have found one undecidable language, other undecidable languages can be found through reduction, as illustrated by the proof of the halting problem of Turing machines.

Theorem 9.2.6. *The halting problem of Turing machines is undecidable. That is,* $H_{TM} = \{\langle M, w \rangle \mid \text{TM } M \text{ halts on input } w\}$ *is undecidable.*

Proof. Assume H_{TM} is decidable, then there exists a hypothetical decider H for H_{TM}. Since H is a decider, H will halt on any input, and H accepts $\langle M, w \rangle$ iff M halts on w.

Using H, we can construct another decider S for A_{TM} as follows:

$S =$ "On input $\langle M, w \rangle$, M is a TM and w is an input to M,
1. Simulate H on $\langle M, w \rangle$ until H halts.
2. If H rejects, *reject*.
3. Simulate M on w until M halts.
4. If M accepts, *accept*; if M rejects, *reject*."

At line 2, when H rejects $\langle M, w \rangle$, it means M loops on w, so w cannot be accepted by M. This justifies the rejection of w by S. At line 3, we know M will halt w. When M halts on w, S returns the output of M at line 4. All four lines of

S will be finished in a finite number of steps, so S is a decider for A_{TM}. Since S cannot exist, H cannot exist, either. Thus, H_{TM} must be undecidable. □

The above two theorems are what we needed to show the undecidability of some logical problems.

Reduction is a powerful tool to show that a problem X is undecidable: Assume X is decidable and A is a decider for X. We use A to construct another decider B for a known undecidable problem Y. Since B cannot exist, so A cannot exist and X is thus undecidable. In the proof of the above theorem, X is H_{TM}, A is H, Y is A_{TM} and B is S. We showed that if there exists a decider for H_{TM}, then there exists a decider for A_{TM}.

We may switch the role of H_{TM} and A_{TM} by showing that if there exists a decider for A_{TM}, then there exists a decider for H_{TM}. Suppose A_{TM} is decidable and A is its decider. We may construct a decider B for H_{TM} as follows:

$B =$ "On input $\langle M, w\rangle$, M is a TM and w is an input to M,

1. Let $\langle M'\rangle$ be a copy of $\langle M\rangle$ and we modify $\langle M'\rangle$ as follows:
   For any state q and tape symbol x of M, if q is not q_a
   (the final state), and $\delta(q, x)$ is not defined, add in M':
   $\delta'(q, x) = (q_a, x, R)$.
2. Simulate A on $\langle M', w\rangle$ until A halts.
3. If A accepts, *accept*; if A rejects, *reject*."

It is easy to check when M rejects w, M' will accept w; when M accepts w or loops on w, so does M'. Thus, $loop(M) = loop(M')$, $reject(M') = \emptyset$, and $L(M') = L(M) \cup reject(M)$. B accepts $\langle M, w\rangle$ iff A accepts $\langle M', w\rangle$, iff M' accepts w, iff M accepts w or M rejects w. In other words, B accepts $\langle M, w\rangle$ iff M halts on w; B rejects $\langle M', w\rangle$ iff M loops on w. This justifies the output of B at line 3. Since all the three lines of B can be done in a finite number of steps, B is a decider for H_{TM}.

We have just shown that both A_{TM} and H_{TM} are undecidable. Are they recognizable? The answer is yes for both of them.

$U =$ "On input $\langle M, w\rangle$, M is a TM and w is an input to M,

1. Simulate M on w.
2. When M halts on w, *accept* if M accepts; *reject* if M rejects."

It is clear that U will accept $\langle M, w\rangle$ if M accepts w; U will loop if M loops on w. Thus $L(U) = A_{TM}$; U is called the *universal Turing machine*, because it simulates any Turing machine.

Similarly, H_{TM} can be recognized by a Turing machine.

T = "On input $\langle M, w\rangle$, M is a TM and w is an input to M,
1. Simulate M on w.
2. When M halts on w, *accept*."

T accepts iff M halts on w, thus $L(T) = H_{TM}$.

Proposition 9.2.7. *If a formal language L is recognizable but undecidable, then $\overline{L}$ is not recognizable.*

Proof. (sketch) Let M_1 be the recognizer of L. If $\overline{L}$ is also recognizable, let M_2 be its recognizer. We may use M_1 and M_2 to create a decider for L:

T = "On input $\langle M, w\rangle$, M is a TM and w is an input to M,
1. Simulate M_1 on w for one move;
2. If M_1 accepts, *accept*;
3. Simulate M_2 on w for one move;
4. If M_2 accepts, *reject*;
5. go to 1. and continue the simulations. "

For any $w \in \Sigma^*$, either $w \in L$ or $w \in \overline{L}$. So either M_1 accepts w or M_2 accepts w. T will halt when one of them accepts. If $w \in L$, T will accept; otherwise, T will reject. Thus, T is a decider for L. □

Applying the above proposition to A_{TM} and H_{TM}, we now know that $\overline{A_{TM}}$ and $\overline{H_{TM}}$ are not recognizable.

9.3 Turing Completeness

By the Church-Turing thesis, any computing model can be simulated by a Turing machine. On the other hand, it becomes easy to design a new computing model which has the same power as a Turing machine, as long as the new computing model is shown to do what a Turing machine can do. The concept of *Turing completeness* just expresses this property: If a computing model has the ability to simulate a Turing machine, it is *Turing complete*. Both Gödel's recursive functions and Church's λ-calculus are Turing complete.

Example 9.3.1. A two-stack machine is a variation of Turing machine in that the tape is replaced by two stacks: The first stack stores all the symbols from the tape head (inclusive) to right; the second stack stores all the symbols from the tape head (exclusive) to left. The state information remains the same. It is easy to show that the two-stack is Turing complete by simulating all the Turing moves by the

operations on stacks. Initially, the input string is stored in the first stack with the first symbol on the top of the stack. The simulation of a Turing move goes as follows: A symbol a is popped off from the first stack; if the stack is empty, use the blank symbol. If $\delta(q, a) = (p, b, R)$, then push b into the second stack; if $\delta(q, a) = (p, b, L)$, then push b into the first stack, and also pop the top symbol from the second stack and push it into the first stack. □

The significance of being Turing complete is that all the undecidable properties of Turing machines will be inherited by the new computing model. Thus, the halting problem of a two-stack machine is undecidable, because if we have an algorithm to tell the termination of a two-stack machine, this algorithm can be used to show the termination of a Turing machine. Similarly, if we can show the Turing completeness of a logic, we can derive the same result.

9.3.1 Turing Completeness of Prolog

Proposition 9.3.2. *Prolog is Turing complete.*

Proof. The proposition can be shown by using Prolog to simulate a Turing machine, using the idea of two-stack machines. The content of the Turing machine tape is split into two lists, L and R, which correspond to the two stacks in a two-stack machine. Initially, $L = []$ and R contains the input string. We transform L to L' and R to R' according to the Turing machine, until we reach the final state. The transformation is illustrated by a Prolog program for the Turing machine in Example 9.1.1

```
% M1 checks a^jb^k for j =< k.
delta(q0, b, qa, bl, right).
delta(q0, a, q1, x, right).
delta(q0, b, q3, b, right).
delta(q0, y, q3, y, right).
delta(q1, a, q1, a, right).
delta(q1, y, q1, y, right).
delta(q1, b, q2, y, left).
delta(q2, y, q2, y, left).
delta(q2, a, q2, a, left).
delta(q2, x, q0, x, right).
delta(q3, b, q3, b, right).
delta(q3, y, q3, y, right).
delta(q3, bl, qa, bl, left).
```

```
% General code for all Turing machines
turing(W, Tape) :-           % W: input string, Tape: at the end
   write('   '), write(q0), write(W), nl, % initial configuration
   move([], L, q0, W, R),      %  ([], q0, W) |-* (rev(L), qa, R)
   reverse(L, Lr),
   append(Lr, R, Tape).

move(L, L, qa, R, R) :- !.           % qa is the final state
move(L0, L, Q, R0, R) :-
  current(R0, A, Rest),              % load the current symbol in A
  delta(Q, A, P, B, Direct),         % delta(Q, A) = (P, B, Direct)
  result(Direct, L0, L1, B, Rest, R1),    % update L and R
  write('|- '), write(rev(L1)), write(P), write(R1), nl,
  move(L1, L, P, R1, R).             % continue to move

current([], bl, []).                 % current symbol is a blank bl
current([Sym|Rest], Sym, Rest).      % current symbol is Sym

result(left, [C|L], L, B, R, [C,B|R]). % move top symbol of L to R
result(right, L, [B|L], B, R, R).      % push B at the top of L

?- turing([a,a,b,b], Tape).
   q0[a,a,b,b]
|- rev([x])q1[a,b,b]
|- rev([a,x])q1[b,b]
|- rev([x])q2[a,y,b]
|- rev([])q2[x,a,y,b]
|- rev([x])q0[a,y,b]
|- rev([x,x])q1[y,b]
|- rev([y,x,x])q1[b]
|- rev([x,x])q2[y,y]
|- rev([x])q2[x,y,y]
|- rev([x,x])q0[y,y]
|- rev([y,x,x])q3[y]
|- rev([y,y,x,x])q3[]
|- rev([y,x,x])qa[y,bl]
```

This illustrates how any computation can be expressed declaratively as a sequence of Turing machine moves, implemented in Prolog as a relation between successive states of interest. The Prolog code may be used as a debugging tool for designing Turing machines. □

Theorem 9.3.3. *The termination of a Prolog program is undecidable.*

Proof. (sketch) If we have an algorithm to decide the termination of a Prolog program on any input, then we can use the algorithm to decide the termination of a Turing machine on an input, as the moves of the Turing machine can be simulated by a Prolog program. The termination of the Prolog program in the simulation ensures the termination of the Turing machine. □

Theorem 9.3.4. *The satisfiability of a set of Horn clauses is undecidable.*

Proof. (sketch) If we have an algorithm to decide the satisfiability of a Horn clauses, then the algorithm can be used to decide the clauses in the Prolog program for simulating a Turing machine, together with the query derived from q_0w, is satisfiable or not. If it is unsatisfiable, then there exists a resolution proof derived from the query as this is the only negative clause (the clauses without this one is satisfiable). The resolution proof ensures that w is accepted by the Turing machine. Thus, the algorithm can tell if the Turing machine can accept w or not. This is a contradiction to the fact that the acceptance problem of Turing machines is undecidable. □

Theorem 9.3.5. *The satisfiability of a first-order formula is undecidable.*

The proof is left as an exercise.

9.3.2 Turing Completeness of Rewrite Systems

In theoretical computer science and mathematical logic, a *string rewrite system* (SRS), historically called a *semi-Thue system*, is a rewrite system over strings from an alphabet. Given a binary relation $\rightarrow$ between fixed strings over the alphabet, called *rewrite rules*, denoted by $s \rightarrow t$, an SRS extends the rewriting relation to all strings in which the left- and right-hand side of the rules appear as substrings, that is $usv \rightarrow utv$, where s, t, u, and v are strings.

Proposition 9.3.6. *SRS is Turing complete.*

Proof. Any configuration of a Turing machine is a string of tape and state symbols, assuming the states and the tape do not share symbols. We may use an SRS to

describe all the moves of a Turing machine as follows: If $\delta(q, a) = (p, b, R)$, we have a single rewrite rule $qa \to bp$; if $\delta(q, a) = (p, b, L)$, we have a set of rewrite rules: $cqa \to pcb$, where c is any tape symbol. Then it is trivial to show that $q_0 w \vdash^* \alpha q_a \beta$ iff $q_0 w \Rightarrow^* \alpha q_a \beta$. □

A term rewrite system (TRS) is a binary relation over terms. TRS are more expressive than SRS as an SRS is easily converted into a TRS, where each symbol f in the SRS is replaced by a unary function $f/1$ in the TRS. For example, a rewrite rule $abc \to cd$ in SRS is replaced by $a(b(c(x))) \to c(d(x))$. Since SRS is Turing complete, so are the term rewrite systems.

Theorem 9.3.7. *The termination of a rewrite system is undecidable.*

Proof. (sketch) If we have an algorithm to decide the termination of a rewrite system (either SRS or TRS), then we can use the algorithm to decide the termination of a Turing machine as the moves of the Turing machine can be expressed as a set of rewrite rules. The termination of the rewrite system ensures the termination of the Turing machine. □

9.4 Exercise Problems

1. Some Turing machines allow the tape head stays after a move (only changing state or the current symbol). How to simulate this action in the Prolog program of section 9.3.1?

2. Some Turing machines allow the tape has infinite symbols on the left side of the head tape (so there is no "move off the tape"). How to simulate this action in the Prolog program of section 9.3.1?

3. Provide a complete design of Turing machine M_2 for recognizing $L_2 = \{b^i a^j b^k \mid i * j = k, i, j \geq 0\}$. Test your code in the Prolog program on the input string $w = b^2 a^1 b^2$.

4. Provide a high-level description of a Turing machine M_3 for recognizing $L_3 = \{a^i b^j c^k \mid k = \lfloor i/j \rfloor, i, j \geq 0\}$.

5. Prove that the satisfiability of a first-order formula is undecidable.

CHAPTER 10
HOARE LOGIC

One of the most important research objectives in computer science and engineering is to obtain means of reasoning about computer systems. Proving properties of such systems is extremely important. A 2003 study commissioned by the Department of Commerce's National Institute of Standards and Technology found that software bugs cost the US economy $59.5 billion annually. In safety-critical systems even small mistakes can provoke disasters. Since the amount of data which has to be handled in all the application domains is usually huge, computer support is necessary. Many approaches have been proposed to formally verify the properties of software and hardware systems. For software formal verification, Hoare logic is one of the most influential approaches for proving the correctness of software designs in various programming languages.

Hoare logic (also known as *Floyd–Hoare logic* or *Hoare rules*) is a formal system with a set of logical rules for reasoning rigorously about the correctness of imperative programs, where a program describes steps that change the state of the computer. It was proposed in 1969 by the British computer scientist and logician Tony Hoare, and subsequently refined by Hoare and other researchers. The original ideas were seeded by the work of Robert Floyd, who had published a similar system for flowcharts.

We will use the word "state" to denote a valuation of all variables in an imperative program. A command in the program may change the value of variables, thus causing a transition of states. For example, suppose x holds the value 1 before the command "$x := x+1$", the value of x will be 2 after the execution of $x := x+1$. A state can be easily specified by a first-order formula, called *assertions*, such as "$x = 1$", "$A[0] = 2$", etc.

Executing an imperative program has the effect of changing the state. To use such a program, one first establishes an initial state by specifying the initial values of variables. One then executes the program and this transforms the initial state into a final one through a sequence of state transitions. One then inspects the values of variables in the final state to get the desired results. For example, to compute "$z := exp(x, y)$", the initial state specifies the value of x and y, and the final state specifies the value of z satisfying "$z = x^y$" to judge that the program $exp(x, y)$ is correct.

Hoare logic is an axiomatic approach to formal verification, which consists of (1) a first-order logic language for making assertions about programs; and (2) rules for establishing state transitions, that is, what assertions hold after a transition of states. From the requirement on the input, we create a precondition of a program, and from the requirement on the output, we create a postcondition of the program. Both the precondition and the postcondition are assertions, i.e., formulas of the first-order logic. We then use a theorem prover to prove that the postcondition is a logical consequence of a formula derived from the precondition and the program code by the rules of Hoare logic. Other approaches to formal verification include model checking and abstract interpretation, which are directed at finding specific errors, such as the possible violation of a safety property.

10.1 Hoare Triples

The central idea of Hoare logic is *Hoare triples*, a notation proposed first by Hoare, which specify exactly what precondition that a program asks before the execution and what postcondition, i.e., the desired behavior of the program, that the program will achieve. A Hoare triple describes how the execution of a piece of code changes the state of the computation. A *Hoare triple* is of the form

$$\{P\}\, C\, \{Q\}$$

where P and Q are assertions, which are formulas in a first-order language, and C is a program code. P is named the *precondition* and Q the *postcondition*: When the precondition is met, executing the code C establishes the postcondition. For instance, $\{x = 1\}x := x + 1\{x = 2\}$ is true, because the value of x changes from 1 to 2 after the execution of $x := x+1$. On the other hand, $\{x = 1\}x := x+1\{x = 0\}$ is false. Since a triple may be true or false, we call it a *statement* in the general sense.

Example 10.1.1. The following code computes the sum of all numbers in an integer array.

{ P: A is an array of n integers, and $n \geq 0$ }
$s := 0$;
for $i := 0$ **to** $n - 1$ **do**
    $s := s + A[i]$;
{ Q: $s = \sum_{i=0}^{n-1} A[i]$ }

□

At first glance, P and Q are not the formulas we saw in the previous chapters. An array is not different from a list in logic. In Prolog, we have seen that a list is just a compound term. Thus, it is not strange to represent an array by a term in first-order logic, and we may simply treat an array as a term in the logic. Similarly, we can also treat $\sum_{i=0}^{n-1} A[i]$ as another term. In fact, we may define a function $sum : N, Array \longrightarrow N$ recursively as follows:

$$\begin{aligned} sum(0, A) &= 0; \\ sum(i+1, A) &= sum(i, A) + A[i] \end{aligned}$$

Then $\sum_{i=0}^{n-1} A[i]$ is equivalent to the term $sum(n, A)$.

Now suppose we can specify preconditions and postconditions in first-order logic, how do we prove that a postcondition is a theorem from the precondition and the program? This is handled by the Hoare rules for each type of commands in a small programming language. In this language, we consider only three types of commands:

1. assignment, i.e., $x := E$;
2. conditional, i.e., **if** B **then** S_1 **else** S_2 **fi**; and
3. while loop, i.e., **while** B **do** S **od**.

Commands are separated by semicolons. This language is small but covers the essential part of most imperative programming languages. There are many variations of the loop construct in imperative programming languages: the "while" form, which uses a continuation condition; the "do-until" form, which always executes the loop body at least once, testing for the condition at the end rather than on entry; and the "for" or "do" forms, which iterate over an integer interval or a data structure. They can all be expressed in a straightforward way as a while loop, on which we will rely throughout this chapter. For instance, a *for-loop* can be expressed by a *while-loop*. The for-loop in Example 10.1.1 can be replaced by the equivalent while loop.

$s := 0;$
for $i := 0$ **to** $n - 1$ **do**
    $s := s + A[i]$
od

$s := 0; i := 0;$
while $(i < n)$ **do**
    $s := s + A[i]; i := i + 1$
od

10.1.1 Hoare Rules

Hoare rules cover the three basic commands in the small programming language, plus one rule for the composition of commands and one rule for logical implications.

10.1.1.1 Assignment Rule

The assignment rule states that, after the assignment, any predicate that was previously true for the right-hand side of the assignment now holds for the variable. Formally, let $P(x)$ be an assertion in which the variable x is free. Then:

$$\frac{\top}{\{P[x \leftharpoonup E]\}\, x := E\, \{P(x)\}}$$

where $P[x \leftharpoonup E]$ denotes the assertion obtained from $P(x)$ in which each free occurrence of x is replaced by the expression E.

The assignment rule means that the truth of $P[x \leftharpoonup E]$ is equivalent to the after-assignment truth of $P(x)$. Thus if $P[x \leftharpoonup E]$ is true prior to the assignment, by the assignment rule, then $P(x)$ would be true after the assignment. All preconditions that are not modified by the assignment can be carried over to the postcondition.

Examples of valid triples include: $\{x = 4\}\, y := x + 1\, \{y = 5\}$ and $\{x \leq N - 1\}\, x := x + 1\, \{x \leq N\}$. In the first example, to show that $y = 5$ is true, we need the precondition $x + 1 = 5$, which is equivalent to $x = 4$. Since assigning $y := x + 1$ does not change the fact that $x = 4$, so both $x = 4$ and $y = 5$ may appear in the postcondition. Formally, this result is obtained by applying the assignment rule with $P(y)$ being $x = 4 \wedge y = 5$ and $P[y \leftharpoonup x + 1]$ is $x = 4 \wedge x + 1 = 5$, which can in turn be simplified to the given precondition $x = 4$.

Since we have not given a formal definition of "state", we cannot prove rigorously the correctness of the assignment rule. However, we may argue informally. To show that $\{P[x \leftharpoonup E]\}\, x := E\, \{P\}$ is sound, let s be the state before $x := E$ and s' the state after. So, $s' = s[x \mapsto E]$ (assuming E has no side-effect, i.e., no variables changed values by E). $P[x \leftharpoonup E]$ holds in s iff P holds in $s' = s[x \mapsto E]$, because

1. Every variable, except x, has the same value in s and s',
2. $P[x \leftharpoonup E]$ has every x in P replaced by E, and
3. P has every x evaluated to E in $s' = s[x \mapsto E]$.

The assignment rule is equivalent to saying that to find the precondition, first take the postcondition and replace all occurrences of the left side of the assignment

with the right side of the assignment. Be careful not to try to do this backwards by following this incorrect way of thinking: $\{P(x)\}\, x := E\, \{P[x \leftharpoonup E]\}$; this rule leads to examples like: $\{x = 4\}\, x := 5\, \{5 = 4\}$. Another incorrect rule looking tempting at first glance is $\{P(x)\}\, x := E\, \{P \wedge (x = E)\}$; it leads to illogical examples like: $\{x = 5\}\, x := x + 1\, \{x = 5 \wedge x = x + 1\}$.

While a given postcondition $P(x)$ uniquely determines the precondition $P[x \leftharpoonup E]$ of an assignment, the converse is not true. That is, different postconditions are available for the same precondition and the same code. For example,

- $\{0 \leq y \wedge y \leq 9\}\, x := y\, \{0 \leq x \wedge x \leq 9\}$,
- $\{0 \leq y \wedge y \leq 9\}\, x := y\, \{0 \leq y \wedge x \leq 9\}$,
- $\{0 \leq y \wedge y \leq 9\}\, x := y\, \{0 \leq y \wedge x \leq 9\}$, and
- $\{0 \leq y \wedge y \leq 9\}\, x := y\, \{0 \leq y \wedge y \leq 9\}$

are valid instances of the assignment rule.

The assignment rule proposed by Hoare can apply to multiple mutual assignments. For example, to swap two elements of an array, some programming languages support the command $A[i], A[j] := A[j], A[i]$. The assignment rule for this type of commands is

$$\frac{\top}{\{P[x \leftharpoonup E_1, y \leftharpoonup E_2]\}\, x, y := E_1, E_2\, \{P(x)\}}$$

10.1.1.2 Implication Rule

The implication rule, sometimes called the consequence rule, is purely logical as it does not involve any command of a programming language, but concerns about the implication relation of logical formulas.

$$\frac{P_1 \to P_2 \qquad \{P_2\}\, S\, \{Q_2\} \qquad Q_2 \to Q_1}{\{P_1\}\, S\, \{Q_1\}}$$

This rule can be split into two rules:

$$\frac{P_1 \to P_2 \qquad \{P_2\}\, S\, \{Q\}}{\{P_1\}\, S\, \{Q\}} \qquad\qquad \frac{\{P\}\, S\, \{Q_2\} \qquad Q_2 \to Q_1}{\{P\}\, S\, \{Q_1\}}$$

We say formula A is weaker than formula B if $B \models A$, but not $A \models B$. If we regard the entailment relation $\models$ as an order $\succ$ on formulas, then $\bot$ is the strongest

(or the largest) and $\top$ is the weakest (or the least). We say P is the *weakest precondition* for program S and the postcondition Q if (a) $\{P\}S\{Q\}$ is true; and (b) for any precondition P', if $\{P'\}S\{Q\}$ is true, then $P' \models P$.

The implication rule allows us to strengthen the precondition P_2 to P_1, i.e., assume more than we need, and to weaken the postcondition Q_2 to Q_1, i.e., conclude less than we are allowed to.

For instance, if we have shown that $\{n > 0\}\, sort(A)\, \{n > 0 \wedge sorted(A)\}$, and the target assertion is $sorted(A)$, since $n > 0 \wedge sorted(A) \rightarrow sorted(A)$, the implication rule allows us to get $\{n > 0\}\, sort(A)\, \{sorted(A)\}$.

10.1.1.3 Conditional Rule

$$\frac{\{B \wedge P\}\, S\, \{Q\} \qquad\qquad \{\neg B \wedge P\}\, T\, \{Q\}}{\{P\}\, \textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ fi}\, \{Q\}}$$

The conditional rule states that a postcondition Q common to the two branches of an if-then-else statement is also a postcondition of the whole if-then-else statement, i.e., **if** B **then** S **else** T **fi** $\{Q\}$.

In the then-part and the else-part, B and $\neg B$ are added to the precondition P, respectively. The condition B must not have side-effect, i.e., it does not change the variables during the computation of E.

Example 10.1.2. To show the following triple,

$$\{P : \top\}\, \textbf{if } x < 10 \textbf{ then } x := 1 \textbf{ else } x := 10 \textbf{ fi}\, \{Q : x = 1 \vee x = 10\},$$

is true by the conditional rule, we need to show

$$\begin{array}{ll} (1) & \{P_1 : \top \wedge x < 10\}\, x := 1\, \{Q : x = 1 \vee x = 10\}, \\ (2) & \{P_2 : \top \wedge x \geq 10\}\, x := 10\, \{Q : x = 1 \vee x = 10\}. \end{array}$$

To prove (1) and (2), we obtain the following by the assignment rule:

$$\begin{array}{ll} (1') & \{1 = 1 \vee 1 = 10\}\, x := 1\, \{Q : x = 1 \vee x = 10\}, \\ (2') & \{10 = 1 \vee 10 = 10\}\, x := 10\, \{Q : x = 1 \vee x = 10\}. \end{array}$$

Since $(1 = 1 \vee 1 = 10) \equiv \top$ and $P_1 \models \top$, by the implication rule, (1) is true. (2) is true by the same reasoning. $\square$

If the else-part is missing or T is *skip* or "$x := x$", the conditional rule becomes:

$$\frac{\{B \wedge P\}S\{Q\} \qquad\qquad \neg B \wedge P \rightarrow Q}{\{P\}\, \textbf{if } B \textbf{ then } S \textbf{ fi}\, \{Q\}}$$

Example 10.1.3. To show the following statement

$$\{P : x = a \wedge y = b\} \textbf{ if } x > y \textbf{ then } y := x \textbf{ fi } \{Q : y = max(a, b)\}$$

is true, by the conditional rule, we need to show

(1) $\{P_1 : x = a \wedge y = b \wedge x > y\}\, y := x\, \{Q : y = max(a, b)\}$;
(2) $(x = a \wedge y = b \wedge \neg(x > y)) \rightarrow y = max(a, b)$.

From the assignment rule, we have

$$\{x = max(a, b)\}\, y := x\, \{Q : y = max(a, b)\}$$

To prove (1), by the implication rule, we need to show $P_1 \rightarrow x = max(a, b)$, which can be simplified by equality crossing as $x > y \rightarrow x = max(x, y)$, which holds for all x and y. (2) can also be simplified by equality crossing to $\neg(x > y) \rightarrow y = max(x, y)$, which is true in arithmetic, too. □

10.1.1.4 Iteration Rule

$$\frac{\{I \wedge B\}\, S\, \{I\}}{\{I\} \textbf{ while } B \textbf{ do } S \textbf{ od } \{\neg B \wedge I\}}$$

Here I is the so-called *loop invariant*, which is true at the beginning of the loop and is preserved by the loop body S. That is, after the loop is finished, this invariant I still holds. If the loop condition B becomes false, $\neg B$, which is called the *exit condition* of the loop, is added to I as the postcondition of the loop. As in the conditional rule, B must not have side effects.

Example 10.1.4. To show that the following statement is true,

$$\{I : x \leq 5\} \textbf{ while } x < 5 \textbf{ do } x := x + 1 \textbf{ od } \{Q : x = 5\}$$

we apply the iteration rule and show that

$$\{x \leq 5 \wedge x < 5\}\, x := x + 1\, \{I : x \leq 5\}$$

By the assignment rule, we have $\{x + 1 \leq 5\}\, x := x + 1\, \{I : x \leq 5\}$, so we just need to prove $(x \leq 5 \wedge x < 5) \rightarrow (x + 1 \leq 5)$ by the implication rule. When x is an integer, $(x < 5) \rightarrow (x + 1 \leq 5)$ is true. So $(x \leq 5 \wedge x < 5) \rightarrow (x + 1 \leq 5)$ is true. Finally, we need to show $(I \wedge \neg B) \rightarrow Q$ by the implication rule. Since $x \leq 5 \wedge \neg(x < 5) \rightarrow x = 5$ is true for all number x, the program is correct when x is an integer. □

10.1.1.5 Sequence Rule

The assignment, conditional and iteration rules cover the three basic commands of the small programming language. The sequence rule applies to sequentially executed programs S and T, where S executes prior to T and is written as $S;T$.

$$\frac{\{P\}\, S\, \{Q\} \qquad\qquad \{Q\}\, T\, \{R\}}{\{P\}\, S;T\, \{R\}}$$

where Q is called the *middle assertion*. For example, consider the following two instances of the assignment axiom: $\{x = 4\}\, y := x + 1\, \{y = 5\}$ and $\{y = 5\}\, z := y\, \{z = 5\}$. By the sequence rule, one concludes: $\{x = 4\}\, y := x + 1; z := y\, \{z = 5\}$.

10.1.2 Examples of Formal Verification

Now, we have all the rules needed to show the correctness of any program in the small programming language.

Example 10.1.5. Consider the program in Example 10.1.1:

$\{n \geq 0\}$ $\qquad\qquad$ $sum(0, A) = 0$
$s := 0; i := 0;$ $\qquad\qquad$ $sum(i + 1, A) = sum(i, A) + A[i]$
while $(i < n)$ **do**
$\quad s := s + A[i]; i := i + 1$
od
$\{s = sum(n, A)\}$

Since the code contains a while loop, a crucial step in applying the Hoare logic is to find a loop invariant. Examining the body of the loop, we can see that the values of s and i are changed inside the body. A good loop invariant must reflects these changes. Since the variable s accumulates the sum of the first i elements of A, the first guess for a suitable loop invariant would be $I : s = sum(i, A)$.

By the sequence rule, we need to prove the following statements:

- (1) $\{n > 0\}\, s := 0; i := 0\, \{I\}$;
- (2) $\{I\}$ **while** $i < n$ **do** $s := s + A[i]; i := i + 1$ **od** $\{s = sum(n, A)\}$

The proof of (1) is easy as we just need to prove $n > 0 \rightarrow I[i \leftharpoondown 0, s \leftharpoondown 0]$, which comes from $\{I[i \leftharpoondown 0, s \leftharpoondown 0]\}\, s := 0\, \{I[i \leftharpoondown 0]\}$ and $\{I[i \leftharpoondown 0]\}\, i := 0\, \{I\}$.

Since $I[i \leftharpoonup 0, s \leftharpoonup 0]$ is $0 = sum(0, A)$, which is true by the definition of sum, hence (1) is true.

The proof of (2) is reduced to the following two statements by the iteration rule and the implication rule:

- (2.1) $\{I \wedge i < n\}\, s := s + A[i]; i := i + 1\, \{I\}$, and
- (2.2) $I \wedge \neg(i < n) \rightarrow \{s = sum(n, A)\}$.

The way to prove (2.1) is the same as that of (1), and we need to prove the following formula:

$$(2.1.1) \quad (I \wedge i < n) \rightarrow I[i \leftharpoonup i + 1, s \leftharpoonup s + A[i]]$$

Opening I in (2.2.1), we have

$$(s = sum(i, A) \wedge i < n) \rightarrow (s + A[i] = sum(i + 1, A))$$

By the definition of sum, $sum(i + 1, A) = sum(i, A) + A[i] = s + A[i]$, so (2.1.1) is true, thus (2.1) is true.

Opening I in (2.2), we have $(s = sum(i, A) \wedge i \geq n) \rightarrow (s = sum(n, A))$. Since the condition $i \geq n$ cannot give us $i = n$, the proof cannot go through. From this failure, we can see that we need to strengthen the loop invariant by adding $i \leq n$ to it: the new loop invariant I' should be $s = sum(i, A) \wedge i \leq n$. Using I', the proofs of (1) and (2.1) will go through as before and the proof of (2.2) becomes easy as $i \leq n \wedge i \geq n \rightarrow i = n$. □

The above example shows that the loop invariant should contain all the information about how the loop works:

- It reflects what has been done so far together with what remains to be done and should contain all the changing variables;
- It holds at both the beginning and the end of each iteration;
- Together with the negation of the loop condition, it gives the desired result when the loop terminates;
- If a proof cannot go through, we may need to strengthen the loop invariant by adding a condition which is needed in the proof.

Example 10.1.6. Consider the following program which finds the maximal element in an array:

$\{n > 0\}$ $\quad\quad$ $arrMax(1, A) = A[0]$
$m := A[0]; i := 1;$ $\quad\quad$ $arrMax(i + 1, A) = max(arrMax(i, A), A[i])$
while $i < n$ **do**
    if $(m < A[i])$ **then** $m := A[i];$
    $i := i + 1$
od
$\{m = arrMax(n, A)\}$

Examining the body of the loop, we can see that the values of m and i are changed inside the body. A good loop invariant must reflects these changes. Since the variable m contains the maximal value of the first i elements in A, a reasonable guess for a loop invariant would be $I : m = arrMax(i, A) \wedge i \leq n$.

By the sequence rule, we need to prove the following statements:

- (1) $\{n > 0\}\, m := A[0]; i := 1\, \{I\}$;
- (2) $\{I\}$ **while** $i < n$ **do** ... **od** $\{m = arrMax(n, A)\}$

The proof of (1) is easy as we just need to prove $n > 0 \rightarrow I[i \leftharpoonup 1, m \leftharpoonup A[0]]$, which comes from $\{I[i \leftharpoonup 1, m \leftharpoonup A[0]]\}\, m := A[0]\, \{I[i \leftharpoonup 1]\}$ and $\{I[i \leftharpoonup 1]\}\, i := 1\, \{I\}$. Since $I[i \leftharpoonup 0, m \leftharpoonup A[0]]$ is $A[0] = arrMax(1, A) \wedge 1 \leq n$, which is true by the definition of $arrMax$ and $n > 0$, hence (1) is true.

The proof of (2) is reduced to the following two statements by the iteration rule and the implication rule:

- (2.1) $\{I \wedge i < n\}$ **if** $(m < A[i])$ **then** $m := A[i]; i := i + 1\, \{I\}$, and
- (2.2) $I \wedge \neg(i < n) \rightarrow \{m = arrMax(n, A)\}$.

The proof of (2.1) is focused on the proof of the following statement:

(2.1.1) $\quad \{I \wedge i < n\}$ **if** $(m < A[i])$ **then** $m := A[i]$ $\{I[i \leftharpoonup i + 1]\}$

Applying the conditional rule, we need to prove the following two statements:

- (2.1.1.1) $\{I \wedge i < n \wedge m < A[i]\}\, m := A[i]\, \{I[i \leftharpoonup i + 1]\}$, and
- (2.1.1.2) $(I \wedge i < n \wedge \neg(m < A[i])) \rightarrow I[i \leftharpoonup i + 1]$.

Applying the assignment and application rules to (2.1.1.1), we need to prove

(2.1.1.1.1) $(I \wedge i < n \wedge m < A[i] \rightarrow I[i \leftharpoonup i + 1, m \leftharpoonup A[i]]$

Opening I in (2.1.1.1.1), we need to show that

$$(m = arrMax(i, A) \wedge i \leq n \wedge i < n \wedge m < A[i]) \rightarrow (A[i] = arrMax(i{+}1, A) \wedge (i{+}1) \leq n).$$

Note that $(i + 1) \leq n$ is true because $i < n$ and

$$\begin{array}{rcll} arrMax(i+1, A) & = & max(arrMax(i, A), A[i]) & \text{by the definition of} arrMax \\ & = & max(m, A[i]) & \text{by} m = arrMax(i, A) \\ & = & A[i] & \text{by} m < A[i] \end{array}$$

Hence, (2.1.1.1.1) is true. Using the same approach, (2.1.1.2) can be proved to be true. Hence, (2.1.1) is true.

Opening I in (2.2), we have

$$(m = arrMax(i, A) \wedge i \leq n \wedge \neg i < n) \rightarrow (m = arrMax(n, A)).$$

Since $i \leq n$ and $\neg i < n$ imply that $i = n$, hence $m = arrMax(i, A)$ implies $m = arrMax(n, A)$. Thus, the proofs of (1) and (2) are complete and I is indeed a good loop invariant for showing $m = arrMax(n, A)$. □

Example 10.1.7. The following program will compute the integer quotient.

$\{x, y > 0\}$
$r := x; q := 0;$
while $y \leq r$ **do**
    $r := r - y;$
    $q := q + 1$
od
$\{q = \lfloor x/y \rfloor\}$

Since both r and q are changed inside the loop, let the loop invariant I be $y > 0 \wedge r \geq 0 \wedge x = r + y \cdot q$. By the sequence rule, we need to show the following statements:

- (1) $\{x, y > 0\}\, r := x; q := 0\, \{I\}$;
- (2) $\{I\}$ **while** $y \leq r$ **do** $r := r - y; q := q + 1$ **od** $\{q = \lfloor x/y \rfloor\}$

(1) is easy to prove by using the assignment rules twice, the sequence rule once, and the implication rule once: $x, y > 0 \rightarrow I[r \leftarrow x, q \leftarrow 0]$, that is, $x, y > 0 \rightarrow (y > 0 \wedge x \geq 0 \wedge x = x + y \cdot 0)$ is true.

(2) needs the iteration rule and the implication rule: We need to prove

$$(2.1)\ \{I \wedge y \leq r\}\, r := r - y; q := q + 1\, \{I\};$$
$$(2.2)\ (I \wedge \neg(y \leq r)) \rightarrow (q = \lfloor x/y \rfloor).$$

The way to prove (2.1) is the same as to prove (1), we need to show

$$(I \wedge y \leq r) \rightarrow I[r \leftharpoonup r - y, q \leftharpoonup q + 1]$$

which, when I is open up, is

$$(y > 0 \wedge r \geq 0 \wedge x = r + y \cdot q \wedge y \leq r) \rightarrow (y > 0 \wedge r - y \geq 0 \wedge x = (r - y) + y \cdot (q + 1))$$

Since $(r - y) + y \cdot (q + 1) = r + y \cdot q$ and $y > 0 \wedge y \leq r$ imply $r - y \geq 0$, (2.1) is true.

The proof of (2.2) is easy as $x = r + y \cdot q$ iff $x/y = r/y + q$ when $y > 0$. Since $(y > r \wedge r \geq 0)$ implies $0 \leq r/y < 1$, so $\lfloor x/y \rfloor = \lfloor r/y + q \rfloor = \rfloor r/y \lfloor + q = 0 + q = q$. □

10.1.3 Hoare Rule for Function Calls

In addition to the rules for the simple language in Hoare's original paper, rules for other language constructs have been developed since then by Hoare and many other researchers. There are rules for functions, procedures, jumps, and pointers.

10.1.4 Partial and Total Correctness

Hoare rules presented in the previous section concern about the *partial correctness* of programs: If the program stops, the postcondition will hold. If a program is both partially correct and terminating, we say the program is *totally correct*. The relationship between partial and total correctness can be informally expressed by the equation:

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness}.$$

As shown in the previous chapter, the termination of programs is an undecidable problem and there are no decision procedures to solve this problem in general.

Using the Hoare triples, only partial correctness can be proven, while termination needs to be proved separately. Thus the intuitive reading of a Hoare triple is: Whenever P holds of the state before the execution of C, then Q will hold afterwards, or C does not terminate. In the latter case, there is no "after", so Q can be any statement at all. Indeed, one can choose Q to be false to express that C does not terminate.

Example 10.1.8. Consider the statement $\{T\}$ **while** $x \neq a$ **do od** $\{x = a\}$. Let the loop invariant I be $\top$, then both $I \wedge x \neq a \rightarrow I$ and $I \wedge \neg(x \neq a) \rightarrow x = a$ are trivially true. Thus, the program is partially correct: If the program stops, we must have $x = a$. The program will loop forever unless it starts with $x = a$, therefore it is not totally correct. □

Total correctness can be proved with an extended version of the iteration rule, where the value of a function on the changing variables of the loop decreases under a well-founded order. Total correctness is what we are ultimately interested in, but it is usually easier to prove it by establishing partial correctness and termination separately. Termination is often straightforward to establish, and the complexity analysis of algorithms often provides finer results than the termination. For example, the program in Example 10.1.5 is terminating because the value of $f(i) = n - i$ decreases by one after each iteration and the minimum value of $f(i)$ is 0 when the looping condition $i < n$ becomes false. For the program in Example 10.1.7, let $f(q) = \lfloor x/y \rfloor - q$, then $f(q)$ decreases by one after each iteration and the minimum value of $f(q)$ is 0 when the looping condition is false. The complexity of both algorithms is $O(n)$ ($n = \lfloor x/y \rfloor$ in the second example).

10.2 Automated Generation of Assertions

After seeing only a few examples, the following two things is painfully clear:

- Proofs are typically long and boring, even if the program being verified is quite simple.
- There are lots of fiddly little details to get right, many of which are trivial, e.g. proving $r = x \wedge q = 0 \rightarrow (x = r + y \cdot q)$.

Many attempts have been made (and are still being made) to automate correctness proofs by designing systems to do the boring and tricky bits of generating formal proofs in Hoare logic. Unfortunately logicians have shown that it is impossible in principle to design a decision procedure to decide automatically the truth or falsehood of an arbitrary mathematical statement. However, this does not mean that one cannot have procedures that will prove many useful theorems. The nonexistence of a general decision procedure merely shows that one cannot hope to prove everything automatically. In practice, it is quite possible to build a system that will mechanize many of routine aspects of verification. This section describes one commonly taken approach to doing this.

In the previous section it was shown how to prove $\{P\}\, S\, \{Q\}$ by reducing this goal to several subgoals and then putting these together using the Hoare rules to

get the desired property of S itself. For example, to prove $\{P\}\, S;T\, \{Q\}$, first prove $\{R\}\, T\, \{Q\}$ and $\{P\}\, S\, \{R\}$ for a middle assertion R, and then deduce $\{P\}\, S;T\, \{Q\}$ by the sequence rule. This process is called *goal reduction*, or *backward reasoning*, because one works backwards: Starting from the goal of showing $\{P\}\, S\, \{Q\}$, one generates subgoals, sub-subgoals, etc., until the problem is solved. Prolog also uses this process for computation.

Example 10.2.1. Suppose one wants to show:

$$\{x = x_0 \wedge y = y_0\}\, r := x; x := y; y := r\, \{y = x_0 \wedge x = y_0\}$$

then by the assignment and sequence rules, the above statement is reduced to the subgoal

$$\{x = x_0 \wedge y = y_0\}\, r := x; x := y\, \{r = x_0 \wedge x = y_0\}$$

because $\{r = x_0 \wedge x = y_0\}\, y := r\, \{y = x_0 \wedge x = y_0\}$. By a similar argument this subgoal can be reduced to

$$\{x = x_0 \wedge y = y_0\}\, r := x\, \{r = x_0 \wedge y = y_0\}$$

which is reduced further to $(x = x_0 \wedge y = y_0) \rightarrow (x = x_0 \wedge y = y_0)$. Every step of the above reduction clearly follows from the assignment rule. The middle assertions for the sequence rule are automatically generated. □

We wish that the process illustrated by the above example can be extended to arbitrary programs so that all middle assertions can be generated automatically, releasing programmers from the burden of providing all assertions. A programmer just needs to provide the pre- and post-conditions of the program derived from the specification of the problem and all the needed assertions can be generated automatically. Coupling with a powerful theorem prover, all the formulas derived from Hoare rules can be proved automatically, and we arrive at the automation of software verification.

10.2.1 Verification Conditions

The formal verification of a program can be described succinctly as follows:

1. The program S is annotated by inserting into it the assertions expressing conditions that are meant to hold at various intermediate points. This step needs a good understanding of how the program works. The user needs to provide at least the pre- and post-conditions and loop invariants; other assertions can be generated automatically.

2. A set of formulas called *verification conditions* is then generated from the annotated specification. This process is purely mechanical and easily done by a software tool. We will describe how this done in this section.

3. The verification conditions are proved by a theorem prover. Automating this remains to be a big challenge.

Definition 10.2.2. *Given a Hoare triple* $\{P\}\, S\, \{Q\}$*, let* $vc(S, Q)$ *denote the formula* P' *such that both* (1) $P \to P'$ *and* (2) $\{P'\}\, S\, \{Q\}$ *are true, then* P' *is called a* verification condition *of* S *with respect to* Q*.*

By the implication rule, if a verification condition $vc(S, Q)$ exists, then the triple $\{P\}S\{Q\}$ is true. Moreover, if $vc(S, Q)$ can be generated automatically from S and Q, then the process of generating all assertions becomes automated. Besides being the basis for automated verification systems, verification conditions are a useful way of working with Hoare logic by hand.

We will view $vc(S, Q)$ as a recursive procedure on S, coinciding with the goal reduction process. The definition of $vc(S, Q)$ follows closely with Hoare rules on the three constructs and the sequence rule of our small programming language. We add a *skip* command, which does nothing, so that it is easy to use the if-then-else command for the if-then command: **if** B **then** S **fi** becomes **if** B **then** S **else** *skip* **fi**. In summary, there are only five cases to consider in the definition of $vc(S, Q)$.

In fact, the verification conditions are the *weakest preconditions* in most cases, with the exception of while loops, but we will neither make this more precise nor go into details here. An in-depth study of weakest preconditions can be found in Dijkstra's book.

For while loops, we do not have an algorithm which can generate $vc(S, Q)$ for every loop; only some heuristic methods are available. To help the verification tool to generate $vc(S, Q)$, a programmer needs to suggest a loop invariant. That is, instead of the code **while** B **do** S **od**, we write **while** B **do** $\{I\}$ S **od**, where I is the suggested loop invariant. The pre- and post-conditions can be also inserted into a program code. This type of program codes with assertions is called "annotated program", where assertions are provided by the programmer to assist the verification.

Another exception with while loops is that, in addition to returning a verification condition for the loop, it will create another formula whose validity is needed to ensure the validity of the Hoare triple. This formula will be pushed into a stack denoted by s during the execution of vc. The stack s is a global variable and is initially empty; it will hold as many formulas as the number of while loops in the program.

Algorithm 10.2.3. The algorithm vc will take an annotated program C, a formula Q and returns a formula as a precondition P for $\{P\}\, C\, \{Q\}$ being true.

proc $vc(C, Q)$
1 **if** C is "$x := E$" **return** $Q[x \leftarrow E]$
2 **else if** C is "*skip*" **return** Q
3 **else if** C is "**if** B **then** S **else** T **fi**"
4 **return** $(B \rightarrow vc(S, Q)) \wedge (\neg B \rightarrow vc(T, Q))$
5 **else if** C is "**while** B **do** $\{I\}$ S **od**"
6 $push(I \rightarrow ((B \rightarrow vc(S, I)) \wedge (\neg B \rightarrow Q)), s)$;
7 **return** $((\neg B \wedge Q) \vee I)$
8 **else** C is "$S; T$" **return** $vc(S, vc(T, Q))$

In the above algorithm, lines 1-2 handles the base cases, when the command is an assignment or skip command; Lines 3-4 handle the conditional (for simplicity, we ignore the conditional case when the else part is missing); Lines 5-7 handle the while loop; and line 8 handles the sequence of commands. The last three cases involve the recursive calls of vc.

Theorem 10.2.4. *The output of $vc(S, Q)$ is a formula P' in first-order logic such that (a) $P \models P'$ and (b) all the formulas generated by $vc(S, Q)$ in the stack s are valid, then $\{P\}\, S\, \{Q\}$ is true.*

The following subsection will be devoted the proof of the above theorem.

10.2.2 Proof of Theorem 10.2.4

At first, we need to show that vc will terminate. This is an easy task because the first argument of vc, which is a piece of code, becomes strictly smaller for each recursive call, and it will come down to an assignment or a skip command, there cannot exist an infinite chain of recursive calls. So the algorithm vc must terminate.

The proof of the above theorem will be by induction on the structure of C. Such inductive arguments have two parts. First, it is shown that the result holds for the assignment and skip commands. Second, it is shown that when C is not an assignment command, then if the result holds for the constituent commands of C (this is called the induction hypothesis), then it holds also for C. The first of these parts is called the base cases of the induction and the second is called the inductive cases. From the base and the inductive cases, it follows that the result holds for all commands. For instance, to show that vc returns a formula, since all the base cases return a formula, assuming all the recursive calls will return a formula, then vc will return a formula by examining all the inductive cases.

The case of the skip command is trivial; the other cases will be discussed separately.

10.2.2.1 Assignments

The only verification condition for $x := E$ and Q is $Q[x \leftharpoonup E]$, which is the weakest precondition for $x := E$ and Q. Thus, we have $vc(x := E, Q) = Q[v \leftharpoonup E]$.

10.2.2.2 Sequences

The only verification condition for $S; T$ and Q is $vc((S; T), Q) = vc(S, vc(T, Q))$. If both $vc(S, Q')$ and $vc(T, Q)$ are the weakest preconditions for S and T, respectively, then $vc(S, vc(T, Q))$ is the weakest precondition for $S; T$.

Example 10.2.5. The verification condition for the code in Example 10.2.1 is easily obtained as it involves only two cases: assignment and sequence.

$$\begin{aligned} & vc((r := x; x := y; y := r), y = x_0 \wedge x = y_0) \\ = \;& vc((r := x; x := y), vc(y := r, y = x_0 \wedge x = y_0)) \\ = \;& vc((r := x; x := y), r = x_0 \wedge x = y_0) \\ = \;& vc(r := x, vc(x := y, r = x_0 \wedge x = y_0)) \\ = \;& vc(r := x, r = x_0 \wedge y = y_0) \\ = \;& x = x_0 \wedge y = y_0 \end{aligned}$$

□

10.2.2.3 Conditionals

Let $P' = vc(\textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ fi}, Q)$, by the definition of vc, $P' = (B \rightarrow vc(S, Q)) \wedge (\neg B \rightarrow vc(T, Q))$. Note that P' is the weakest precondition for the conditional command, if both $vc(S, Q))$ and $vc(T, Q))$ are the weakest preconditions for S and T, respectively. By the induction hypothesis, both $\{vc(S, Q)\}S\{Q\}$ and $\{vc(T, Q)\}T\{Q\}$ are true. If $P \rightarrow P'$ is true, then both $P \rightarrow (B \rightarrow vc(S, Q))$ and $P \rightarrow (\neg B \rightarrow vc(T, Q))$ are true. Since $P \rightarrow (B \rightarrow vc(S, Q)) \equiv (P \wedge B) \rightarrow vc(S, Q)$, by the implication rule, $\{P \wedge B\}\, S\, \{Q\}$. Following the same reasoning, we have $\{P \wedge \neg B\}\, T\, \{Q\}$. By the conditional rule,

$$\{P\}\, \textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ fi}\, \{Q\}$$

is true.

Example 10.2.6.

$$\begin{aligned} & vc(\textbf{if } (x < 10) \textbf{ then } x := 0 \textbf{ else } x := 10 \textbf{ fi}, x = 0 \vee x = 10) \\ = \; & (x < 10 \rightarrow vc(x := 0, x = 0 \vee x = 10)) \wedge (x \geq 10 \rightarrow vc(x := 10, x = 0 \vee x = 10)) \\ = \; & (x < 10 \rightarrow 0 = 0 \vee 0 = 10) \wedge (x \geq 10 \rightarrow 10 = 0 \vee 10 = 10) \\ = \; & (x < 10 \rightarrow \top) \wedge (x \geq 10 \rightarrow \top) \\ = \; & \top \end{aligned}$$

□

10.2.2.4 Iterations

Let $P' = vc(\textbf{while } B \textbf{ do } \{I\}\, S \textbf{ od}, Q)$, by the definition of vc, P' is $(\neg B \wedge Q) \vee I$, which is a disjunction of two formulas: (a) $\neg B \wedge Q$, and (b) I. (a) covers the case when B is false in the beginning and the while loop is the same as a skip command. (a) can be omitted if we let vc to return a stronger precondition. (b) ensures that I holds before entering the loop.

Moreover, the formula $I \rightarrow ((B \rightarrow vc(S, I)) \wedge (\neg B \rightarrow Q))$, which is equivalent to $(I \wedge B \rightarrow vc(S, I)) \wedge (I \wedge \neg B \rightarrow Q)$, is created and stored in the stack s. The validity of $I \wedge B \rightarrow vc(S, I)$ ensures that the iteration rule can be applied. By the induction hypothesis, $\{vc(S, I)\}\, S\, \{I\}$ is true. Since $I \wedge B \rightarrow vc(S, I)$, by the implication rule, $\{I \wedge B\}\, S\, \{I\}$ is true, that is, I is indeed a loop invariant. Thus, by the iteration rule, $\{I\}\, \textbf{while } B \textbf{ do } S \textbf{ od}\, \{I \wedge \neg B\}$ is true. Finally, the validity of $(I \wedge \neg B) \rightarrow Q$ ensures that Q holds when the loop terminates. By the implication rule, $\{I\}\, \textbf{while } B \textbf{ do } S \textbf{ od}\, \{Q\}$ holds. This completes the proof of Theorem 10.2.4. □

The algorithm $vc(S, Q)$ shows how assertions needed for Hoare rules are derived from the postcondition and loop invariants automatically in a goal-reduction or backward reasoning process. Backward reasoning can be useful for nested loops, where the inner loop's postcondition can be derived from the outer loop's invariant.

Example 10.2.7. The precondition for the following program is $P : x = n \wedge y = 1$ and the loop invariant is $I : x! \cdot y = n!$, where $x!$ is the factorial function of x.

$$\begin{aligned} & vc(\textbf{while } (x \neq 0) \textbf{ do } \{I : x!y = n!\}\, y := x * y; x := x - 1 \textbf{ od}, y = n!) \\ = \; & (x = 0 \wedge y = n!) \vee (x!y = n!) \end{aligned}$$

The formula pushed into the stack and needs to be proved is

$$\begin{aligned}
& x!y = n! \to ((x! = 1 \to vc([y := y * x; x := x - 1], x!y = n!)) \wedge (x = 1 \to y = n!)) \\
= \;& x!y = n! \to ((x! = 1 \to vc(y := y * x, vc(x := x - 1, x!y = n!))) \wedge (x = 1 \to y = n!)) \\
= \;& x!y = n! \to ((x! = 1 \to vc(y := y * x, (x - 1)!y = n!)) \wedge (x = 1 \to y = n!)) \\
= \;& x!y = n! \to ((x! = 1 \to (x - 1)!(yx) = n!) \wedge (x = 1 \to y = n!)) \\
= \;& x!y = n! \to ((x! = 1 \to x!y = n!) \wedge (x = 1 \to y = n!)) \\
= \;& x!y = n! \to (x = 1 \to y = n!)
\end{aligned}$$

□

The last step used the equivalence $X \to ((Y \to X) \wedge Z) \equiv X \to Z$, where X is $x!y = n!$, Y is $x! = 1$, and Z is $x = 1 \to y = n!$.

Can we get rid the stack used in vc? That is, instead of storing $I \to ((B \to vc(S, I)) \wedge (\neg B \to Q))$, can we let $vc(\textbf{while } B \textbf{ do } \{I\}\, S \textbf{ od}, Q)$ returns $I \to ((B \to vc(S, I)) \wedge (\neg B \to Q))$ as part of its output? The following counterexample shows that this approach is unsound.

Example 10.2.8. Let the Hoare triple be

$$\{x = 0\}\textbf{while } (x < 5) \textbf{ do } x := x + 1 \textbf{ od } \{x = 5\},$$

which can be proved to be true by the Hoare rules with the loop invariant $x \leq 5$.

If we modify vc such that $vc(\textbf{while } B \textbf{ do } \{I\}\, S \textbf{ od}, Q)$ returns

$$(\neg B \wedge Q) \vee I \wedge (I \to ((B \to vc(S, I)) \wedge (\neg B \to Q))),$$

which is equivalent to

$$(\neg B \wedge Q) \vee I \wedge (B \to vc(S, I)) \wedge (\neg B \to Q).$$

Let I be $x = 0 \vee x = 1$ for the above program, then $vc(\textbf{while } (x < 5) \textbf{ do } \{I\}\ x := x + 1 \textbf{ od}, x = 5)$ returns

$$(x \geq 5 \wedge x = 5) \vee I \wedge (x < 5 \to vc(x := x + 1, I)) \wedge (x \geq 5 \to x = 5).$$

which can be is simplified to

$$P' : x = 5 \vee (x = 0 \vee x = 1) \wedge (x < 5 \to (x = -1 \vee x = 0)) \wedge (x \geq 5 \to x = 5),$$

because $vc(x := x + 1, I)$ returns $(x + 1 = 0 \vee x + 1 = 1)$, which is equivalent to $x = -1 \vee x = 0$. Now it is easy to show (by equality crossing) that $x = 0 \to P'$ is true (since $P'[x \leftarrow 0]$ is true). By the original Hoare rule, we cannot show that $x = 0 \vee x = 1$ is a valid loop invariant. Thus, the modification of vc is unsound. □

10.2.3 Implementing vc in Prolog

Since Prolog is good at pattern matching, it is easy to implement the algorithm $vc(S, Q)$ in Prolog. The first job is to let the Prolog program parses codes easily. For this purpose, we change the syntax of our programming language as follows:

command	**before**	**now**
assignment	$x := E$	$\text{let}(x, E)$
conditional	**if** B **then** S **else** T **fi**	$\text{ite}(B, S, T)$
iteration	**while** B **do** $\{I\}$ S **od**	$\text{while}(B, I, S)$
skip	skip	skip
sequence	$S_1; S_2; ...; S_n$	$[S_1, S_2, ..., S_n]$

For the iteration, instead of using a stack to store formulas, we simply write the formulas on the screen. Below is the Prolog code for treating assignments, conditionals, iterations, and skip:

```
% vc1(S1, Post, VC): VC = vc(S1, Post), S1 is a single command

% assignment
vc1(let(X, E), Post, VC) :- substitute(X, E, Post, VC).

% conditional
vc1(ite(B, S, T), Post, VC) :-
    vc(S, Post, VC1), vc(T, Post, VC2),
    VC = and((B -> VC1), (neg(B) -> VC2)).

% iteration
% use vc1(while(B, Inv, S), Post, Inv) as an alternative.
vc1(while(B, Inv, S), Post, or(and(neg(B), Post), Inv)) :-
    vc(S, Inv, VC),                          % VC is vc(S, Inv)
    write((and(Inv, B) -> VC)), nl,          % part 1 of a conjunction
    write((and(Inv, neg(B)) -> Post)), nl. % part 2 of a conjunction
```

The handling of assignment uses a utility function `substitute`, which performs the substitution:

```
% substitute(X, V, E, R) succeeds when every occurrence of X in
% expression E is replaced by V and the modified E is R.
substitute(X, V, X, V):- !.
```

```
substitute(X, V, E, R) :-
        E =.. [F|Args0],
        maplist(substitute(X, V), Args0, Args),
        R =.. [F|Args].
```

For a sequence S of commands, we first reverse S into S_2 and work on S_2 from the beginning to the end, as illustrated by the following code.

```
% vc(S, Post, VC) succeeds when a VC is generated from
% from the program S and the post condition Post.

vc(P, Post, VC) :- is_list(P), !, reverse(P, P2), vcR(P2, Post, VC).
vc(P, Post, VC) :- vc1(P, Post, VC).     % if S is not a list

% vcR(SR, Post, VC) generates a VC from SR and Post, where
% SR is a list of instructions in reversed order.
vcR([], Post, Post).
vcR([Last | Rest], Post, VC) :-
     vc1(Last, Post, Post1), vcR(Rest, Post1, VC).
```

In the following examples, we define a triple $tri_i(P, S, Q)$, where i is a number. We then define q_i for the query of generating and displaying a verification condition for the triple $tri_i(P, S, Q)$. The validity of all the displayed formulas ensures that the triple is true.

Example 10.2.9. The triple and the query are specified as follows:

```
tri1(true,
    [ ite(a>b, let(x, b), let(x, a)), ift(x > c, let(x, c))],
     x=min(a,min(b,c))).

q1 :- tri1(P, S, Q), vc(S, Q, VC), write((P -> VC)), nl.
```

The query `?- q1.` produces one formula:

```
  true->and((a>b->and((b>c->c=min(a,min(b,c))),
                (neg(b>c)->b=min(a,min(b,c))))),
               (neg(a>b)->and((a>c->c=min(a,min(b,c))),
                              (neg(a>c)->a=min(a,min(b,c))))))
```

which represents the formula

$$\top \rightarrow ((a > b \rightarrow ((b > c \rightarrow c = m) \wedge (\neg(b > c) \rightarrow b = m))) \wedge (\neg(a > b) \rightarrow ((a > c \rightarrow c = m) \wedge (\neg(a > c) \rightarrow a = m))))$$

where $m = min(a, min(b, c))$. □

Example 10.2.10. The triple and the query are specified as follows:

```
tri2(and(n>=0, and(k=0, result=0)),
     while(k<n,
           and(0=<k, and(k=<n, result=sum(0, k, a))),
          [let(result, result+a(k)),
           let(k, k+1)]),
     result=sum(0, n, a)).

q2 :- tri2(P, S, Q), vc(S, Q, VC), write((P -> VC)), nl.
```

The query `?- q2` produces three formulas:

```
and(and(0=<k,and(k=<n,result=sum(0,k,a))),k<n)->
   and(0=<k+1,and(k+1=<n,result+a(k)=sum(0,k+1,a)))
and(and(0=<k,and(k=<n,result=sum(0,k,a))),neg(k<n))->
   result=sum(0,n,a)
and(n>=0,and(k=0,result=0))->
   or(and(neg(k<n),result=sum(0,n,a)),
      and(0=<k,and(k=<n,result=sum(0,k,a))))
```

where $sum(b, e, A) = 0$ if $(b \geq e)$; otherwise $sum(b, e, A) = sum(b, e-1, A)+A[e-1]$. The three formulas can be rewritten as

$$(0 \leq k \wedge k \leq n \wedge result = sum(0, k, a) \wedge k < n) \rightarrow (0 \leq k + 1 \wedge k + 1 \leq n \wedge result + a(k) = sum(0, k + 1, a))$$
$$(0 \leq k \wedge k \leq n \wedge result = sum(0, k, a) \wedge \neg(k < n)) \rightarrow result = sum(0, n, a)$$
$$(n \geq 0 \wedge k = 0 \wedge result = 0) \rightarrow ((\neg(k < n) \wedge result = sum(0, n, a)) \vee (0 \leq k \wedge k \leq n \wedge result = sum(0, k, a)))$$

□

Example 10.2.11. The triple and the query are specified as follows:

```
tri3(and(n >= 0, and(x=n, y=1)),
     while(neg(x = 0),
           \{ fac(x)*y = fac(n)\},
```

```
          [let(y, x*y),
           let(x, x-1)]),
     y=fac(n)).

q3 :- tri3(P, S, Q), vc(S, Q, VC), write((P -> VC)), nl.
```

The query `?- q3` produces three formulas:

```
and(fac(x)*y=fac(n),neg(x=0))->fac(x-1)*(x*y)=fac(n)
and(fac(x)*y=fac(n),neg(neg(x=0)))->y=fac(n)
and(n>=0,and(x=n,y=1))->
   or(and(neg(neg(x=0)),y=fac(n)),fac(x)*y=fac(n))
```

where $fac(n)$ is the factorial function: $fac(0) = 1$, $fac(i+1) = fac(i) * (i+1)$. The three formulas can be rewritten as

$(fac(x) * y = fac(n) \wedge \neg(x = 0)) \rightarrow fac(x-1) * (x * y) = fac(n)$
$(fac(x) * y = fac(n) \wedge \neg\neg(x = 0)) \rightarrow y = fac(n)$
$(n >= 0 \wedge x = n \wedge y = 1) \rightarrow ((\neg\neg(x = 0) \wedge y = fac(n)) \vee fac(x) * y = fac(n))$

□

The above three examples show how formulas can be generated from post-conditions and loop invariants. All middle assertions, with the exception of loop invariants, are generated automatically. These formulas are then fed to an automated theorem prover. Once they are proved to be valid, the Hoare triple is shown to be true and the program is partially correct.

More than often, a theorem prover cannot prove these formulas automatically; if it fails, advice is sought from the user. The analysis of the failure may suggest that we would need to add a conjunct to a loop invariant, to the precondition, or modify the postcondition. We then run *vc* with new annotated code, obtain new formulas and feed them to the theorem prover. This process repeats until we find the right pre- and post-conditions and loop invariants.

The aim of much current research is to build systems which reduce the role of the slow and expensive human expert to a minimum. This can be achieved by:

- reducing the number and complexity of the annotations required, and
- increasing the power of the theorem prover.

From the implementation of *vc*, it is now clear that the success of Hoare logic to formal verification relies on (*i*) obtaining a good loop invariant for each while loop in a program; (*ii*) having a powerful theorem prover which can prove the validity of the formulas derived by *vc*. The first issue will be addressed in the next section. The second issue has been addressed in the previous chapters of this book and will be discussed in the next chapter. Since proofs of correctness of programs are typically very long and boring, it is thus useful to check proofs mechanically or automatically, even if they can only be carried out with human assistance.

10.3 Obtaining Good Loop Invariants

Finding an invariant for a loop is traditionally the responsibility of a human: either the person performing the verification or the programmer writing the loop in the first place. A better solution, when applicable, is *program synthesis*, a constructive approach to programming advocated by Dijkstra and others. Automated generation of loop invariants is still an ongoing research topic.

For the triple $\{I\}$ **while** B **do** S **od** $\{Q\}$, I is a correct invariant for the loop if it satisfies the following conditions:

1. I is true in the state preceding the loop execution.
2. Every execution of the loop body S, started in any state in which both I and B are true, will yield a state in which I holds again.

If these properties hold, then any terminating execution of the loop will yield a state in which both I and $\neg B$ hold; the latter is called the *exit condition*.

We may look at the notion of loop invariant from the constructive perspective of a programmer directing his or her program to reach a state satisfying a certain desired property, the postcondition. In this view, program construction is a form of problem solving, and the various control structures are problem-solving techniques; a loop solves a problem through successive approximation. This idea has been strongly advocated in Furia *et al.*'s survey article on loop invariants. In the following, our presentation of the idea is based on their survey.

- Generalize the postcondition (the characterization of possible solutions) into a broader condition: the invariant.
- As a result, the postcondition can be defined as the conjunction of the invariant and another condition, i.e., the exit condition.
- Find a way to reach the invariant from the previous state of the computation: the initialization.

- Find a way, given a state that satisfies the invariant, to get to another state, still satisfying the invariant but closer, in some appropriate sense, to the exit condition: the body.

The importance of this presentation of the loop process is that it highlights the nature of the invariant: it is a generalized form of the desired postcondition, which in a special case (represented by the exit condition) will give us that postcondition. This view of the invariant, as a particular way of generalizing the desired goal of the loop computation, explains why the loop invariant is such an important property of loops; one can argue that understanding a loop means understanding its invariant.

10.3.1 Invariants from Generalizing Postconditons

To illustrate the idea of generalizing the postcondition to obtain an invariant, we will use the thousands-year-old Euclid's algorithm for computing the greatest common divisor (gcd) of two positive integers. The postcondition of the algorithm is

$$y = gcd(a, b),$$

where the precondition is $a > 0$ and $b > 0$. The first step of the generalization is to replace this condition by

$$x = 0 \wedge gcd(x, y) = gcd(a, b)$$

with a new variable x, taking advantage of the mathematical property that, for every y, $gcd(0, y) = y$. The second conjunct, i.e., $gcd(x, y) = gcd(a, b)$, a generalization of the postcondition, will serve as the invariant; the first conjunct will serve as the exit condition. To obtain the loop body, we take advantage of another mathematical property: for every x and y, $gcd(x, y) = gcd(y, x)$ and $gcd(x, y) = gcd(x - y, y)$ if $x > y$, yielding the well-known algorithm:

$x := a; y := b;$
while $x \neq 0$ **do**
  $\{I : x \geq 0 \wedge y > 0 \wedge gcd(x, y) = gcd(a, b)\}$
  if $x < y$ **then** $z := x; x := y; y := z$ **fi**;
  $x := x - y$
od

The proof of correctness follows directly from the mathematical property of *gcd*. This form of Euclid's algorithm uses subtraction; we may replace the subtraction by the remainder function, i.e., replace $x - y$ by $x\%y$, resulting a more efficient algorithm. In this case, we need the property that $gcd(x, y) = gcd(x\%y, y)$.

Using the classification criteria proposed in Furia *et al.*'s survey article, the invariant for the *gcd* algorithm is divided into two categories:

- The last conjunct of the invariant is an *essential* invariant, representing a weakening of the postcondition. The first two conjuncts are *bounding* invariants, indicating that the state remains within certain general boundaries and ensuring that the "essential" part is well defined.
- The strategy that leads to the essential invariant is *uncoupling*, which replaces a property of one variable (y in this case), used in the postcondition, by a property of two variables (y and x), used in the invariant.

As illustrated by the above example, the essential invariant is a mutation (often, a weakening) of the loop's postcondition. The following mutation techniques are particularly common.

- **Constant Relaxation**
  Constant relaxation replaces a constant n (more generally, an expression which does not change during the execution of the algorithm) by a variable i, and use $i = n$ as part or all of the exit condition. In Example 10.1.5, constant relaxation is used to obtain the loop invariant, i.e., $s = sum(i, A)$, from the postcondition $s = sum(n, A)$. This condition is trivial to establish initially for $i = 0$ and $s = 0$, and is easy to extend to an incremented i, and yields the postcondition when i reaches n.
- **Uncoupling** Uncoupling is to replace a variable by two, using their equality as part or all of the exit condition. Uncoupling is used in the gcd algorithm where we replace $y = gcd(a, b)$ by $x = 0 \wedge gcd(x, y) = gcd(a, b)$. Uncoupling is also used in Example 10.2.7 where we replace $y = n!$ by $x = 0 \wedge x!y = n!$.
- **Term dropping** Term dropping removes a subformula (typically a conjunct), which gives a straightforward weakening. Term dropping often applies after uncoupling. $x = 0$ is dropped from $x = 0 \wedge gcd(x, y) = gcd(a, b)$ in the gcd algorithm, and $x = 0 \wedge x!y = n!$ in the factorial algorithm to obtain the essential invariant. The dropped term often goes into the exit condition of the loop.
- **Aging** Aging is to replace a variable (more generally, an expression) by an expression that represents the value the variable had at previous iterations of the loop. Aging typically accommodates "off-by-one" discrepancies between when a variable is evaluated in the invariant and when it is updated in the

loop body. Typically, aging is used for generating bounding invariants. For example, if $i > 0$ appears in the loop condition and $i := i - 1$ appears in the loop body, aging will generate $i + 1 > 0$ (equivalently $i \geq 0$) as a bounding invariant. On the other hand, if $j < n$ appears in the loop condition and $j := j+1$ appears in the body, then aging will generate $j-1 < n$ (equivalently $j \leq n$) as a bounding invariant.

The above generalization techniques can be implemented by heuristic methods and thus generating invariants automatically. The amount of research work on the automated generation of invariants is substantial and spread over more than three decades; this reflects the cardinal role that invariants play in the formal analysis and verification of programs. The methods for automatic generation of invariants can classified as *static* and *dynamic*. Static methods use only the program text, whereas dynamic methods summarize the properties of many program executions with different inputs.

Historically, the earliest methods for invariant inference were static. Abstract interpretation and the constraint-based approach are the two most widespread frameworks for static invariant inference. Abstract interpretation is a symbolic execution of programs over abstract domains that over-approximates the semantics of loop iteration. Constraint-based techniques rely on sophisticated decision procedures over nontrivial mathematical domains (such as polynomials) to represent concisely the semantics of loops with respect to certain template properties. Static methods are sound and often complete with respect to the class of invariants that they can infer. Soundness and completeness are achieved by leveraging the decidability of the underlying mathematical domains they represent; this implies that the extension of these techniques to new classes of properties is often limited by undecidability.

Only in the last decade have dynamic techniques been applied to invariant generation. In a nutshell, the dynamic approach consists of testing a large number of candidate properties against several program runs; the properties that are not violated in any of the runs are retained as "likely" invariants. This implies that the method is not sound but only an "educated guess": dynamic invariant generation is to static generation what testing is to program proofs. Nonetheless, just like testing is quite effective and useful in practice, dynamic invariant generation can work well if properly implemented. To date, dynamic invariant inference has been mostly used to infer pre- and postconditions or intermediate assertions, whereas it has been only rarely applied to loop invariant generation.

10.3.2 Program Synthesis from Invariants

Program synthesis is the task to construct a program that provably satisfies the postcondition. In contrast to program verification, the program is to be constructed rather than given. Both fields make use of formal proof techniques, and both comprise approaches of different degrees of automation. In contrast to automatic programming techniques, postconditions in program synthesis are usually non-algorithmic statements in an appropriate logical language.

If we are capable of generating loop invariants from the postcondition automatically, then it is relatively easy to construct the code that satisfies the postcondition. Since different algorithms for solving the same problem require different invariants, generating different invariants lead us to different algorithms.

Suppose we like to sort an array A of n elements. Let A_0 denote the original array A. The postcondition of a sorting algorithm is $perm(A, A_0) \wedge sorted(A, 0, n)$, where $perm(A, A_0)$ is true iff A is a permutation of A_0 and $sorted(A, 0, n)$ is true iff the elements of A from 0 to $n-1$ are sorted. Suppose the only way to change A is by $swap(A, i, j)$, which swaps the elements $A[i]$ and $A[j]$, then $perm(A, A_0)$ will remain true through the algorithm. That is, the triple

$$\{permu(A, A_0)\}\, swap(A, i, j)\, \{permu(A, A_0)\}$$

is always true ($swap(A, i, j)$ represents the code $z := A[i]; A[i] := A[j]; A[j] := z$). Thus, for simplicity, we drop $perm(A, A_0)$ from our discussion and focus only on $sorted(A, 0, n)$.

Intuitively, $sorted(A, i, j)$ is true iff $A[i] \leq A[i+1] \leq \cdots \leq A[j-1]$. For verification purpose, the program library does not need an implementation of $sorted(A, i, j)$. On the other hand, the programmer and theorem prover needs to know the meaning of $sorted(A, i, j)$, which can be defined recursively as follows:

$$\begin{aligned} sorted(A, i, j) &= true \textbf{ if } i \geq j; \\ sorted(A, i, j+1) &= sorted(A, i, j) \wedge A[j-1] \leq A[j] \textbf{ if } i < j+1. \end{aligned}$$

Example 10.3.1. If we wish to generate an invariant for the insertion sort algorithm, the first technique to apply is constant relaxation: Replace $sorted(A, 0, n)$ by $k = n \wedge sorted(A, 0, k)$. Dropping $k = n$ as the exit condition, the outer loop looks as follows:

```
1   k := 1;
2   while k ≠ n do
3       {I : g > 0 ∧ sorted(A, 0, k)}
        ...
```

```
10      k := k + 1
11  od
```

It is easy to see that the exit condition of the outer loop, i.e., $k = n$, will give us the postcondition $sorted(A, 0, n)$.

Now apply uncoupling and aging to $sorted(A, 0, k)$ by introducing a new variable j, which breaks the first $k+1$ elements of A into two sorted lists: $A[0], ..., A[j-1]$, and $A[j], ..., A[k]$. We obtain invariant I' for the inter loop:

$$I' : 0 \leq j \leq k \wedge sorted(A, 0, j) \wedge sorted(A, j, k + 1)$$

which has three conjuncts. The first conjunct is the bounding invariant. When the exit condition of the inner loop, i.e., $\neg(j > 0 \wedge A[j-1] > A[j])$ becomes true, we have $j = 0 \vee A[j-1] \leq A[j])$. Together with the last two conjuncts of I', we can show that $sorted(A, 0, k + 1)$ is true.

```
1   k := 1;
2   while k ≠ n do
3       {I : k > 0 ∧ sorted(A, 0, k)}
4       j := k;
5       while (j > 0 ∧ A[j − 1] > A[j]) do
6           {I' : 0 ≤ j ≤ k ∧ sorted(A, 0, j) ∧ sorted(A, j, k + 1)}
7           swap(A, j, j − 1);
8           j := j − 1;
9       od
10      k := k + 1;
11  od
```

It is easy to see that $sorted(A, 0, j)$ and $sorted(A, j, k+1)$ are true when $j = k$, because $sorted(A, 0, j)$ comes from I and $sorted(A, j, k + 1)$ from the definition of *sorted*. To show that $0 \leq j \leq k$ is true when $j = k$, we need to know $k \geq 0$. To have this, we may add $k > 0$ as a bounding invariant into I of the outer loop. Once this is done, I' can be shown to be true before entering the inner loop.

To see that I' is an invariant for the inner loop, when the loop condition is true, $0 \leq j-1$ is true because $j > 0$. $sorted(A, 0, j-1)$ is true because of $sorted(A, 0, j)$, $sorted(A, j-1, k+1)$ is true because $sorted(A, j, k+1)$ and $A[j-1] < A[j]$ (after swapping). Thus, I' is indeed a loop invariant.

The exit condition of the inner loop is $j \leq 0 \vee A[j-1] \leq A[j]$. If $j \leq 0$ is true, from $0 \leq j$, we know $j = 0$, hence $sorted(A, 0, k+1)$ is true from $sorted(A, j, k+1)$. If $A[j-1] \leq A[j]$ is true, from $sorted(A, 0, j)$, $A[j-1] < A[j]$, and $sorted(A, j, k+1)$,

we conclude also that $sorted(A, 0, k+1)$ is true. After the execution of line 10, $k := k+1$, we arrive at the invariant of the outer loop, i.e., $k > 0 \wedge sorted(A, k)$. □

The insertion algorithm in the above example can be regarded as an example of "program synthesis", if the invariants can be generated automatically.

Example 10.3.2. If we are asked to write a program that "store in y the maximum of x and y". The postcondition is naturally $y = max(x, y)$. Assume the precondition is $\top$, what will be the code C in $\{\top\}\ C\ \{y = max(x, y)\}$. In fact, C is not unique. For example, some choices are:

- **if** $x > y$ **then** $y := x$ **fi**;
- **if** $x > y$ **then** $x := y$ **fi**;
- $y := x$;
- $x := y$;

What will be the "correct" code? With respect to the postcondition, the above four codes are all correct. If the intended meaning is that y contains the maximum of the original values of x and y, in this case, we need auxiliary variables, such as x_0 and y_0, to denote the original values of x and y. The correct postcondition is $y = max(x_0, y_0)$ and the first of the above codes is correct with respect to this postcondition. If we ask further that $y = max(x, y)$, x and y hold the original values of x and y, then the correct postcondition is $y = max(x_0, y_0) \wedge (x = x_0 \wedge y = y_0 \vee x = y_0 \wedge y = x_0)$, and none of the above codes is correct. □

The above simple example illustrates that there exist many codes to achieve the same postcondition; postconditions should be carefully chosen to reflect precisely the requirement. When loops invariants are needed, the complexity of program synthesis grows exponentially. Since there are too many choices to create proper loop invariants, thus leading to various algorithms, program synthesis remains a very challenging task. Here is a quotation from Wikipedia on an annual competition on program synthesis.

> The early 21st century has seen a surge of practical interest in the idea of program synthesis in the formal verification community and related fields. In 2013, a unified framework for program synthesis problems was proposed by researchers at UPenn, UC Berkeley, and MIT. Since 2014 there has been a yearly program synthesis competition comparing

the different algorithms for program synthesis in a competitive event, the Syntax-Guided Synthesis Competition or SyGuS-Comp. Still, the available algorithms are only able to synthesize small programs.

10.3.3 Choosing A Good Language for Assertions

Assertions are written in a first-order language. However, there are many ways to specify assertions, because first-order languages are so expressive and you can use various predicates and functions to specify the same property. There are two possibilities:

- Allow assertions, in particular postconditions and loop invariants, to use functions and predicates defined using some appropriate mechanism (often, the programming language's function declaration construct) to express high-level properties based on a domain theory covering specifics of the application area. We call this approach *domain theory*.
- Disallow the previous possibility, requiring assertions always to be expressed in terms of the constructs of the assertion language, without functions. We call this approach *atomic assertions*.

10.4 Exercise Problems

1. Prove formally the following triple is true by choosing a proper loop invariant.

$$\{x = 0\}\ \textbf{while}\ (x < 100)\ \textbf{do}\ x := x + 1\ \textbf{od}\ \{x = 100\}$$

2. Given the following annotated program, provide the loop invariant and prove its total correctness by Hoare logic. For every simplification step used in the proof, please state clearly which arithmetic/algebraic/logic law is used for simplification.

$\{A : b \geq 0\}$
$int\ x = a, y = b, z = 1;$
while $y \neq 0$ **do**
    if $(y\%0 = 1)$
        $y := y - 1; z := x * z$
    else
        $x := x * x; y := y/2$
$\{B : z = a^b\}$

CHAPTER 11
TEMPORAL LOGIC

In the previous chapter, we have shown that Hoare logic can be used to verify a program can achieve the desire property after its execution. However, when a program involves multiple concurrent processes, its verification becomes much more complex and the first order logic is inadequate. For example, when you deposit cash into an automated teller machine (ATM), the program running inside the ATM will read the balance of your bank account and add the cash amount into your account. At the same time, another program running inside the bank may use the same account to pay off your credit card. The two programs cannot be run at the same time, otherwise, the balance of your account may go wrong. Modifying the same data at the same time by multiple processes is called "race condition" in concurrent programming. To avoid race condition, a locking mechanism is needed so that an account can be accessed when it is free (unlocked). Due to locking mechanisms, two programs may go to a "deadlock" state. For instance, when both you and your friend send money to each other at the same time, one program locked account A and asks for account B and the other program locked account B and asks for account A. In such cases, we cannot prove the termination of the program by counting steps. To verify a program works correctly without race condition or deadlock, we need a formalism to express time, such as "two processes cannot access a critical data at the same time". or "the program will terminate eventually".

In logic, the term *temporal logic* is any system of rules and symbolism for representing and reasoning about time and temporal information, as well as their formal representation. Temporal logic is sometimes also used to refer to tense logic, a modal logic-based system introduced by Arthur Prior in the late 1950s and has been further developed by computer scientists and logicians. Applications of temporal logic include its use as a formalism for clarifying philosophical issues about time, as a framework within which to define the semantics of temporal expressions in natural language, as a language for encoding temporal knowledge in artificial intelligence.

Temporal logic has found as an important tool in formal verification, where it is used to specify requirements of computer programs and systems and conduct formal analysis and verification of the executions of computer programs. For instance, one may wish to say that whenever a request is made, access to a resource is eventually

granted, but it is never granted to two requestors simultaneously. Such a statement can conveniently be expressed in a temporal logic.

Consider the statement "I possess account A". Though its meaning is constant in time, the statement's truth value can vary in time. Sometimes it is true, and sometimes false, but never simultaneously true and false. In temporal logic, a statement can have a truth value that varies in time. This treatment of truth value over time differentiates temporal logic from classical first-order logic.

11.1 An Approach from Modal Logic

A modal is an expression (like "necessarily" or "possibly") that is used to qualify the truth of a statement. Modal logic is the study of the deductive behavior of the expressions containing the phrases "it is necessary that" and "it is possible that". Modal logic has been extended to include logics for belief, for tense and other temporal expressions where "necessarily" (moral) is interpreted as "always" (temporal) and "possibly" as "eventually". An understanding of modal logic and its extensions is particularly valuable in the formal analysis of philosophical argument, where expressions from the modal family are both common and confusing.

Like most formal logical system, modal logic as well as temporal logic can be defined as an extension of propositional logic. Here, we present a minimal temporal logic (MTL) by adding two modal operators, $\Box$ and $\Diamond$, to propositional logic.

11.1.1 Modal Operators

The first operator $\Box$ denotes "always" and takes a formula as its argument. For example, if p denotes "it rains today", then $\Box p$ is a formula of MTL and its intuitive meaning is "it will rain everyday".

The second operation $\Diamond$ denotes "eventually" and also takes a formula as its argument. Thus, if p denotes "it rains today", $\Diamond p$ is a formula of MTL and its intuitive meaning is "it will rain someday".

Informally, $\Box$ states a universal property in the future while $\Diamond$ states an existential property in the future. For example, to specify the statement "if a request is made to print a file, the file will be printed eventually", the statement can be denoted by $\Box(r \rightarrow \Diamond p)$, where r stands for "request is made" and p for "the file is printed".

Using the BNF grammar, the formulas of MTL can be formally defined as follows:

Definition 11.1.1. *Let op be the binary logical operators of propositional logic and V_P be the propositional variables used in the current application, then the* formulas

of MTL can be defined by the following BNF grammar:

$$\begin{array}{rcl} \langle op \rangle & ::= & \wedge \mid \vee \mid \rightarrow \mid \oplus \mid \leftrightarrow \\ \langle V_P \rangle & ::= & p \mid q \mid r \mid s \mid t \\ \langle Formulas \rangle & ::= & \top \mid \bot \mid \langle V_P \rangle \mid \neg \langle Formulas \rangle \mid (\langle Formulas \rangle \ \langle op \rangle \ \langle Formulas \rangle) \mid \\ & & \Box \langle Formulas \rangle \mid \Diamond \langle Formulas \rangle \end{array}$$

Here are some examples of MTL formulas:

$$(p \wedge \Box \neg q), \quad \Diamond (r \rightarrow (\Diamond p \wedge \neg q)), \quad \Box((p \wedge \neg q) \vee \Diamond r) \quad (\Box(p \wedge \neg q) \vee \Diamond r)$$

Note the difference of the last two formulas, where $\Box$ applies to the whole formula in one and only to the first argument of $\vee$ in the other.

11.1.2 Kripke Semantics

The two modal operators of MTL are directly borrowed from classical modal logic, where $\Box$ is usually read as "necessarily" and $\Diamond$ as "possibly". The first formal semantics of modal logic is called *Kripke frames*, after Saul Kripke created it in early 1960s. Kripke semantics (also known as relational semantics or frame semantics) is a formal semantics for modal logic systems created in the late 1950s by Saul Kripke and Andre Joyal. It was adapted to temporal logic and other non-classical systems. The development of Kripke semantics was a breakthrough in the theory of non-classical logics, because the model theory of such logics was almost non-existent before Kripke.

As we have known from Chapter 2, an interpretation of a propositional formula is a mapping of propositional variables to truth values. Given n propositional variables, we have exactly 2^n different interpretations. For example, if we have two propositional variables, say p and q, then there are four interpretations, $I_1 = TT$ (it means $I_1(p) = \top$, $I_1(q) = \top$), $I_2 = TF$, $I_3 = FT$, and $I_4 = FF$. Using the notation from Chapter 2, $V_P = \{p, q\}$ and $IV_P = \{TT, TF, FT, FF\}$.

Definition 11.1.2. *Given a set of propositional variables V_P, a* Kripke frame *is a directed graph $K = (W, R)$, where W, called* states*, is a non-empty subset of IV_P, the set of all interpretations over V_P, and R is a binary relation over W.*

Example 11.1.3. Let $V_P = \{p, q\}$, $W = IV_P = \{TT, TF, FT, FF\}$, and $R = \{\langle TT, FT\rangle, \langle TF, TT\rangle, \langle TF, TF\rangle, \langle FT, FT\rangle, \langle FT, FF\rangle, \langle FF, TF\rangle\}$. Then $G = (W, R)$, displayed in Figure 11.1.1, is a Kripke frame. □

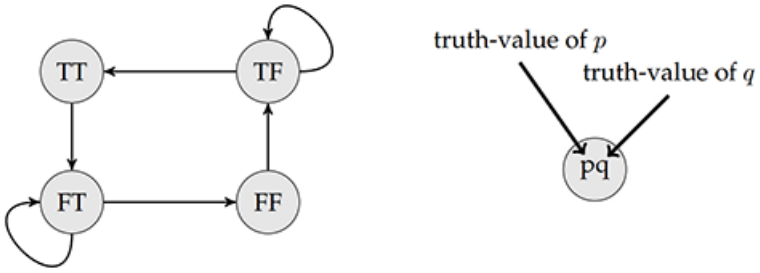


Figure 11.1.1: A graphical display of the Kripke frame

It should be understood that in the discussion, every MTL formula uses the propositional variables from V_P and every state w in a Kripke frame $K = (W, R)$ is an interpretation of V_P. If $\langle w, w' \rangle \in R$, we say w' is a *successor* of w in K.

Definition 11.1.4. *Given a Kripke frame $K = (W, R)$ and a state $w \in W$, an MTL formula C is* true *in w of K recursively, denoted by $K, w \models A$, if one of the following conditions is true:*

1. *if C is $\top$;*
2. *if $C \in V_P$ and $w(C) = \top$;*
3. *if C is $\neg A$ and it is not the case that $K, w \models A$;*
4. *if C is $A \wedge B$, $K, w \models A$ and $K, w \models B$;*
5. *if C is $A \vee B$ and either $K, w \models A$ or $K, w \models B$;*
6. *if C is $A \to B$ and $K, w \models \neg A \vee B$;*
7. *if C is $\Box A$ and for every successor w' of w, $K, w' \models A$;*
8. *if C is $\Diamond A$ and for some successor w' of w, $K, w' \models A$.*

Otherwise, we say C is false in w.

Example 11.1.5. Consider the Kripke frame in Figure 11.1.1 and the formula $A = p \wedge q \to \Box\neg p$. A is true in $w = FT$ or FF, because p is false in w. A is true in $w = TF$ because q is false in w. A is also true in $w = TT$ because $\Box\neg p$ is true in $w' = FT$. For the sams analysis, we can show that $B = p \wedge q \to \Box\Box\neg p$ is true in every state. However, $C = p \wedge q \to \Box\Box\Box\neg p$ is false in $w = TT$. $\Box$

In modal logic, a Kripke frame specifies the transitions of states (also called *worlds*). *Necessarily* means "in all successive states (successors)", and *possibly* means "in one of the successors". In the same way, in MTL, a formula $\Box A$ is true in the current state if A is *always* true in every successor; $\Diamond A$ is true in the current state if A is *eventually* true in one of the successors. Note that if the current state does not have any successor, then $\Box A$ is true but $\Diamond A$ is false for any formula A in the current state.

Definition 11.1.6. *Given a Kripke frame $K = (W, R)$, an MTL formula A is* true *in K, written $K \models A$, if for every state $w \in W$, $K, w \models A$, and we say K is a* Kripke model *of A.*
A is satisfiable *in MTL if A has a Kripke model. That is, there exists a Kripke frame K such that $K \models A$.*
A is valid *in MTL, written $\models A$, if A is true in every Kripke frame. That is, every Kripke frame is a model of A.*

Example 11.1.5 shows that both $p \wedge q \to \Box\neg p$ and $p \wedge q \to \Box\Box\neg p$ are satisfiable as they are true in the Kripke frame in Figure 11.1.1. However, this frame is not a model of $p \wedge q \to \Box\Box\Box\neg p$.

Given a MTL formula A, let $\mathcal{M}(A)$ denote the set of all Kripke models of A.

Definition 11.1.7. *Given two MTL formulas A and B, A and B are* equivalent*, written $A \equiv B$, if $\mathcal{M}(A) = \mathcal{M}(B)$.*
We say A entails *B, or B is a* logical consequence *of B, written $A \models B$, if $\mathcal{M}(A) \subseteq \mathcal{M}(B)$.*

Given a set V_P of n propositional variables, the number of choices for W in a Kripke frame is enormous ($2^{2^n} - 1$) and the number of choices for R is also huge ($2^{|W|^2}$ for each W). Hence, it is infeasible to check if a MTL formula is valid by enumerating all frames. However, all tautologies in propositional logic are valid in MTL and all major results from propositional logic are still useful here.

Theorem 11.1.8. (a) **Substitution** *Any instance of a valid MTL formula is valid in MTL. That is, if p is a propositional variable in A and B is any formula, then $A[p \leftharpoonup B]$ is valid whenever A is valid.*

(b) **Substitution of equivalence** *For any formulas A, B, and C, where B is a subformula of A, and $B \equiv C$, then $A \equiv A[B \leftharpoonup C]$.*

(c) **Equivalence** *Given two MTL formulas A and B, $\models A \leftrightarrow B$ iff $A \equiv B$.*

(d) **Entailment** *Given two MTL formulas A and B, $\models A \to B$ iff $A \models B$.*

The proofs of the above theorem are similar to those for propositional logic and omitted here.

The proofs of the following theorems are examples of proving a formula is valid or equivalent to another formula. Later, we will provide some proof techniques based on semantic tableau.

Theorem 11.1.9. (Duality) $\neg\Box p \equiv \Diamond\neg p$.

Proof. For any Kripke frame $K = (W, R)$, if K is a model of $\neg\Box p$, then for every state $w \in W$, $\neg\Box p$ is true in w. Equivalently, $\Box p$ is false in w. That means there exists a successor w' of w such that p is false in w', or $\neg p$ is true in w'. Hence, $\Diamond\neg p$ is true in w. So K is a model of $\Diamond\neg p$.

On the other hand, if K is not a model of $\neg\Box p$, then there exists a state $w \in W$ such that $\neg\Box p$ is false and $\Box p$ is true in w. By definition, for every successor w' of w, p must be true in w'. Hence $\neg p$ is false in every successor of w and $\Diamond\neg p$ is false in w. So K cannot be a model of $\Diamond\neg p$. □

Applying substitution and equivalence, we can easily deduce from the above theorem that $\Box\neg p \equiv \neg \Diamond p$ and $\Box\neg p \leftrightarrow \neg \Diamond p$ is valid.

Example 11.1.10. $(\Box p \rightarrow p) \equiv (\neg p \rightarrow \Diamond\neg p)$. □

The two formulas can be transformed by logical equivalence to the same negation normal form, i.e., $p \vee \Diamond\neg p$. We will discuss negation normal form in the next section.

Theorem 11.1.11. Distribution $\models \Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$.

Proof. $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$ is logically equivalent to

$$(1) \qquad \neg\Box(\neg p \vee q) \vee \neg\Box p \vee \Box q.$$

Applying the duality theorem to (1), we obtain another equivalent formula

$$(2) \qquad \Diamond (p \wedge \neg q) \vee \Diamond\neg p \vee \Box q.$$

For any Kripke frame $K = (W, R)$ and any $w \in W$, if (2) is false in w, that is, $\Diamond(p \wedge \neg q)$, $\Diamond\neg p$, and $\Box q$ are all false in w, then w must have some successors (otherwise, $\Box q$ is true in w) and for any successor w' of w, q must be true in w' because of $\Box q$. p must be true in w' because of $\Diamond\neg p$. Hence, $p \wedge \neg q$ is true in w' and $\Diamond(p \wedge \neg q)$ is true in w, a contradiction. Thus, (2) must be true in w and thus true in every Kripke frame. □

11.1.3 Restrictions and Limitations

Let r denotes "it rains", then the intuitive meaning of $\Box r$ is "it always rains". Given a Kripke frame $K = (W, R)$, if $\Box r$ is true in a state $w \in W$, we know r will be true in any successor of w. What is the truth value of r in the state w? If $\Box$ means "always", should r be true now and then in every state, including w? That is, we ask that $\Box A \to A$ be an axiom of MTL.

We say a Kripke frame $K = (W, R)$ is *reflexive* if every state $w \in W$ is a successor of itself, that is, $\langle w, w\rangle \in R$ (called *loop* in graph theory).

Theorem 11.1.12. (Reflexivity) *A Kripke frame K is reflexive iff K is a model of $\Box A \to A$ for any formula A.*

Proof. Let $K = (W, R)$ be reflexive. If $\Box A \to A$ is false in K, then there exists $w \in W$ such that $\Box A$ is true but A is false in w. Since $\Box A$ is true, A is true in every successor of w. However, w is also a successor of w because K is reflexive. So A is true in w, too. This is a contradiction to $\Box A \to A$ being false in K.

On the other hand, if $K = (W, R)$ is not reflexive. Pick a state $w \in W$ such that w is not a successor of itself. Let I_w be the propositional interpretation associated with w. If we view I_w as a conjunction of literals each of which is true in w, then $\neg(I_w)$ is a clause which is false in w. This clause will be true in every interpretation (including all successors of w) other than w. Thus, $\Box\neg(I_w)$ is true in w. Let A be $\neg(I_w)$, then $\Box A \to A$ is false in w. □

The above theorem plays a dual role: By restricting Kripke frames to reflexive ones, we obtain the validity of $\Box A \to A$ in these frames; on the other hand, if $\Box A \to A$ is given as an axiom (a formula assuming to be true), we exclude all non-reflexive Kripke frames as model candidates.

We know from Kripke frames that if $\Box p$ is true in the current state w, then p will be true in all the successors of w. What about the truth value of p in the successors of the successors of w? If $\Box$ means "always", should p be true in them, too? Essentially, we ask $\Box A \to \Box\Box A$ to be true in every state for every formula A. Let us say a Kripke frame $K = (W, R)$ be *transitive* if R is transitive: for any states u, v, w, $\langle u, v\rangle \in R$ and $\langle v, w\rangle \in R$ imply that $\langle u, w\rangle \in R$. The following theorem tells us the relationship between the transitivity and the formula $\Box A \to \Box\Box A$.

Theorem 11.1.13. (Transitivity) *A Kripke frame K is transitive iff K is a model of $\Box A \to \Box\Box A$ for any formula A.*

Proof. Let $K = (W, R)$ be reflexive. If $\Box A \to \Box\Box A$ is false in K, then there exists $w \in W$ such that $\Box A$ is true but $\Box\Box A$ is false in w. The latter implies that there must exist a successor w' of w and a successor w'' of w' such that A is false in w''.

Since $\Box A$ is true, A is true in every successor of w. However, w'' is also a successor of w because K is transitive. So A is true in w'', too. This is a contradiction to $\Box A \rightarrow \Box\Box A$ being false in K.

On the other hand, if $K = (W, R)$ is not reflexive. Pick three states $w_1, w_2, w_3 \in W$ such that w_2 is a successor of w_1, $w3$ is a successor of w_2 but not a successor of w_1. Let I_3 be the propositional interpretation associated with w_3. If we view I_3 as a conjunction of literals each of which is true in w_3, then $\neg(I_3)$ is a clause which is only false in w_3 but true in every interpretation (including all successors of w_1) other than w_3. Let A be $\neg(I_3)$, then A is false in w_3, $\Box A$ is false in w_2 and $\Box\Box A$ is false in w_1. Since $\Box A$ is true in w_1, $\Box A \rightarrow \Box\Box A$ is false in w_1. □

Combing the above two theorems, we can conclude that if a Kripke frame is both reflexive and transitive, then $\Box A \leftrightarrow \Box\Box A$ will be true in this frame.

The *necessitation rule* of modal logic states that if A is a theorem, then so is $\Box A$. That is, if A can be proved to be true now, the same proof can be used everywhere. The axiom for expressing the necessitation rule is the formula $A \rightarrow \Box A$. The Kripke models of $A \rightarrow \Box A$ are those Kripke frames $K = (W, R)$ in which the only successor of each state is itself, i.e., if $\langle w, v\rangle \in R$, then $w = v$. Such frames are called *discrete*. Obviously, the only edges allowed in a discrete frame are loops (i.e., $\langle w, w\rangle$) and such frames are not very useful. Every reflexive frame is discrete, but the reverse may not be true, as some loop may be missing in a discrete frame.

Besides reflexivity, transitivity, and discreteness, there are many other restrictions on the relations in a Kripke frame, such as transitivity, symmetry, dense, etc. There are axioms corresponding to these restrictions.

Semantics is useful for a logic only if the semantic consequence reflects its syntactical counterpart, i.e., the result of the proof system for the logic. It is vital to know which modal logics are sound and complete with respect to a set of Kripke frames. Let C be the set of Kripke frames satisfying some properties. We define $Thm(C)$ to be the set of all formulas that are true in C. A proof system in a modal logic is *sound* with respect to C if every formula proved to be true belongs to $Thm(C)$; it is *complete* with respect to C if every formula in $Thm(C)$ can be proved to be true. There are over a dozen of proof systems for modal logic and the interested reader may refer to a textbook on modal logic.

Temporal logic always has the ability to reason about a timeline, which is linear by nature. The Kripke semantics use directed graphs $K = (W, R)$ and are suitable for reasoning about multiple timelines, as each state allows multiple successors. We may put restrictions over R so that R acts like a linear relation: Each state has at most one successor. This restriction can work with reflexivity by allowing each state having at most one successor other than itself. However, this restriction does not

work well with transitivity because transitivity wants a state to include successors of successors as its own successors. Moreover, the number of states in a Kripke frame is bound by the number of propositional interpretations. We cannot model a timeline of infinite length (other than a cycle in the graph). Hence, we need semantics different from Kripke frames for linear temporal logic; temporal logic based on Kripke frames can be viewed as a branching logic.

11.2 Linear Temporal Logic

Linear temporal logic (LTL) or linear-time temporal logic is a modal temporal logic with modalities referring to time. LTL was first proposed for the formal verification of computer programs by Amir Pnueli in 1977. In LTL, one can encode formulas about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc. For simplicity, we will define LTL as an extension of propositional logic. In terms of expressive power, LTL is a fragment of first-order logic.

In LTL, $\Box$ is interpreted as "always" and $\diamond$ as "eventually". In addition to these two common modal operators, we have the third operator $\circ$ which is interpreted as "next" tick (or step). Like $\Box$ and $\diamond$, $\circ$ takes a formula as argument: If r means "it rains today", then $\circ r$ means "it will main tomorrow", if we interpret that a tick is a day. The operator $\circ$ is an important part of LTL, which has been invented for the formal verification of concurrent programs. However, $\circ$ is rarely used in the specification of concurrent programs, because not much are said about the execution of programs in the *next* step. Furthermore, we want a correctness statement to hold regardless of how the interleaving selects a *next* operation. Therefore, properties are almost invariably expressed in terms of $\Box$ and $\diamond$, not in terms of $\circ$.

The formal definition of the LTL is given as the following BNF grammar:

$$
\begin{array}{rcl}
\langle op \rangle & ::= & \wedge \mid \vee \mid \rightarrow \mid \oplus \mid \leftrightarrow \\
\langle V_P \rangle & ::= & p \mid q \mid r \mid s \mid t \\
\langle Formulas \rangle & ::= & \top \mid \bot \mid \langle V_P \rangle \mid \neg \langle Formulas \rangle \mid (\langle Formulas \rangle \ \langle op \rangle \ \langle Formulas \rangle) \mid \\
 & & \Box \langle Formulas \rangle \mid \diamond \langle Formulas \rangle \mid \circ \langle Formulas \rangle
\end{array}
$$

Here are some examples of LTL formulas:

$$(\circ p \wedge \Box \neg q), \quad \diamond (\circ r \rightarrow (\diamond p \wedge \neg q)), \quad \Box((p \wedge \neg q) \vee \circ \diamond r)$$

11.2.1 Timeline as Interpretation Sequence

The semantics of LTL will be defined by an infinite sequence of propositional interpretations so that a propositional variable has a fixed truth value at any position (called state) in the sequence. Formally, an interpretation sequence σ is a mapping from the set of natural numbers to the set of all propositional interpretations IV_P:

$$\sigma : \{0, 1, 2, ...\} \longrightarrow IV_P$$

For any $i \geq 0$, let $\sigma(i) = s_i$. Then σ can be written as a sequence of interpretations:

$$\sigma = s_0 s_1 s_2 \cdots$$

where $s_i = \sigma(i) \in IV_P$ for any $i \geq 0$ and i is called a *state* of σ and s_i is the interpretation at state i. For $i \geq 0$, we use $\sigma_{\geq i}$ to denote the *suffix* of σ by removing the first i interpretations from σ:

$$\sigma_{\geq i} = t_0 t_1 t_2 \cdots$$

where $t_0 = s_i, t_1 = s_{i+1}, t_2 = s_{i+2}, ...$ Obviously, $\sigma_{\geq 0} = \sigma$, $\sigma(0) = s_0$, and $\sigma = \sigma(0)\sigma_{\geq 1}$. We say $\sigma_{\geq 1}$ is the *immediate suffix* of σ.

Definition 11.2.1. *Given an interpretation sequence σ, an LTL formula C is said to be* true *in σ recursively, denoted by $\sigma \models A$, if one of the following conditions is true:*

1. *if C is $\top$;*
2. *if C is $\neg A$ and it is not the case that $\sigma \models A$;*
3. *if C is $A \wedge B$, $\sigma \models A$ and $\sigma \models B$;*
4. *if C is $A \vee B$ and either $\sigma \models A$ or $\sigma \models B$;*
5. *if C is $A \rightarrow B$ and $\sigma \models \neg A \vee B$;*
6. *if C is $\circ A$ and $\sigma_{\geq 1} \models A$;*
7. *if C is $\Box A$ and for all $i \geq 0$, $\sigma_{\geq i} \models A$;*
8. *if C is $\Diamond A$ and there exists $i \geq 0$, $\sigma_{\geq i} \models A$.*

Otherwise, we say C is false *in σ.*

From the above recursive definition, we can see that a formula $\circ A$ is true in an interpretation sequence σ iff A is true in $\sigma_{\geq 1}$, the sequence obtained by removing the first item from σ. A formula $\Box A$ is true in σ iff A is *always* true in every suffix of σ; $\Diamond A$ is true in σ iff A is *eventually* true in one suffix of σ. In contrast, in the Kripke semantics, $\Box A$ is true at the current state iff A is true in every *successor* state. Note that the above definition of the truthfulness can be extended to sequences of finite length. If σ is an empty sequence, for any formula LTL A, $\Box A$ is true in σ but $\circ A$ and $\Diamond A$ are false in σ.

For $\Diamond\Box p$ to be true in σ, we need to find a state in σ such that p is true in every interpretation of σ from that state. That means, only a prefix of σ may contain a finite number of interpretations which falsify p; p must be true in every interpretation not in the prefix. That is, there exists $i \geq 0$, for all $j \geq i$, $\sigma(j)(p) = \top$. For $\Box \Diamond p$ to be true in σ, we just need p to be true in one of the interpretations of any suffix of σ: For all $i \geq 0$, there exists $j \geq i$, $\sigma(j)(p) = \top$. Thus, in first-order language, σ for $\Diamond\Box p$ satisfies $\exists i \forall j R(i, j)$, while σ for $\Box\Diamond p$ satisfies $\forall i \exists j R(i, j)$, where $R(i, j)$ denotes the formula $i \geq 0 \wedge j \geq i \wedge \sigma(j)(p) = \top$. From the view of the first-order formulas, $\Box \Diamond p$ is not equivalent to $\Diamond\Box p$.

Example 11.2.2. For the formula $\Box \Diamond p \wedge \Box \Diamond \neg p$, we have a sequence σ where the truth value of p is alternating in σ. $\Box\Diamond p$ is true in σ because p is true in every suffix of σ. $\Box\Diamond p$ is also true in σ because $\neg p$ is true in every suffix of σ. Thus $\Box\Diamond p \wedge \Box\Diamond \neg p$ is true in σ. Note that $\Diamond\Box p$ is not true in σ, another evidence $\Box \Diamond p$ and $\Diamond\Box p$ are not equivalent. □

Example 11.2.3. $\Diamond\Box p \wedge \Diamond\Box\neg p$ is not true in any sequence σ, because $\Diamond\Box p$ requires p be true everywhere in one suffix of σ. $\Diamond\Box\neg p$ requires p be false everywhere in one suffix of σ. The two conditions cannot be met at the same time. □

Definition 11.2.4. *A LTL formula A is* satisfiable *if there exists an interpretation sequence σ such that $\sigma \models A$. We say σ is a* model *of A.*
A is valid *in LTL, written $\models A$, if every sequence of interpretations is a model of A.*

Both $\Diamond\Box p$ and $\Box \Diamond p$ are satisfiable but not valid. From Example 11.2.3, $\Box \Diamond p \wedge \Box \Diamond \neg p$ is satisfiable but not valid. $\Diamond\Box p \wedge \Diamond\Box\neg p$ is unsatisfiable and its negation is valid.

Theorem 11.2.5. *For any LTL formula A,*

$$(a)\ \textbf{(reflexivity)} \quad \models \Box A \rightarrow A;$$
$$(b)\ \textbf{(transitivity)} \quad \models \Box A \rightarrow \Box\Box A.$$

Proof. (a) To show $\Box A \to A$ is valid in LTL, consider any interpretation sequence σ for A. If $\Box A \to A$ is false in σ, then $\Box A$ is true and A is false in σ. $\Box A$ means A is true at every state of σ, including state 0, i.e., A is true in σ. That is a contradiction. Hence $\Box A \to A$ is true in σ.

The proof of (b) is similar to that of (a). □

Note that neither $\Box A \to A$ nor $\Box A \to \Box\Box A$ is valid in MTL. Theorem 11.1.12 states that $\Box A \to A$ is true in every reflexive Kripke model and Theorem 11.1.13 states that $\Box A \to \Box\Box A$ is true in every transitive Kripke model. The semantics of interpretation sequences is not a special case of Kripke semantics as Kripke frames have no ways to specify the unique path explicitly defined by an interpretation sequence. On the other hand, interpretation sequences are not expressive enough to specify various relations among propositional interpretations.

Given an LTL formula A, let $\mathcal{M}(A)$ denote the set of all models of A.

Definition 11.2.6. *Given two LTL formulas A and B, A and B are* equivalent*, written $A \equiv B$, if $\mathcal{M}(A) = \mathcal{M}(B)$.*
We say A entails *B, or B is a* logical consequence *of B, written $A \models B$, if $\mathcal{M}(A) \subseteq \mathcal{M}(B)$.*

Theorem 11.2.7. *For any formula LTL formula A, (a) $\Box A \models \circ A$; (b) $\circ A \models \diamond A$; and (c) $\Box A \models \diamond A$.*

The proofs of this theorem are straightforward, similar to the proof of (a) of Theorem 11.2.5.

Like MTL, all tautologies in propositional logic are valid in LTL and all major results from propositional logic are still useful here.

Theorem 11.2.8. (a) **Substitution** *Any instance of a valid LTL formula is valid in LTL. That is, if p is a propositional variable in A and B is any formula, then $A[p \leftarrow B]$ is valid whenever A is valid.*

(b) **Substitution of equivalence** *For any formulas A, B, and C, where B is a subformula of A, and $B \equiv C$, then $A \equiv A[B \leftarrow C]$.*

(c) **Equivalence** *Given two LTL formulas A and B, $\models A \leftrightarrow B$ iff $A \equiv B$.*

(d) **Entailment** *Given two LTL formulas A and B, $\models A \to B$ iff $A \models B$.*

The proofs of the above theorem are similar to those for propositional logic and omitted here.

From Theorem 11.2.5 (a) and (b), it is easy to deduce that $\Box A \to \Box\Box A \equiv \Box A \to \Box\Box A$.

11.2.2 Properties of LTL

The proofs of the following theorems are examples of proving a formula is valid or equivalent to another formula. In the next section, we will provide some proof techniques based on semantic tableau. The operator $\circ$ commutes with $\Box$, $\Diamond$, and $\neg$, but $\Box$ and $\Diamond$ do not commute with each other, as shown in Example 11.2.2.

Theorem 11.2.9. (Commutativity)

$$\begin{array}{lrcl} (a) & \Box \circ p & \equiv & \circ \Box p; \\ (b) & \Diamond \circ p & \equiv & \circ \Diamond p; \\ (c) & \neg \circ p & \equiv & \circ \neg p. \end{array}$$

Proof. The proofs of (a) and (b) are similar to that of (c): Given any interpretation sequence σ, σ is a model of $\neg \circ p$ iff $\circ p$ is false in σ; $\circ p$ is false in σ iff p is false in $\sigma_{\geq 1}$, or equivalently, $\neg p$ is true in $\sigma_{\geq 1}$; $\neg p$ is true in $\sigma_{\geq 1}$ iff $\circ \neg p$ is true in σ. $\Box$

Theorem 11.2.10. (Duality)

$$\begin{array}{lrcl} (a) & \neg \Box p & \equiv & \Diamond \neg p; \\ (b) & \neg \Diamond p & \equiv & \Box \neg p. \end{array}$$

Proof. (a) Given any interpretation sequence σ, $\neg \Box p$ is true in σ iff $\Box p$ is false in σ; $\Box p$ is false in σ iff there exists $i \geq 0$, $\sigma(i)(p) = \bot$, or equivalent, $\sigma(i)(\neg p) = \top$. There exists $i \geq 0$, $\sigma(i)(\neg p) = \top$ iff $\Diamond \neg p$ is true in σ.

(b) Replace p by $\neg q$ in (a), and apply $\neg \neg q \equiv q$, we have $\neg \Diamond q \equiv \Box \neg q$. $\Box$

In Chapter 2, we introduced a concept called "negation normal form", which is a propositional formula where every argument of $\neg$ is a propositional variable. This concept applies to formulas of LTL as well as MTL. Applying the two theorems above, we have the following theorem.

Theorem 11.2.11. *For every LTL formula A, there exists an LTL formula B such that $A \equiv B$ and B is in negation normal form.*

Example 11.2.12. Let us transform $\neg(\circ \vee p \rightarrow \Box q)$ into negation normal form:

$$\begin{array}{rcl} \neg(\circ \Diamond p \vee \Box q & \equiv & \neg \circ \Diamond p \wedge \neg \Box q \\ & \equiv & \circ \neg \Diamond p \wedge \neg \Box q \\ & \equiv & \circ \Box \neg p \wedge \Diamond \neg q \end{array}$$

$\Box$

Example 11.2.13. To show that $\diamond\Box p \rightarrow \Box\diamond p$ is valid, we transform $\neg(\diamond\Box p \rightarrow \Box\diamond p)$ into negation normal form:

$$\begin{aligned}\neg(\diamond\Box p \rightarrow \Box\diamond p) &\equiv \diamond\Box p \wedge \neg\Box\diamond p \\ &\equiv \diamond\Box p \wedge \diamond\neg\diamond p \\ &\equiv \diamond\Box p \wedge \diamond\Box\neg p\end{aligned}$$

It is shown in Example 11.2.3 that $\diamond\Box p \wedge \diamond\Box\neg p$ is unsatisfiable. Thus, $\diamond\Box p \rightarrow \Box\diamond p$ is valid.

On the other hand, the negation normal form of $\neg(\Box\diamond p \rightarrow \diamond\Box p)$ is $\Box\diamond p \wedge \Box\diamond\neg p$, which is satisfiable (Example 11.2.2). Thus, $\Box\diamond p \rightarrow \diamond\Box p$ is not valid. □

Formulas in propositional logic can be transformed into conjunctive normal form (CNF) or disjunctive normal form (DNF). However, this is possible neither in LTL nor in MTL. Like $\forall$ in first-order logic, $\Box$ does not distribute over $\vee$; like $\exists$, $\diamond$ does not distribute over $\wedge$.

Theorem 11.2.14. (Distributivity)

$$\begin{aligned}&(a) \quad \Box(p \wedge q) \equiv \Box p \wedge \Box q; \\ &(b) \quad \diamond(p \vee q) \equiv \diamond p \vee \diamond q; \\ &(c) \quad \circ(p \wedge q) \equiv \circ p \wedge \circ q; \\ &(d) \quad \circ(p \vee q) \equiv \circ p \vee \circ q.\end{aligned}$$

Proof. (a) For any interpretation sequence σ, $\Box(p \wedge q)$ is true in σ iff for any $i \geq 0$, $\sigma(i)(p \wedge q) = \top$. That is, $\sigma(i)(p) = \top$ and $\sigma(i)(q) = \top$ for any $i \geq 0$. This is the same as $\Box p$ and $\Box q$, respectively.

(b) can be obtained from (a) by duality. The proofs of (c) and (d) are left as exercise. □

Example 11.2.15. To show that $\Box(p \vee q)$ is not equivalent to $\Box p \vee \Box q$, consider an interpretation sequence σ where p is true at each odd-number state of σ and q is true at each even-number state of σ. Then $p \vee q$ is true at every state of σ, hence $\Box(p \vee q)$ is true in σ. It is easy to see that neither $\Box p$ nor $\Box q$ is true in σ. Note that $\Box p \vee \Box q \not\models \Box(p \vee q)$. □

Theorem 11.2.16. (Absorption)

$$\begin{aligned}&(a) \quad \Box\Box p \equiv \Box p; \\ &(b) \quad \diamond\diamond p \equiv \diamond p; \\ &(c) \quad \Box\diamond\Box p \equiv \diamond\Box p; \\ &(d) \quad \diamond\Box\diamond p \equiv \Box\diamond p.\end{aligned}$$

Proof. The proofs of (a) and (b) are straightforward. (c) can be obtained from (d) by duality. Here is the proof of (d): For any interpretation sequence σ, $\Diamond\Box\Diamond p$ is true in σ iff there exists $i \geq 0$, $\Box\Diamond p$ is true in $\sigma_{\geq i}$. $\Box\Diamond p$ is true in $\sigma_{\geq i}$ iff for all $j \geq i$, $\Diamond p$ is true in $\sigma_{\geq j}$. $\Diamond p$ is true in $\sigma_{\geq j}$ iff $\Diamond p$ is true in $\sigma_{\geq k}$ for any $k \geq 0$, which is equivalent to $\Box\Diamond p$ is true in σ. □

The above theorem allows us to compress series of $\Box$ and $\Diamond$ operators to no more than two. They are useful in the simplification of LTL formulas.

11.3 Semantic Tableaux for LTL

The method of semantic tableaux introduced in Chapter 2 is a decision procedure for satisfiability in propositional logic. In this section, we may extend this method for LTL by adding rules for the modal operators. The extension for MTL is similar. LTL.

The construction of an LTL model is more complex than that it is for a propositional model because a propositional model is a single propositional interpretation; an LTL model is a sequence of interpretations. We need to find an interpretation for each state in the sequence. Therefore, we need to group nodes in the semantic tableau by states. For example, the β-rule for $\vee$ will generate two children for the node containing $p \vee q$, and these two children belong to the group of their parent. On the other hand, $\circ p$ will generate a child node which belongs to a different group associated with the next state of the current state.

Like propositional logic, the extended tableau method takes an input formula A, create the initial node containing A, and then applies the rules repeatedly to any leaf node to generate its successors. The tableau rules for LTL consist of the rules for propositional logic shown in Section 3.1.1, plus a new α-rule for $\Box$, a new β-rule for $\Diamond$, a new type rule called X-rule for $\circ$. To simplify the discussion, we will apply some simplification rules to the initial formula. The simplification rules include the following rewrite rules and some rules involving the constants $\top$ and $\bot$ (such as $\bot \vee A \rightarrow A$ and $\Box\top \rightarrow \top$).

$$
\begin{array}{lrcl}
(1) & \neg\neg A & \rightarrow & A. \\
(2) & \neg(A \wedge B) & \rightarrow & \neg A \vee \neg B; \\
(3) & \neg(A \vee B) & \rightarrow & \neg A \wedge \neg B; \\
(4) & \neg(A \rightarrow B) & \rightarrow & A \wedge \neg B; \\
(5) & \neg(A \leftrightarrow B) & \rightarrow & (A \wedge \neg B) \vee (\neg A \wedge B); \\
(6) & \neg\Box A & \rightarrow & \Diamond\neg A; \\
(7) & \neg \Diamond A & \rightarrow & \Box\neg A; \\
(8) & \Box \circ A & \rightarrow & \circ\Box A; \\
(9) & \neg \circ A & \rightarrow & \circ\neg A; \\
(10) & \Diamond \circ A & \rightarrow & \circ \Diamond A;
\end{array}
$$

The last three rules are the commutativity rules which move $\circ$ outward; the other rules move $\neg$ inward. These simplification rules are based on logical equivalences and preserve the equivalence relation between the two formulas before and after the application of a rewrite rule. The rules will transform the initial formula to negation normal forms where $\neg, \vee, \wedge, \circ, \Box, \Diamond$ are the only operators.

11.3.1 Rules for Modal Operators

In Chapter 3, we showed how a tableau is created by creating the initial node of the input formula and applying α and β rules to a formula in a node to create successor nodes. A α-rule creates one successor and a β-rule creates two successors. Let n be the current node and $F(n)$ be the set of formulas contained in n. If a α-rule can replace A by $\{B, C\}$ for $A \in F(n)$, then the successor of n, denoted n', is created and $F(n') = F(n) - \{A\} \cup \{B, C\}$. If a β-rule can replace A by B and C for $A \in F(n)$, then two successors of n, denoted n' and n'', are created such that $F(n') = F(n) - \{A\} \cup \{B\}$ and $F(n'') = F(n) - \{A\} \cup \{C\}$.

As in Chapter 3, given a tableau, we say a node is *closed* if it contains a pair of complementary literals. A node is *open* if it is not closed and contains only literals. A node is *expandable* if it is a leaf node which is neither closed nor open. Expandable nodes always contain a formula which allows one of α or β rules to apply. For LTL, we have a new type of nodes, called X-nodes.

Definition 11.3.1. (X-formula, X-node, and X-rule) *If the root symbol of an LTL formula A is $\circ$ and no rules can apply to A, A is called a* next *formula, or X-formula*

*A node of a semantic tableau is called an X-*node *if each formula in the node is either a literal or a X-formula.*

Let F be the formulas in a X-node n:

$$F(n) = \{A_1, A_2, ..., A_k, \circ B_1, \circ B_2, ..., \circ B_m\},$$

where $k \geq 0$, $m \geq 1$, and $\{A_1, A_2, ..., A_k\}$ is a set of consistent literals, then the X-rule will generate an interpretation, which makes all $A_1, A_2, ..., A_n$ true, for the current state and create a new node n', where $F(n') == \{B_1, B_2, ..., B_m\}$ for the next state.

Example 11.3.2. Given the input formula $\neg p \wedge \circ p$, the initial node contains $\{\neg p \wedge \circ p\}$. Applying α-rule for $\wedge$, we arrive at the second node which contains $\{\neg p, \circ p\}$. Here $\neg p$ is a literal, $\circ p$ is an X-formula, and no rules can apply, except the X-rule. The X-rule will generate the third node containing p and this node belongs to a different state. From the second node, we create $s_0 = \{\neg p\}$; from the third node, we create $s_1 = \{p\}$. The interpretation sequence created from this tableau is $s_0, s_1, s_1, s_1, ...$, which is a model of $\neg p \wedge \circ p$. □

By definition, the X-rule can apply to X-nodes only. Neither simplification rules, nor α and β rules can apply to any formula in X-nodes. More application examples of X-rules will be given shortly, after presenting the rules for $\Box$ and $\Diamond$.

To construct a model for $\Box A$, we need to find an interpretation sequence σ such that A is true in every suffix of σ. That is, A is true in $\sigma(0)$ and $\Box A$ must be true in all subsequent states. That is, instead of $\Box A$, we will construct a model for $A \wedge \circ \Box A$. Intuitively, if $\circ$ represents "tomorrow", then A is true everyday iff A is true today and every tomorrow.

Theorem 11.3.3. (Induction)

$$\begin{array}{lrcl} (a) & \Box A & \equiv & A \wedge \circ \Box A; \\ (b) & \Diamond A & \equiv & A \vee \circ \Diamond A. \end{array}$$

Proof. (a) For any interpretation sequence σ, $A \wedge \circ \Box A$ is true in σ iff A is true in $\sigma_{\geq 0}$ and $\Box A$ is true in $\sigma_{\geq 1}$. $\Box A$ is true in $\sigma_{\geq 1}$ iff A is true in $\sigma_{\geq i}$ for every $i \geq 1$. Thus A must be true in $\sigma_{\geq i}$ for every $i \geq 0$, which is exactly "$\Box A$ being true in σ" by definition.

b is the dual case of (a). □

Part (a) of the above theorem serves as an inductive tool for constructing a model of $\Box A$: A is the base case and $\circ \Box A$ is the inductive case. Once both cases are finished, we obtain a model for $\Box A$. Similarly, part (b) provides a tool for constructing a model for $\Diamond A$: A is the base case; if a model of A is found, we use it as a model of $\Diamond A$. Otherwise, we seek a model from the inductive case and try to find a model of $\circ \Diamond A$.

Definition 11.3.4. *The α-rule for $\Box A$ and the β-rule for $\Diamond A$ are given below:*

α	α_1, α_2
$\Box A$	$A, \circ\Box A$

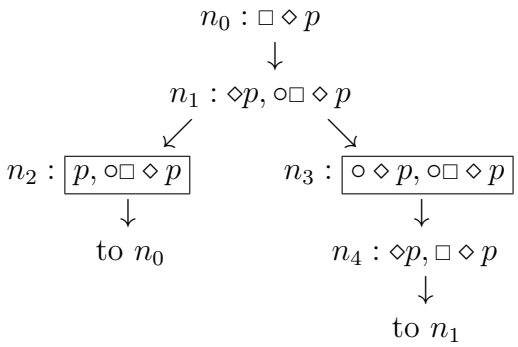
β	β_1	β_2
$\Diamond A$	A	$\circ \Diamond A$

The soundness of these two rules is ensured by Theorem 11.3.3.

The construction of semantic tableaux for LTL is similar to that of propositional logic by repeating α-, β-, and X-rules to open nodes, until no rules can apply. However, there are one minor and one major differences. The minor difference is that we apply the simplification rules first to the input formula to get a negation normal form before the start of tableau construction.

The major difference is that we need to check if a new node contains the same set of formulas as an existing node. If yes, instead of creating a new node, we create a link to the existing node. Because of these links, the created tableau is still a directed graph but no longer a tree. This graph may contain cycles and each node has at most two successors (i.e., two outgoing links). The termination of the tableau construction depends on this major difference as a tableau may contain paths of infinite length.

Example 11.3.5. The tableau for $\Box \Diamond p$ is shown below, where the X-nodes are boxed.



□

Given a tableau as directed graph, we still call a node without outgoing links a *leaf*, which is either closed, open, or expandable. Given two nodes n and n' of a directed graph, n' is said to be *reachable* from n if there exists a path from n to n' in the graph.

For any node n of a tableau, let $F(n)$ denote the set of formulas appearing in n. As in Chapter 3, we assume that $F(n) \equiv \bigwedge_{A \in F(n)} A$, that is, a set of formulas is equivalent to the conjunction of the formulas in the set.

Theorem 11.3.6. (*a*) *The construction of a tableau for an LTL formula always terminates.*

(*b*) *If node n' is derived from n by a α-rule, then $F(n) \equiv F(n')$; if n' and n'' are derived from n by a β-rule, then $F(n) \equiv F(n') \vee F(n'')$.*

(*c*) *If node n' is derived from n by X-rule, then an interpretation sequence σ' is a model of $F(n')$ iff $\sigma = s_0\sigma'$ is a model of $F(n)$, where $s_0 \in IV_P$ is an interpretation which makes every literal of $F(n)$ true.*

Proof. (*a*) The construction stops when no expandable nodes are available. If the construction does not stop, then there will be an infinite path of nodes in the tableau such that each node contains a different set of formulas. Hence the collection of formulas from these nodes is infinite. However, the rules used in the construction can only generate subformulas of the original formula, or adding at most one $\circ$ operator to them. In other words, only a finite number of formulas can be generated. This is a contradiction.

(*b*) All the α and β rules used in the construction preserve the equivalence relation.

(*c*) When the X-rule applies to n, $F(n) = \{A_1, A_2, ..., A_k, \circ B_1, \circ B_2, ... \circ B_m\}$, A_1, A_2, ..., A_k are literals and $F(n') = \{B_1, B_2, ...B_m\}$. By the assumption, $F(n) \equiv (\bigwedge_{1\leq i\leq k} A_i) \wedge \circ(\bigwedge_{1\leq j\leq m} B_j)$. Let σ' be any interpretation sequence and s_0 be an interpretation such that $s_0(A_i) = \top$ for $1 \leq i \leq k$. Then σ' is a model of $F(n')$, i.e., $\bigwedge_{1\leq j\leq m} B_j)$ is true in σ', iff $s\sigma'$ is a model of $\circ(\bigwedge_{1\leq j\leq m} B_j)$ for any $s \in IV_P$, or $\sigma = s_0\sigma'$ is a model of $(\bigwedge_{1\leq i\leq k} A_i) \wedge \circ(\bigwedge_{1\leq j\leq m} B_j)$, since s_0 is a model of $\bigwedge_{1\leq i\leq k} A_i$. That is, $\sigma = s_0\sigma'$ is a model of $F(n)$. □

Part (*a*) of the above theorem tells that the construction of a tableau for an LTL formula will terminate such that every leaf is either closed or open, without expandable nodes. If the tableau has an open node, parts (*b*) and (*c*) tell us how to construct a model (for the input formula in the initial node) from this open node bottom-up, thus have shown that the original formula is satisfiable. What happens when a tableau has no open nodes? We will discuss this case next.

11.3.2 Deciding Satisfiability by Tableaux

We have seen how a tableau can be constructed for every LTL formula. For propositional logic, a formula is satisfiable iff its tableau has an open node. For LTL, at least one side is true.

Theorem 11.3.7. *If the tableau for an LTL formula A has an open node, then A is satisfiable.*

Proof. Let $(n_0, n_1, ..., n_m)$ be a simple path from the initial node n_0 to the open node

n_m in the tableau. We show that $F(n_j)$ has a model σ_j for $j = m, m-1, ..., 1, 0$ by induction.

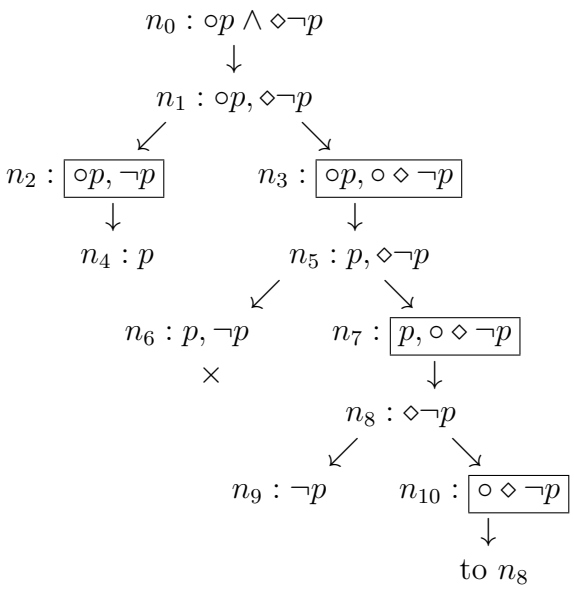
For the base case when $j = m$, $F(n_m)$ is a set of consistent literals. Let s_m be the interpretation which makes every literal of $F(n_m)$ true and $\sigma_m = s_m^+$, where s_m^+ means we repeat s_m forever, then σ_m is a model of $F(n_m)$.

As induction hypothesis, we assume that σ_j is a model of $F(n_j)$. If n_j is a successor of n_{j-1} by either α-rule or β-rule, by Theorem 11.3.6 (b), $\sigma_{j-1} = \sigma_j$ is also a model of $F(n_{j-1})$. If n_j is the successor of n_{j-1} by the X-rule, by Theorem 11.3.6 (c), $\sigma_{j-1} = s_{j-1}\sigma_j$ is a model of $F(n_{j-1})$, where s_{j-1} is an interpretation made from the literals in $F(n_{j-1})$.

Finally when $j = 0$, σ_0 is a model of $F(n_0) = \{A\}$. Thus A is satisfiable. □

For LTL, a tableau may not have an open node, as seen in Example 11.3.5. That tableau does not have closed nodes, either. How can we tell the input formula of the tableau is satisfiable? Let us look at another example.

Example 11.3.8. To show that $\circ p \wedge \diamond \neg p$ is satisfiable, we may construct a model by semantic tableau. Using the tableau rules, we obtain the following tableau, where X-nodes are boxed.



A model is a sequence of interpretations and an interpretation can be created either from an open node or X-node. For instance, consider the path (n_0, n_1, n_2, n_4): from n_2, we create $s_0 = \{\neg p\}$; from n_4, we create $s_1 = \{p\}$. The first two interpretations of our model are s_0 and s_1, respectively. By repeating s_1, we obtain an infinite interpretation sequence denoted by $s_0 s_1^+$, where s_1^+ means we repeat s_1 forever. It

is easy to check that $s_0 s_1^+$ is a model of $\circ p \wedge \diamond \neg p$. Since the interpretations after s_1 does not affect the truth value of p, let s_2 denote an interpretation where p takes any value, then a general representation of the models from this path is $s_0 s_1 s_2^*$, where s_2^* means we can repeat s_2 any finite number of times.

Consider another path ending with an open node:

$$(n_0, n_1, n_3, n_5, n_7, n_8, n_9)$$

where n_3 and n_7 are X-nodes and n_9 is open. n_3 does not specify the truth value of p, and we name the interpretation as s_2 where p can take any truth value. The interpretations created from n_7 and n_9 are $s_0 = \{p\}$ and $s_1 = \{p\}$, respectively. The model created from this path is $s_2 s_1 s_0 s 2^*$.

Consider another path

$$(n_0, n_1, n_3, n_5, n_7, n_8, n_{10})$$

where n_3, n_7, and n_{10} are X-nodes. If the interpretations created from them always make p true, then no model can be found from this path. On the other hand, if the path is

$$(n_0, n_1, n_3, n_5, n_7, n_8, n_{10}, n_8, n_9),$$

the model from this path is $s_2 s_1 s_2 s_0 s 2^*$. If we repeat n_8 and n_{10} a couple of times, then all of the models from this path can be represented by $s_2 s_1 s_2^* s_0 s 2^*$.

In fact, all the models of $\circ p \wedge \diamond \neg p$ can be denoted by $s_0 s_1 s_2^* \cup s_2 s_1 s_2^* s_0 s_2^*$. Each model corresponds to some path in the tableau. □

The above example tells us that when there is an open node, we can construct a model for the input formula. When the path ends with a cycle, a model may or may not be constructed.

Example 11.3.9. Looking back at the tableau in Example 11.3.5: It has no open nodes. If we pick the cycle (n_0, n_1, n_2), let $s_1 = \{p\}$ be the interpretation derived from the X-node n_2, then s_1^+ is a model of the input formula $\Box \diamond p$.

Let us look the path (n_0, n_1, n_3, n_4), where the last three nodes are in a cycle. The interpretation created from n_3 does not care about the truth value of p and we denote it by s_2, then s_2^+ is not a model.

On the other hand, consider the path

$$(n_0, n_1, n_2, n_0, n_1, n_3, n_4).$$

The sequence $(s_1 s_2)^+$ is a model of $\Box \diamond p$, while the sequence $s_1 s_2^+$ is not a model. In other words, we need to repeat altogether all the interpretations generated from the nodes in a cycle.

Note that all the models of $\Box \diamond p$ can be denoted by the sequence $(s_1 s_2 *)^+$. □

The above examples show that some paths ending with a cycle generate models and some don't. Our purpose of using tableaux is to decide if the input formula is satisfiable or not. A model is just an evidence that the input formula is satisfiable. If a tableau has an open node, we know the input formula is satisfiable because a model can be constructed from the path which connects the initial node and the open node. If the tableau has no open nodes, we define below a marking mechanism to check effectively if the input formula is satisfiable or not.

Definition 11.3.10. *Given a tableaux which has no expandable nodes. A node n is marked* dead *if one of the following conditions is true:*

1. *n is closed;*
2. *all successors of n are marked dead; or*
3. *for any formula $\diamond A \in F(n)$, there is no node n' reachable from n and $A \in F(n')$.*

In the third case, the reachable condition is checked among the nodes which are not marked dead. That is, if a node is marked dead, it cannot be used in any path. Node n is marked dead because it contains an *unfulfilled* $\diamond A$: $\diamond A$ is true in a sequence σ iff A is true in a suffix of σ. If A does not appear in any node n' reachable from n, then it is impossible to construct such a suffix of σ such that σ is a model of $F(n)$.

Example 11.3.11. To show that $\Box(p \wedge q) \to \Box p$ is valid, we show that its negation is unsatisfiable by semantic tableau. The negation normal form of $\neg(\Box(p \wedge q) \to \Box p)$ is $\Box(p \wedge q) \wedge \diamond\neg p$, whose tableau is given below:

$$
\begin{array}{c}
n_0 : \Box(p \wedge q) \wedge \diamond\neg p \\
\downarrow \\
n_1 : \Box(p \wedge q), \diamond\neg p \\
\downarrow \\
n_2 : p \wedge q, \circ\Box(p \wedge q), \diamond\neg p \\
\downarrow \\
n_3 : p, q, \circ\Box(p \wedge q), \diamond\neg p \\
\swarrow \qquad\qquad \searrow \\
n_4 : p, q, \circ\Box(p \wedge q), \neg p \qquad n_5 : \boxed{p, q, \circ\Box(p \wedge q), \circ \diamond \neg p} \\
\times \qquad\qquad\qquad\qquad \downarrow \\
\qquad\qquad\qquad\qquad\qquad \text{to } n_1
\end{array}
$$

For $\diamond\neg p \in F(n_1)$, there is no node reachable from n_1. Thus, n_1 is marked dead. n_0 is marked dead because its successor is marked dead. In fact, every node in this tableau can be marked dead. For instance, n_5 is marked dead because its successor is marked dead. □

Example 11.3.12. For the tableau given in Example 11.3.5, $\diamond p$ appears in n_4. Since n_2 is reachable from n_4 through the path (n_4, n_1, n_2), n_4 cannot be marked dead. □

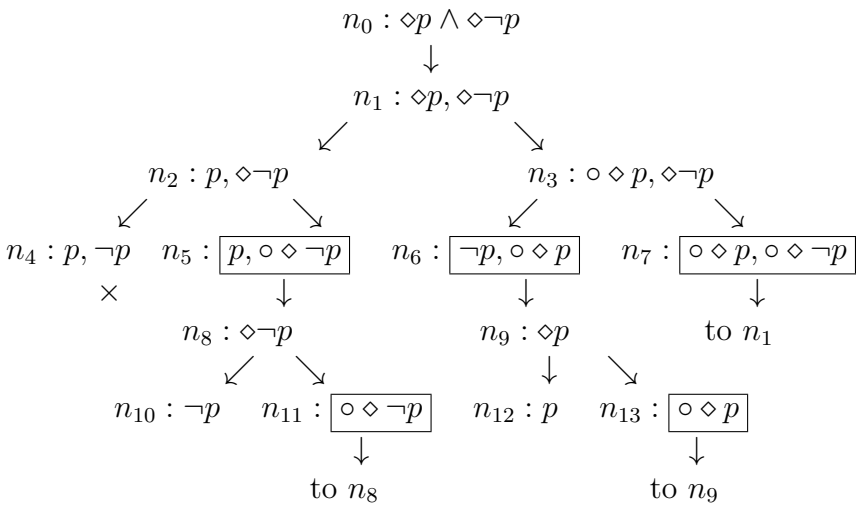
Algorithm 11.3.13. The algorithm *satisfiabilityByTableau* takes an LTL formal A as input and returns "satisfiable" if A is satisfiable; otherwise returns "unsatisfiable".

proc *satisfiabilityByTableau*(A)
1. Create a tableau for A:
   1.1 Create the initial node that contains A;
   1.2 Apply the α, β, and X-rules until no expandable nodes exist.
2. Decide if A is satisfiable:
   2.1 If the tableau has an open node, return "satisfiable";
   2.2 Apply Definition 11.3.10 until no nodes can be marked dead;
   2.3 If the initial node is marked dead, return "unsatisfiable";
   2.4 Otherwise return "satisfiable";

Although the algorithm returns only "satisfiable" or "unsatisfiable", we can still construct a model from a path of the tableau if the algorithm returns "satisfiable", as illustrated by the following examples.

Example 11.3.14. To show that $\diamond p \wedge \diamond\neg p$ is satisfiable, we construct its semantic

tableau.



For $\diamond p \in F(n_1)$, either n_5 or n_{12} contains p; for $\diamond \neg p \in F(n_1)$, either n_6 or n_{10} contains $\neg p$. All the four nodes are reachable from n_1. n_0 cannot be marked dead, so the algorithm will return "satisfiable".

Various models can be constructed from this tableau, depending on which path we choose in the tableau. Let $s_0 = \{p\}$, $s_1 = \{\neg p\}$, and p_2 denotes an interpretation where we do not know the truth value of p.

1. $(n_0, n_1, n_2, n_5, n_8, n_{10})$: The derived model is $s_0 s_1 s_2^*$, where s_0 is derived from n_5 and s_1 from n_{10}. s_2^* is added at the end because n_{10} is open.

2. $(n_0, n_1, n_3, n_6, n_9, n_{12})$: The derived model is $s_1 s_0 s_2^*$, where s_1 is derived from n_6 and s_0 from n_{12}.

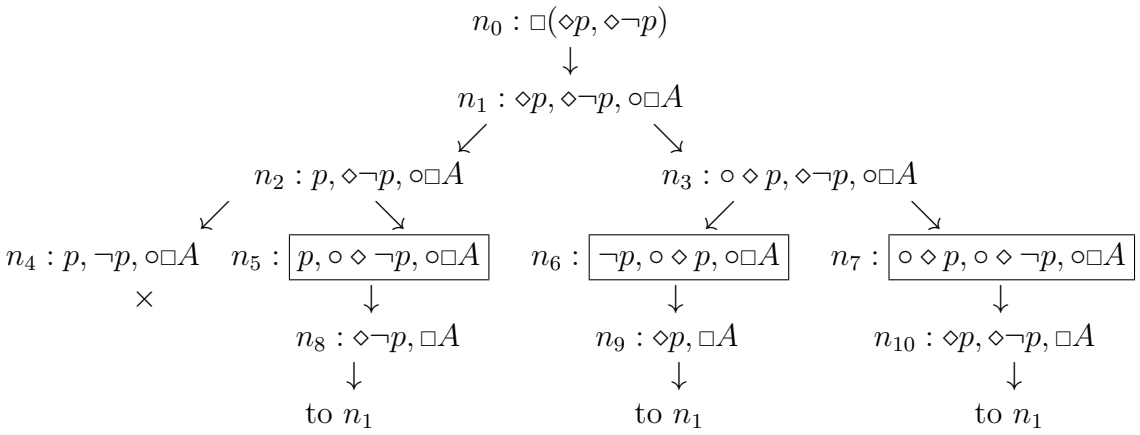
3. $(n_0, n_1, n_2, n_5, n_8, n_{11})$: The last two nodes are in a cycle and no model can be found from this path. To fulfill $\diamond \neg p \in F(n_1)$, we need one of n_6 or n_{10}.

4. $(n_0, n_1, n_3, n_6, n_9, n_{13})$: The last two nodes are in a cycle and no model can be found from this path.

5. (n_0, n_1, n_3, n_7): The last three nodes are in a cycle and no model can be found from this path.

All models of $\diamond p \wedge \diamond \neg p$ can be denoted by $s_2^* s_1 s_2^* s_0 s_2 * \cup s_2^* s_0 s_2^* s_1 s_2 *$ and each model corresponds to some path in the tableau. $\square$

Example 11.3.15. To show that $\Box(\Diamond p \wedge \Diamond \neg p)$ is satisfiable, we construct a semantic tableau where we replace $\wedge$ by "," and A stands for $(\Diamond p, \Diamond \neg p)$.



This tableau has one closed node, i.e. n_4, and no open nodes. n_4 is the only node marked dead. $\Diamond p$ and $\Diamond \neg p$ in n_1 are fulfilled by n_5 ad n_6, respectively. n_1 cannot be marked dead; neither does n_0. So the algorithm will return "satisfiable".

If we wish to construct a model from the tableau, we need to consider a path starting from n_0 and containing both n_5 and n_6. Let $s_0 = \{p\}$ and $s_1 = \{\neg p\}$ be the interpretations derived from X-nodes n_5 and n_6, respectively. Then $(s_0 s_1)^+$ is a model of the input formula. Any model requires that we repeat these two interpretations together; repeating a single interpretation infinitely does not produce a model. Let s_2 be the interpretation where we do not know the value of p. All the models of $\Box(\Diamond p \wedge \Diamond \neg p)$ can be represented by $(s_2^* s_0 s_2^* s_1 s_2^*)^+ \cup (s_2^* s_1 s_2^* s_0 s_2^*)^+$ and each model corresponds some path in the tableau. □

The following theorem provides the correctness of Algorithm 11.3.13.

Theorem 11.3.16. *A LTL formula A is satisfiable iff Algorithm'11.3.13 returns "satisfiable".*

Proof. At first, the algorithm will terminate because (1) the construction of the tableau will terminate (Theorem 11.3.6 (a)) and (2) checking if the initial node is marked dead takes a finite number of steps.

If the tableau has an open node, A is satisfiable by Theorem 11.3.7.

If the initial node is not marked dead, we show that there is a model for A. This model is constructed from a path that starts with the initial node and contains all the nodes required by the *fulfilling* condition.

The path is selected among the nodes not marked dead as follows: (1) the path starts with the initial node; (2) if the last node of the current path has only

one unmarked successor, add the successor to the path; (3) if the current path contains a node n and $\diamond A$ in n has not fulfilled yet, find node n' reachable from n and contain A. If n' does not appear in the path, append the path from n to n' to the current path. If n' already appears in the path, add only necessary nodes so that all the nodes of the cycle containing n and n' are present in the path. That is, the last portion of the path contains a list of nodes from the same cycle and they are reachable to each other.

Note that both if $\diamond A$ and $\diamond B$ appear in n, we will work on them one at a time: When $\diamond A$ is fulfilled in n', either $\diamond B$ has been fulfilled in the path upto n' or $\diamond B$ is still present in n'. In the later case, there is another node n'' reachable from n' and containing B (if not, n' would have been marked dead).

By repeating the selection, one obtains a path ending with a cycle and all $\diamond A$'s are fulfilled. Let $n_{k_0}, n_{k_1}, ..., n_{k_m}$ be the X-nodes in the path and $s_0 s_1 \cdots s_m$ be the interpretations from these X-nodes (keeping them in the same order), we define an interpretation sequence

$$\sigma = s_0 s_1 s_2 \cdots (s_j \cdots s_m)^+,$$

where $s_j, ..., s_m$ are the interpretations generated from the X-nodes in the cycle. It is ready to check that σ is a model of the initial formula by a structural induction on the formulas in each node.

We claim that $\sigma_{\geq i}$ is a model of $F(n_{k_i})$ for $0 \leq i \leq m$. For any node n other than X-nodes, if n_{k_i} is the first X-node following n in the path for some i, then $F(n) \equiv F(n_{k_i})$ and $\sigma_{\geq i}$ is also a model of $F(n)$. For any formula $A \in F(n)$, the following statements are true.

- If A does not have any temporal operator, then A is true in $\sigma_{\geq i}$;
- If A is $\circ B$, then A is true in $\sigma_{\geq i}$ because B appears in the node n' following n_{k_i} by the X-rule and B is true in the model of n' by induction hypotheses (because B is smaller than $\circ B$ and this model is the immediate suffix of $\sigma_{\geq_i}$).
- If A is $\diamond B$, then A is true in $\sigma_{\geq i}$ because B appears in a reachable node n' from n and B is true in the model for n' by induction hypotheses (B is smaller than A) and this model is a suffix of $\sigma_{\geq i}$.
- If A is $\Box B$, then A is true in $\sigma_{\geq i}$ because B is true in $\sigma_{\geq i}$ (by the α-rule for $\Box$) and $\Box A$ is also true in the immediate suffix of $\sigma_{\geq i}$ (by the X-rule).

On the other hand, if the initial node is marked dead, then A is unsatisfiable because for any node n which is marked dead, $F(n)$ is unsatisfiable. This statement

can be proved by induction according to the recursive definition of that "n is marked dead". There are cases to consider:

1. *n is closed*: $F(n)$ contains a pair of complementary literals and cannot be satisfiable.

2. *all successors of n are marked dead*: Using induction hypotheses, the formulas in the successors are unsatisfiable. $F(n)$ is unsatisfiable by Theorem 11.3.6 (b) and (c).

3. *$\diamond A \in F(n)$ and there is no node reachable from n and containing A*: $\diamond A$ cannot be true in any interpretation sequence σ, since any model corresponds to some path in the tableau. If A does not appear in any path starting from n, A cannot be true in any suffix of σ. Hence $\diamond A$ is false in any σ. □

The above theorem ensures that Algorithm 11.3.13 is a decision procedure for LTL. In the literature, there is another decision procedure based on Büchi automaton (BA), which is an extension of finite-state automaton for input strings of infinite length. For LTL, these input strings represent interpretation sequences. Given a tableau, a Büchi automaton can be easily constructed from the tableau by keeping only X-nodes and open nodes (they are not marked dead) as *states*. A transition from one state to another is added if there is a path from the corresponding node to another in the tableau. The language recognized by BA should be the set of models of an LTL formula. There are algorithms that translate an LTL formula to a Büchi automaton without tableaux. These algorithms differ in their construction strategies but they all have common underlying principle, i.e., each state in the constructed automaton represents a set of LTL formulas that are expected to be satisfied by the remaining input string after occurrence of the state during an execution of BA. Büchi automata have important application in finite state model checking for formal verification of concurrent programs and its presentation is out of the score of this book.

11.4 Binary Temporal Operators

In temporal logic, we sometimes want to express a relationship of two statements in terms of time. For example, after *start*, a computer will keep *running* **until** it's *halted*. To express this sentence, we write

$$start \rightarrow \circ(running \; \mathcal{U} \; halted)$$

where the binary temporal operation $\mathcal{U}$ has the meaning of **until**.

11.4.1 The *until* and *release* Operators

The intuitive meaning of $(A \mathcal{u} B)$ is that A has to be true at least until B becomes true and B must hold at the current or a future state.

The operator $\mathcal{u}$ has a dual operator called *release*, denoted by $\mathcal{R}$. The intuitive meaning of $A \mathcal{R} B$ is that B has to be true until and including the state where A first becomes true; if A never becomes true, B must remain true forever.

We will extend LTL by adding $\mathcal{u}$ and $\mathcal{R}$ to the definition of LTL formulas such that if A and B are LTL formulas, so are $(A \mathcal{u} B)$ and $(A \mathcal{R} B)$.

We also need to extend Definition 11.2.1 with the following condition:

Given an interpretation sequence σ, an LTL formula C is true in σ if

- *C is $A \mathcal{u} B$ and there exists $i \geq 0$, $\sigma_{\geq i} \models B$ and for all $0 \leq j < i$, $\sigma_{\geq j} \models A$.*
- *A is $A \mathcal{R} B$ and either $\Box B$ or there exists $i \geq 0$, $\sigma_{\geq i} \models A$ and for all $0 \leq j \leq i$, $\sigma_{\geq j} \models B$.*

Example 11.4.1. By definition, $p \mathcal{u} q$ is true in σ means q will be eventually true and p must be true before q becomes true. Let $s_0 = \{p, q\}$, $s_1 = \{p, \neg q\}$, $s_2 = \{\neg p, q\}$, and $s_3 = \{\neg p, \neg q\}$.

$p \mathcal{u} q$ is true in s_0^+, $s_1 s_0^+$, $s_1^* s_0^+$, $s_1 s_2^+$, $s_1^* s_2^+$, s_2^+, and $s_2 s_3^+$.
$p \mathcal{u} q$ is false in s_1^+, s_3^+, and $s_1^* s_3^+$. □

The following theorem displays some properties of $\mathcal{u}$.

Theorem 11.4.2. *For any LTL formulas A and B,*

(a) **(idempotency)**	$A \mathcal{u} A$	$\equiv$	A;
(b) **(abbreviation)**	$\Diamond A$	$\equiv$	$\top \mathcal{u} A$;
(c) **(absorption)**	$A \mathcal{u} (A \mathcal{u} B)$	$\equiv$	$A \mathcal{u} B$;
(d) **(absorption)**	$(A \mathcal{u} B) \mathcal{u} B$	$\equiv$	$A \mathcal{u} B$;
(e) **(distributivity)**	$\circ(A \mathcal{u} B)$	$\equiv$	$(\circ A) \mathcal{u} (\circ B)$;
(f) **(induction)**	$A \mathcal{u} B$	$\equiv$	$B \vee A \wedge \circ(A \mathcal{u} B)$.

Proof. (a) For any interpretation sequence σ, $\sigma_{\geq 0} = \sigma$. If A is true in $\sigma_{\geq 0}$, then $i = 0$ and for $1 \leq j < i$, A is true in $\sigma_{\geq j}$ vacuously. So A is true in $\sigma_{\geq 0}$. If A is false in $\sigma_{\geq 0}$, then $A \mathcal{u} A$ is false in $\sigma_{\geq}$ by definition.

(b) Here B is $\top$ and C is A, the truthfulness of $B \mathcal{u} C$ is reduced to the first condition that there exists $i \geq 0$, $\sigma_{\geq i} \models C$, identical to that of $\Diamond C$.

(c) For any interpretation sequence σ, if $\diamond B$ is false, then both $A\mathcal{U}(A\mathcal{U}B)$ and $A\mathcal{U}B$ are false in σ. If $\diamond B$ is true, let k be the state such that B is true in $\sigma_{\geq k}$ but false in $\sigma_{\geq j}$ for $1 \leq j < k$. Since B is true in $\sigma_{\geq k}$, $A\mathcal{U}B$ is trivially true in $\sigma_{\geq k}$. $A\mathcal{U}B$ is true in σ iff A is true in $\sigma_{\geq j}$ for $1 \leq j < k$. $A\mathcal{U}(A\mathcal{U}B)$ is true in σ iff A is true in $\sigma_{\geq j}$ for $1 \leq j < k$, too.

The proofs of (d) and (e) are left as exercise.

(f) For $A\mathcal{U}B$ to be true, either B is true now, or we put off to the next tick the requirement to satisfy $A\mathcal{U}B$, while requiring that A be true now. □

Example 11.4.3. By definition, $p\mathcal{R}q$ is true in σ means either q is true forever, or after p becomes true, the condition of q being true is released. Let $s_0 = \{p, q\}$, $s_1 = \{p, \neg q\}$, $s_2 = \{\neg p, q\}$, and $s_3 = \{\neg p, \neg q\}$.

$p\mathcal{R}q$ is true in s_0^+, s_2^+, $s_2s_0^+$, and $s_2s_0s_3^+$.

$p\mathcal{R}q$ is false in s_1^+, $s_1s_0^+$, $s_1^*s_0^+$, s_1^+, $s_1^*s_2^+$, $s_2s_1^+$, $s_2s_3^+$, and s_3^+. □

The following theorem displays some properties of $\mathcal{R}$.

Theorem 11.4.4. *For any LTL formulas A and B,*

(a) **(idempotency)**	$A\mathcal{R}A$	$\equiv$	A;
(b) **(abbreviation)**	$\bot\mathcal{R}A$	$\equiv$	$\Box A$;
(c) **(absorption)**	$A\mathcal{R}(A\mathcal{R}B)$	$\equiv$	$A\mathcal{R}B$;
(d) **(absorption)**	$(A\mathcal{R}B)\mathcal{R}B$	$\equiv$	$A\mathcal{R}B$;
(e) **(distributivity)**	$\circ(A\mathcal{R}B)$	$\equiv$	$(\circ A)\mathcal{R}(\circ B)$;
(f) **(induction)**	$A\mathcal{R}B$	$\equiv$	$B \wedge (A \vee \circ(A\mathcal{R}B))$;
(g) **(duality)**	$A\mathcal{R}B$	$\equiv$	$\neg(\neg A\mathcal{U}\neg B)$.

Proof. (a)–(f) can be deduced from (g) and Theorem 11.4.2. To simplify the proof of (g), we prove $p\mathcal{R}q \equiv \neg(\neg p\mathcal{U}\neg q)$, where p and q are propositional variables. Replacing $\sigma(i)$ by $\sigma_{\geq i}$ in the following proof would give us the proof of $A\mathcal{R}B \equiv \neg(\neg A\mathcal{U}\neg B)$.

Assume $p\mathcal{R}q$ is true in σ. If q is always true, $\neg p$ and $\neg p\mathcal{U}\neg q$ are always false. Thus $\neg(\neg p\mathcal{U}\neg q)$ is always true in σ. If q is not always true, let k be minimal such that q is false in $\sigma(k)$. Since $\sigma \models p\mathcal{R}q$, there must exist $i < k$ such that p is true in $\sigma(k)$ and for all $1 \leq j \leq i$, q is true in $\sigma(j)$. In this case, let us check the truth value of $\neg p\mathcal{U}\neg q$: since $\sigma(k)$ is the first interpretation in which $\neg q$ is true and $\neg p$ is false in $\sigma(i)$ for $i < k$, $\neg p\mathcal{U}\neg q$ must be false in σ by definition. Hence $\neg(\neg p\mathcal{U}\neg q)$ must be true in σ.

Now assume $p\mathcal{R}q$ is false in σ. q cannot be always true and let k be minimal such that q is false in $\sigma(k)$. Hence q is true in $\sigma(j)$ for $1 \leq j < k$. Because $p\mathcal{R}q$ is

false in σ, q must be false in $\sigma(j)$ for $1 \leq j < k$. Equivalently, $\neg q$ must be true in $\sigma(j)$ for $1 \leq j < k$. Hence $\neg p \mathcal{U} \neg q$ is true in σ because $\neg q$ is true in $\sigma(k)$. So we conclude that $\neg(\neg p \mathcal{U} \neg q)$ is false in σ. □

11.4.2 The *weak until* and *strong release* Operators

Some researchers also define a *weak until* binary operator, denoted $\mathcal{W}$, with semantics similar to that of the until operator but the stop condition is not required to occur (similar to *release*). The *strong release* binary operator, denoted $\mathcal{S}$, is the dual of *weak until*. It is defined similar to the *until* operator, so that the release condition has to be true at some point. Therefore, it is stronger than the *release* operator.

Definition 11.4.5. *The additional temporal operators are defined as follows:*

- **Weak until** $A \mathcal{W} B$*: A has to be true at least until B; if B never becomes true, A must remain true forever.*
- **Strong release** $A \mathcal{S} B$*: B has to be true until and including the state where A first becomes true, and A must be true at the current or a future state.*

Instead of extending Definition 11.2.1 to include the formal meanings of $A \mathcal{W} B$, and $A \mathcal{S} B$, we provide below the equivalent relations as an alternative definition:

Theorem 11.4.6. *For any LTL formulas A and B,*

$$\begin{array}{lrcl} (a)\ \textbf{(weak until)} & A \mathcal{W} B & \equiv & \Box A \vee (A \mathcal{U} B); \\ (b)\ \textbf{(strong release)} & A \mathcal{S} B & \equiv & \Diamond A \wedge (A \mathcal{R} B). \end{array}$$

The above theorem shows that $\mathcal{W}$ and $\mathcal{S}$ can be defined in terms of $\mathcal{U}$ and $\mathcal{R}$, just like $\rightarrow$ and $\leftrightarrow$ can be defined in terms of $\wedge$ and $\vee$. $\mathcal{R}$ can be defined in terms of $\mathcal{U}$ by duality. In fact, any one of the four binary temporal operators can be used to define the other three operators. We have seen in Theorems 11.4.2 and 11.4.4 that $\Diamond$ and $\Box$ can be defined in terms of $\mathcal{U}$ and $\mathcal{R}$. Thus, the minimal set of LTL needs only two temporal operators: $\circ$ and one of the four binary operators.

Theorem 11.4.7. *For any LTL formulas A and B,*

$$\begin{array}{lrcl}
(a) & A\,u\,B & \equiv & \diamond B \wedge (A\,w\,B);\\
(b) & A\,\pi\,B & \equiv & \Box B \vee (A\,s\,B);\\
(c) & A\,u\,B & \equiv & B\,s\,(A \vee B);\\
(d) & A\,\pi\,B & \equiv & B\,w\,(A \wedge B);\\
(e) & A\,w\,B & \equiv & B\,\pi\,(A \vee B);\\
(f) & A\,s\,B & \equiv & B\,u\,(A \wedge B);\\
(g) & A\,w\,B & \equiv & A\,u\,(\Box A \vee B);\\
(h) & A\,s\,B & \equiv & A\,\pi\,(\diamond A \wedge B);\\
(i) & \Box A & \equiv & A\,w\,\bot;\\
(j) & \diamond A & \equiv & A\,s\,\top;\\
(k) & \neg(A\,s\,B) & \equiv & \neg A\,w\,\neg B.
\end{array}$$

The last equivalence is the duality of w and s. The proof of this theorem is left as exercise.

11.4.3 Extension of Semantic Tableau

Using the duality and the rewrite rules $A\,w\,B \rightarrow \Box A \vee (A\,u\,B)$ and $A\,s\,B \rightarrow \diamond A \wedge (A\,\pi\,B)$, the negation can be pushed downward and all the formulas of LTL can be transformed into negation normal form, where all negations appear only in front of the propositional variables, and only the temporal operators $\Box$, $\diamond$, $\circ$, u, and π can appear. Note that the transformation to the negation normal form does not blow up the size of the formula.

Constructing a semantic tableau for a formula that uses the $A\,u\,B$ and $A\,\pi\,B$ operators is straightforward. The β-rule for $A\,u\,B$ and α-rule for $A\,\pi\,B$ are given below:

α	α_1, α_2
$A\,\pi\,B$	$B, A \vee \circ(A\,\pi\,B)$

β	β_1	β_2
$A\,u\,B$	B	$A, \circ(A\,u\,B)$

The α-rule is based on Theorem 11.4.4 (f), and the β-rule is based on Theorem 11.4.2 (f).

After a tableau is constructed use these rules, we need to decide if a model can be generated from the tableau. In this case, Definition 11.3.10 needs to be extended to consider the occurrence of $(A\,u\,B)$ in a node of the tableau: A node n is marked dead if

- for any formula $(A\,u\,B) \in F(n)$, there is no node n' reachable from n and $B \in F(n')$.

11.5 Verification of Concurrent Programs

This chapter begins with an example of how to access a bank account concurrently and correctly. The example serves as a motivation for the application of temporal logic for formal verification of concurrent programs. We will end this chapter with an example of formal verification, which is taken from a tutorial of the software system STeP (**S**tanford **Te**mporal **P**rover). STeP is a system developed by Zohar Manna et al. to support the computer-aided formal verification of concurrent and reactive systems based on temporal logic. STeP was designed with the objective of combining the expressiveness of deductive methods. Efficient simplification methods, decision procedures, and invariant generation techniques are invoked automatically to prove resulting first-order verification conditions with minimal assistance.

11.5.1 The BAKERY(2) Program

Figure 11.5.1 1 shows a program that implements Lamport's Bakery algorithm for mutual exclusion. The program is accepted as input by STeP. Two processes, $P1$ and $P2$, coordinate access to a critical section, where at most one process should reside at any given time. For instance, modifying the balance of a bank account should be done in a critical section. Each process selects a "ticket number" in y_1 and y_2, as the customers in a bank would, and the process with the lowest number is allowed to enter the critical section. Initially, $y_1 = y_2 = 0$. A ticket with value 0 indicates that the process is not interested in accessing the critical section. Since there are only two processes, it can be proved that $y_1, y_2 \in \{0, 1, 2\}$. This informal description of the algorithm can be made precise if the system and its properties are modeled formally.

The meaning of "**await** $(y_2 = 0 \vee y_1 \leq y_2)$" is that the execution pauses until the condition $(y_2 = 0 \vee y_1 \leq y_2)$ is true. This is the same as

$$\textbf{while } \neg(y_2 = 0 \vee y_1 \leq y_2) \text{ /* do nothing */}$$

The properties we would like to show are:

- *Mutual exclusion*: control is never at locations l_3 and m_3 at the same time.
- *Accessibility*: if control resides at l_1 (resp. m_1), it will always eventually reach l_3 (resp. m_3). That is, once a process has expressed interest in entering the critical section, it will eventually do so.

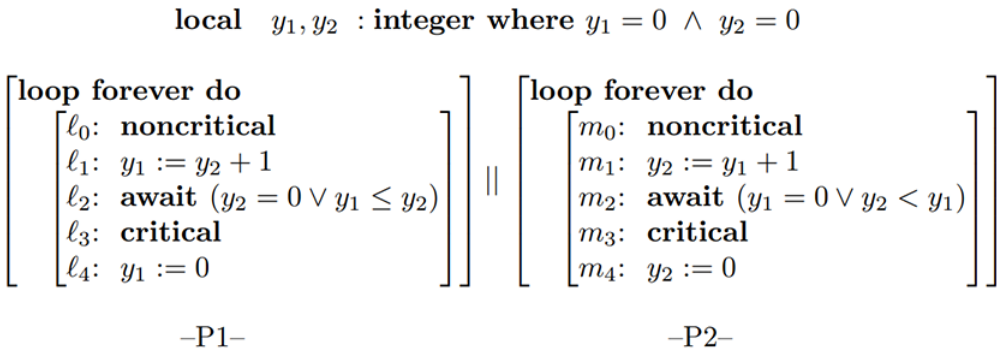


Figure 11.5.1: Program BAKERY(2)

- *One-bounded overtaking*: if one process wants to enter the critical section, the other process can enter the critical section at most once before the first one does.

Early solutions to mutual exclusion suffered from *starvation*, where a process could forever be denied access to the critical section, and other solutions do not satisfy one-bounded overtaking. Lamport's Bakery algorithm is the first solution that satisfied the three main properties above for the general N-process (in our example, $N = 2$).

11.5.2 States and Transition Conditions

The computation of a concurrent program is viewed as the interleaving of the atomic operations of its processes, where each process is a sequential program and has a control variable to locate the code of the current execution. STeP generates automatically the control variables π_1 for $P1$ and π_2 for $P2$, where $\pi_1, \pi_2 \in \{0, 1, 2, 3, 4\}$ and $\pi_1 = i$ (resp. $pi_2 = i$) iff the current execution of $P1$ (resp. $P2$) is at the line labeled by l_i (resp. m_i). A *state* of BAKERY(2) is defined by the values of (π_1, π_2, y_1, y_2), called *state variables*, and the initial state is $(0, 0, 0, 0)$. Since π_i has 5 values and y_2 has 3 values, the total number of states is 225 $(5 \times 5 \times 3 \times 3)$, though not all states are accessible from the initial state.

An execution of BAKERY(2) can be described by a sequence of states. For instance, the following sequence shows that $P1$ enters the **critical** section once, while $P2$ is still in the **noncritical** section.

$$(0,0,0,0) \Rightarrow (1,0,0,0) \Rightarrow (2,0,1,0) \Rightarrow (3,0,1,0) \Rightarrow (4,0,1,0) \Rightarrow (0,0,0,0) \Rightarrow \cdots$$

A *transition* of the states is just one move in a state sequence. Transitions are dominated by the variables π_1 and π_2, and each variable defines five transitions for a total of 10 transitions. This set of transitions is the basis for the verification of BAKERY(2).

The so-called *transition condition* for l_i (resp. m_i) is an assertion regarding the states before and after the execution of the program at l_i (resp. m_i). For instance, the transition condition for the **await** statement at l_2 is:

$$\rho_{l_2} : \pi_1 = 2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \wedge \pi_1' = 3 \wedge \pi_2' = \pi_2 \wedge y_1' = y_1 \wedge y_2' = y_2$$

where x' is the new value of x after the program execution at l_2 for any state variable x. If we use $same(\pi_2, y_1, y_2)$ for $\pi_2' = \pi_2 \wedge y_1' = y_1 \wedge y_2' = y_2$, The above formula will have a compact form:

$$\rho_{l_2} : \pi_1 = 2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \wedge \pi_1' = 3 \wedge same(\pi_2, y_1, y_2)$$

The transition conditions for l_1 and l_4 are

$$\rho_{l_1} : \pi_1 = 1 \wedge \pi_1' = 2 \wedge y_1' = y_2 + 1 \wedge same(\pi_2, y_2)$$
$$\rho_{l_4} : \pi_1 = 4 \wedge \pi_1' = 0 \wedge y_1' = 0 \wedge same(\pi_2, y_2)$$

They can be generated automatically by STeP from the program code, as pre-conditions are generated from post-conditions in Hoare logic. For the programs at l_0 and l_3, we cannot prove, for example, that if $\pi_1 = 0$, then it will eventually be $\pi_1' = 1$. Thus, we need axioms to ensure that if the *noncritical* section terminates and does not change state variables, then $\pi_1 = 0 \wedge \pi_1' = 1$ will be true for l_0. Thus the transition conditions for l_0 and l_3 are assumed as axioms:

$$\rho_{l_0} : \pi_1 = 0 \wedge \pi_1' = 1 \wedge same(\pi_2, y_1, y_2)$$
$$\rho_{l_3} : \pi_1 = 3 \wedge \pi_1' = 4 \wedge same(\pi_2, y_1, y_2)$$

The transition conditions for π_2 are defined similarly.

11.5.3 Transition Conditions in LTL

At first, the transition conditions are written in first-order logic and LTL is an extension of propositional logic. There are two approaches: either we extend LTL to first-order logic, or we restrict all the temporal operators to propositional formulas. For simplicity, we take the second approach, since all the state variables take a finite number of values and we can convert formulas to propositional formulas.

For $0 \leq i \leq 2$, let propositional variables p_i stands for $y_1 = i$ and q_i stands fro $y_2 = i$. Then, with the assumption that only one of p_i (resp. q_j) is true,

$$\begin{aligned} (y_1 = y_2 + 1) &\equiv p_1 \wedge q_0 \vee p_2 \wedge q_1 \\ (y_1 \leq y_2) &\equiv p_0 \wedge q_0 \vee p_0 \wedge q_1 \vee p_0 \wedge q_2 \vee p_1 \wedge q_1 \vee p_1 \wedge q_2 \vee p_2 \wedge q_2 \end{aligned}$$

In the following discussion, we will still use y_1 and y_2 instead of p_i and q_j for readability.

For π_1 and π_2, the labels l_i and m_j will be used as propositional variables for $\pi_1 = i$ and $\pi_2 = j$, respectively. Of course, we also need axioms to ensure that only one of l_i (resp. m_j) is true.

Secondly, given a state variable x, we use x' for the value of x after the transition. Since x' appears in the next state after the transition, we may use $\circ x$ to denote x'. For instance, $y_1' = y_1$ is written as $y_1 = \circ y_1$ and $\pi_2' = \pi_2$ as $\pi_2 = \circ\pi_2$, with the understanding that

$$\begin{array}{rcl} (y_1 = \circ y_1) & \equiv & (p_0 \leftrightarrow \circ p_0) \wedge (p_1 \leftrightarrow \circ p_1) \wedge (p_2 \leftrightarrow \circ p_2) \\ (\pi_1 = \circ\pi_1) & \equiv & (l_0 \leftrightarrow \circ l_0) \wedge (l_1 \leftrightarrow \circ l_1) \wedge (l_2 \leftrightarrow \circ l_2) \wedge (l_3 \leftrightarrow \circ l_3) \wedge (l_4 \leftrightarrow \circ l_4) \end{array}$$

All the ten transition conditions of BAKERY(2) are rewritten as follows:

$$\begin{array}{rl} \rho_{l_0}: & l_0 \wedge \circ l_1 \wedge same(\pi_2, y_1, y_2) \\ \rho_{l_1}: & l_1 \wedge \circ l_2 \wedge \circ y_1 = y_2 + 1 \wedge same(\pi_2, y_2) \\ \rho_{l_2}: & l_2 \wedge \circ l_3 \wedge (y_2 = 0 \vee y_1 \leq y_2) \wedge same(\pi_2, y_1, y_2) \\ \rho_{l_3}: & l_3 \wedge \circ l_4 \wedge same(\pi_2, y_1, y_2) \\ \rho_{l_4}: & l_4 \wedge \circ l_0 \wedge \circ y_1 = 0 \wedge same(\pi_2, y_2) \\ \rho_{m_0}: & m_0 \wedge \circ m_1 \wedge same(\pi_2, y_1, y_2) \\ \rho_{m_1}: & m_1 \wedge \circ m_2 \wedge \circ y_1 = y_2 + 1 \wedge same(\pi_2, y_2) \\ \rho_{m_2}: & m_2 \wedge \circ m_3 \wedge (y_2 = 0 \vee y_1 \leq y_2) \wedge same(\pi_2, y_1, y_2) \\ \rho_{m_3}: & m_3 \wedge \circ m_4 \wedge same(\pi_2, y_1, y_2) \\ \rho_{m_4}: & m_4 \wedge \circ m_0 \wedge \circ y_1 = 0 \wedge same(\pi_2, y_2) \end{array}$$

where $same(\pi_1, \pi_2, y_1, y_2)$ stands for $\pi_1 = \circ\pi_1 \wedge \pi_2 = \circ\pi_2 \wedge y_1 = \circ y_1 \wedge y_2 = \circ y_2$. Moreover, we will use $l_{2,3,4}$ to abbreviate the assertion $l_2 \vee l_3 \vee l_4$.

Recall that a state of BAKERY(2) is a value tuple of the state variables (π_1, π_2, y_1, y_2). Now, using l_i, m_i, p_j, and q_j, where $0 \leq i \leq 4$ and $0 \leq j \leq 3$, for the state variables, there is a one-to-one correspondence between a state and interpretation of l_i, m_i, p_j, and q_j. Hence, a state sequence is equivalent to an interpretation sequence and the truthfulness of the formulas about BAKERY(2) will be decided by state sequences.

Now it is easy to express the three desired property in temporal logic:

- *Mutual exclusion*: $\Box\neg(l_3 \wedge m_3)$, control is never at locations l_3 and m_3 at the same time.

- *Accessibility*: $\Box(l_1 \rightarrow \Diamond l_3)$ and $\Box(m_1 \rightarrow \Diamond m_3)$. If control resides at l_1 (resp. m_1), it will always eventually reach l_3 (resp. m_3).

- *One-bounded overtaking*:

$$\Box(l_2 \rightarrow \neg m_3 \, \mathcal{W} \, (m_3 \, \mathcal{W} \, (\neg m_3 \, \mathcal{W} \, l_3))).$$

  This formula states that whenever control is at l_2, meaning that $P1$ wants to enter the critical section (l_3), the following must occur: there may be an interval in which $P2$ is not in the critical section (so all states in the interval satisfy $\neg m_3$), followed by an interval where $P2$ is in the critical section (states satisfying m_3), followed by an interval where $P2$ is again not in the critical section (states satisfying $\neg m_3$), followed finally by a state where $P1$ is in the critical section (l_3). Thus, $P2$ can enter the critical section (m_3) at most once before $P1$ enters (l_3). Each of the intervals $\neg m_3$, m_3, and $\neg m_3$ that precede l_3 can possibly be empty but (as far as this formula is concerned) could extend indefinitely as well.

Mutual exclusion and one-bounded overtaking belong to the class of temporal safety properties. Informally, such properties state that something "bad" can never happen, and are falsified if a bad state is ever reached: if a safety formula A is false in a model, then there is a finite prefix of the model such that A is also false in every extension of this prefix.Mutual exclusion and one-bounded overtaking belong to the class of temporal safety properties. Informally, such properties state that something "bad" can never happen, and are falsified if a bad state is ever reached: if a safety formula A is false in a model, then there is a finite prefix of the model such that A is also false in every extension of this prefix.

Accessibility, on the other hand, is a response property, and belongs to the larger class of progress properties. These properties state that something "good" is guaranteed to happen eventually. The verification of each class of properties has its particular requirements. For instance, safety properties are independent of the termination requirements of the given program, whereas the verification of response properties relies on termination.

11.5.4 Hoare Triples and Invariants

In Chapter 10, we present the concept of Hoare triple which describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form

$$\{P\} \, C \, \{Q\}$$

where P and Q are assertions, which are formulas in a first-order language, and C is a program code. For concurrent programs considered in STeP, each line of program

C has a line label τ and there is transition condition for τ, so we can define the Hoare triple as follows:

$$\{P(x)\}\,\tau : C\,\{Q(x)\} \stackrel{\text{def}}{=} P(x) \wedge \rho_\tau(x) \rightarrow Q(\circ x)$$

where ρ_τ is the transition condition of τ, x is the list of state variables appearing in P and Q, and $\circ x$ denotes the new values of variables x after the execution of C. Since τ catches all the effect of C regarding the state variables, we may drop C in $\{P\}\,\tau : C\,\{Q\}$ and simply write $\{P\}\,\tau\,\{Q\}$, called *verification conditions*, which ensures that if C is executed in a state that satisfies P, it will reach to a state that satisfies Q. Note that this verification condition is trivially true if the transition condition of τ is false.

We will call formulas of form $\Box A$ as *invariants*, as they are always true in any state sequence. Once the BAKERY(2) program is loaded, STeP can automatically generate invariants of the following kinds:

- **Local invariants**: These invariants relate control and data with implications of the form "whenever control resides at location x then the values of the data variables are y." STeP automatically generates these local invariants for BAKERY(2):

$$\Box(l_{0,1} \rightarrow y_1 = 0)$$
$$\Box(m_{0,1} \rightarrow y_1 = 0)$$

- **Reaffirmed invariants**: These are the assertions derived as as refinement of $l_{0,1,2,3,4}$ and $m_{0,1,2,3,4}$, applying one step of transitions:

$$\Box(l_0 \wedge y_1 = 0 \vee l_1 \vee l_2 \wedge y_1 = y_2 + 1 \vee l_3 \wedge (y_1 \leq y_2 \vee y_2 = 0) \vee l_4 \vee$$
$$m_0 \wedge y_2 = 0 \vee m_1 \vee m_2 \wedge y_2 = y_1 + 1 \vee m_3 \wedge (y_2 < y_1 \vee y_1 = 0) \vee m_4)$$

- **Linear invariants**: This class of invariants captures relationships between variables that can be expressed as linear equations. For BAKERY(2), STeP's linear invariant mechanism generates the obvious control invariants, stating that control will always be at exactly one of $l_0, ..., l_4$ and $m_0, ..., m_4$:

$$\Box(l_0 + l_1 + l_2 + l_3 + l_4 = 1)$$
$$\Box(m_0 + m_1 + m_2 + m_3 + m_4 = 1)$$

  These formulas use arithmetization, where the value of a boolean expression is treated as 0 (false) or 1 (true) within an arithmetic expression.

- **Polyhedral invariants**: Finally, polyhedral invariants generalize linear invariants, capturing relationships between variables that can be expressed as linear equivalence and inequalities. The polyhedral invariant generation tools in STeP add the following two invariants for BAKERY(2):

$$\Box(l_{2,3,4} \rightarrow y_1 > 0)$$
$$\Box(m_{2,3,4} \rightarrow y_2 > 0)$$

All the above are bottom-up invariants, since they are generated from the analysis of the system independently of any temporal property to be proved. Once generated, these invariants are part of the set of background properties, which can also include axioms and previously proved properties.

11.5.5 Verification of Mutual Exclusion

The mutual exclusion of BAKERY(2) asks the valid of $\Box\neg(l_3 \wedge m_3)$. In STeP, the first choice for verifying such invariance properties is the *basic invariance rule*.

Definition 11.5.1. (Basic invariance rule) *For any assertion $P(x)$, if* (1) *$P(x)$ is true initially and* (2) *for every transition τ, $\{P(x)\}\,\tau\,\{P(\circ x)\}$ is true, then $\Box P(x)$.*

For the mutual exclusion of BAKERY(2), P is $\neg(l_3 \wedge m_3)$, STeP generates 11 verification conditions: one for each transition, plus $(y_0 = 0 \wedge y_2 = 0) \rightarrow \neg(l_3 \wedge m_3)$, the implication for the initial condition.

Example 11.5.2. Consider now the verification condition for transition l_2:

$$(1) \qquad \neg(l_3 \wedge m_3) \wedge \rho_{l_2} \rightarrow \neg(\circ l_3 \wedge \circ m_3)$$

Its negation is

$$(2) \qquad \neg(l_3 \wedge m_3) \wedge \rho_{l_2} \wedge \circ l_3 \wedge \circ m_3.$$

Recall that ρ_{l_2} is

$$\rho_{l_2} : l_2 \wedge (y_2 = 0 \vee y_1 \leq y_2) \wedge \circ l_3 \wedge same(\pi_2, y_1, y_2)$$

The deduction system will simplify (2) to

$$(3) \qquad l_2 \wedge \circ l_3 \wedge m_3 \wedge \circ m_3 \wedge (y_2 = 0 \vee y_1 \leq y_2)$$

where m_3 comes from $\circ m_3$ and $same(\pi_2, y_1, y_2)$.

From the polyhedral invariants, $\Box(l_{2,3,4} \rightarrow y_1 > 0)$ and $\Box(m_{2,3,4} \rightarrow y_2 > 0)$, we know $y_1 > 0$ and $y_2 > 0$. From the reaffirmed invariants, we obtain $l_2 \wedge (y_1 = y_2 + 1) \vee m_3 \wedge (y_2 < y_1)$. If $l_2 \wedge y_1 = y_2 + 1$ is true, then $(y_2 = 0 \vee y_1 \leq y_2)$ in (3) is false. If $m_3 \wedge (y_2 < y_1)$ is true, then $(y_2 = 0 \vee y_1 \leq y_2)$ is false, too. Hence, (3) is false and (1) must be true. □

The proof for the transition m_2 is symmetric to that of l_2. The proofs for other transitions are much easier using the linear invariants. All these verification conditions are valid relative to the previously generated invariants. Using these invariants as background properties, the STeP validity checker proves them automatically.

11.5.6 Verification of Accessibility

The accessibility means if control resides at l_1 (resp. m_1), it will always eventually reach l_3 (resp. m_3): $\Box(l_1 \rightarrow \Diamond l_3)$ and $\Box(m_1 \rightarrow \Diamond m_3)$. To prove this property, we need to know the termination of the program code. For the execution of l_2 : **await** $(y_2 = 0 \vee y_1 \leq y_2)$, once $(y_2 = 0 \vee y_1 \leq y_2)$ becomes true, l_2 changes to l_3; for the other lines of $P1$, the execution will progress to the next line, assuming the termination of **noncritical** and **critical**.

Definition 11.5.3. (Progress axiom) *The following formulas are assumed to be true for* BAKERY(2)*:*

(1) $\Box(l_2 \wedge \Diamond(y_2 = 0 \vee y_1 \leq y_2) \rightarrow \Diamond l_3)$
(2) $\Box(l_i \rightarrow \Diamond l_{i+1}$ *for* $i \in \{0, 1, 3\}$
(3) $\Box(l_4 \rightarrow \Diamond l_0)$
(4) $\Box(m_2 \wedge \Diamond(y_1 = 0 \vee y_2 < y_1) \rightarrow \Diamond m_3)$
(5) $\Box(m_i \rightarrow \Diamond m_{i+1}$ *for* $i \in \{0, 1, 3\}$
(6) $\Box(m_4 \rightarrow \Diamond m_0)$

In the definition, $(1) - (3)$ (resp. $(4) - (6)$) ensure the termination of each line in $P1$ (resp. $P2$). With the progress axioms, the proof of the accessibility property is straightforward.

Theorem 11.5.4. $\Box((l_1 \rightarrow \Diamond l_3) \wedge (m_1 \rightarrow \Diamond m_3))$.

Proof. For any state sequence σ, we show that $(l_1 \rightarrow \Diamond l_3) \wedge (m_1 \rightarrow \Diamond m_3)$ is true in σ. By the progress axiom, $(l_1 \rightarrow \Diamond l_2)$ and $(m_1 \rightarrow \Diamond m_2)$, so $\Diamond l_2$ and $\Diamond m_2$ are true in σ. Since $(y_1 \leq y_2) \vee (y_2 < y_1)$ is always true, one of $(y_1 \leq y_2)$ and $(y_2 < y_1)$ is true in any suffix σ' of σ. If $(y_1 \leq y_2)$ is true in σ', by the progress axiom (1), $\Diamond l_3$ is true in σ' as well as in σ. By the progress axiom (2), $\Diamond l_4$ is true in σ. So $y_1 := 0$ will

$$
\begin{array}{ll}
\textbf{in} & N : \textbf{integer where } N > 0 \\
\textbf{local} & y : \textbf{array } [1..N] \textbf{ of integer where } \forall i : [1..N] \,.\, y[i] = 0 \\
\textbf{value} & \max : \textbf{array } [1..N] \textbf{ of (integer} \rightarrow \textbf{integer)}
\end{array}
$$

$$
\begin{array}{c} N \\ \| \\ i = 1 \end{array} :: \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{l} \ell_0\text{: } \textbf{noncritical} \\ \ell_1\text{: } y[i] := 1 + \max(y) \\ \ell_2\text{: } \textbf{await } \forall j : [1..N].(i \neq j \;\rightarrow\; y[j] = 0 \;\vee\; y[i] \leq y[j]) \\ \ell_3\text{: } \textbf{critical} \\ \ell_4\text{: } y[i] := 0 \end{array} \right] \end{array} \right]
$$

Figure 11.5.2: Program BAKERY(N)

be executed at l_4. $y_1 = 0$ and $(y_1 = 0 \vee y_2 < y_1)$ will be true in a suffix of σ', So $\diamond m_3$ will be true σ by the program axiom (4). The case when $(y_2 < y_1)$ is true is symmetric. Since σ is arbitrary, so the theorem holds. □

Note that in the above proof, we have used the transitivity property of $\diamond$:

$$\Box(A \rightarrow \diamond B) \wedge \Box(B \rightarrow \diamond C) \rightarrow \Box(A \rightarrow \diamond C)$$

which is a valid formula of LTL.

The proof of *one-bounded overtaking* is beyond the scope of this book due to long length; the interested reader may consult the STeP survey paper by Zohar Manna et al. A general version of BAKERY(N) is given in Figure 11.5.2. All the three properties of BAKERY(2) can be carried over to BAKERY(N).

The computation of a concurrent program is an interleaving of the atomic operations of its processes. There are a huge number of execution sequences when $N > 2$ in BAKERY(N). Since a concurrent program must be correct for every possible computation, it is very difficult to test or debug programs by exhausting all interleaving sequences. Formal verification provides a feasible tool to verify the correctness of concurrent programs. We presentation is limited to LTL and there are approaches based on automaton theory and model checking, which are out of the scope of this book on logic.

11.6 Exercise Problems

Drill Questions

1. For the MTL formula $A = p \rightarrow \Box p$, find two Kripke frames such that A is true in one and false in the other.

2. For the MTL formula $A = \Box(p \vee q) \rightarrow (\Box p \vee \Box q)$, find two Kripke frames such that A is true in one and false in the other.

3. For the LTL formula $A = p \rightarrow \Box p$, find two interpretation sequences such that A is true in one and false in the other.

4. For the LTL formula $A = \Box(p \vee q) \rightarrow (\Box p \vee \Box q)$, find two interpretation sequences such that A is true in one and false in the other.

5. Find an equivalent negation formal form for the following formulas:

$$\begin{array}{ll}
(a) & (A \,\mathcal{U}\, B) \leftrightarrow \Diamond B \wedge (A \,\mathcal{W}\, B); \\
(b) & (A \,\mathcal{R}\, B) \leftrightarrow \Box B \vee (A \,\mathcal{S}\, B); \\
(c) & (A \,\mathcal{U}\, B) \leftrightarrow B \,\mathcal{S}\, (A \vee B); \\
(d) & (A \,\mathcal{R}\, B) \leftrightarrow B \,\mathcal{W}\, (A \wedge B); \\
(e) & (A \,\mathcal{W}\, B) \leftrightarrow B \,\mathcal{R}\, (A \vee B); \\
(f) & (A \,\mathcal{S}\, B) \leftrightarrow B \,\mathcal{U}\, (A \wedge B); \\
(g) & (A \,\mathcal{W}\, B) \leftrightarrow A \,\mathcal{U}\, (\Box A \vee B); \\
(h) & (A \,\mathcal{S}\, B) \leftrightarrow A \,\mathcal{R}\, (\Diamond A \wedge B); \\
(i) & \Box A \leftrightarrow A \,\mathcal{W}\, \bot; \\
(j) & \Diamond A \leftrightarrow A \,\mathcal{S}\, \top; \\
(k) & \neg(A \,\mathcal{S}\, B) \leftrightarrow \neg A \,\mathcal{W}\, \neg B.
\end{array}$$

6. A state of program BAKERY(2) is (π_1, π_2, y_1, y_2). List all the state sequences of length 10 such that in the first state, $\pi_1 = 0$ and the values of π_1 and π_2 alternate in the sequence.

Exercises

1. Prove that the following formulas are valid in MTL:

$$\begin{array}{lrcl}
(a) & \Box\Box p & \equiv & \Box p; \\
(b) & \Diamond\Diamond p & \equiv & \Diamond p.
\end{array}$$

2. Write a Prolog program `nnf`(X, Y), which takes LTL formula X as input and output Y as a negation normal form of X.

3. Prove by semantic tableau that the LTL formulas are valid:

$$\begin{array}{ll}
(a) & (\Box p \vee \Box q) \rightarrow (\Box p \vee \Box); \\
(b) & \Diamond\Diamond p \rightarrow \Diamond p; \\
(c) & \Box\Diamond\Box p \rightarrow \Diamond\Box p; \\
(d) & \Diamond\Box\Diamond p \rightarrow \Box\Diamond p.
\end{array}$$

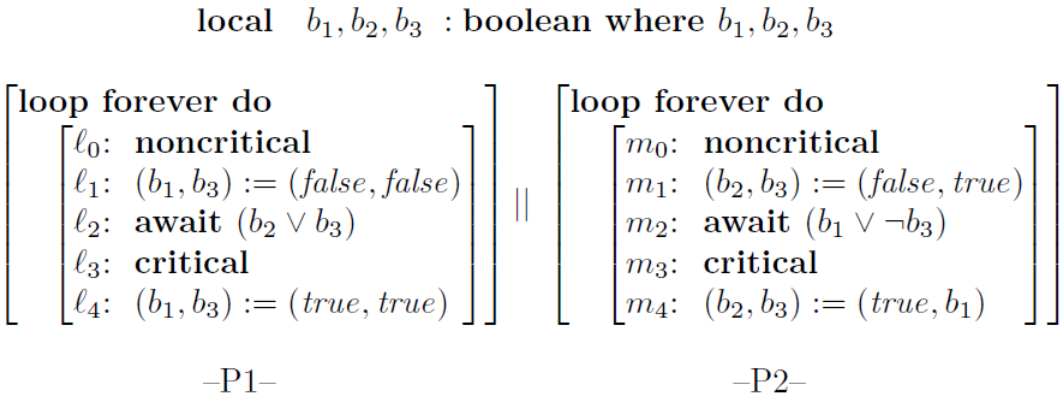


Figure 11.6.1: Program BAKERY(N)

4. Prove by semantic tableau that the LTL formulas are valid:

$$
\begin{array}{ll}
(a) & p \mathcal{R} p \rightarrow p; \\
(b) & \bot \mathcal{R} p \rightarrow \Box p; \\
(c) & p \mathcal{R} (p \mathcal{R} q) \rightarrow p \mathcal{R} q; \\
(d) & (p \mathcal{R} q) \mathcal{R} q \rightarrow p \mathcal{R} q; \\
(e) & p \mathcal{R} q \rightarrow q \wedge (p \vee \circ (p \mathcal{R} q)); \\
(f) & p \mathcal{R} q \rightarrow \neg (\neg p \mathcal{U} \neg q).
\end{array}
$$

5. Prove by semantic tableau that the LTL formulas are valid:

$$
\begin{array}{ll}
(a) & p \mathcal{U} q \rightarrow \Diamond q \wedge (p \mathcal{W} q); \\
(b) & p \mathcal{R} q \rightarrow \Box q \vee (p \mathcal{S} q); \\
(c) & p \mathcal{U} q \rightarrow q \mathcal{S} (p \vee q); \\
(d) & p \mathcal{R} q \rightarrow q \mathcal{W} (p \wedge q); \\
(e) & p \mathcal{W} q \rightarrow q \mathcal{R} (p \vee q); \\
(f) & p \mathcal{S} q \rightarrow q \mathcal{U} (p \wedge q); \\
(g) & p \mathcal{W} q \rightarrow p \mathcal{U} (\Box p \vee q); \\
(h) & p \mathcal{S} q \rightarrow p \mathcal{R} (\Diamond p \wedge q); \\
(i) & \Box p \rightarrow p \mathcal{W} \bot; \\
(j) & \Diamond p \rightarrow p \mathcal{S} \top; \\
(k) & \neg (p \mathcal{S} q) \rightarrow \neg p \mathcal{W} \neg q.
\end{array}
$$

6. Prove the mutual exclusion and accessibility properties of the program given in Figure 11.6.1.

CHAPTER 12
DECISION PROCEDURES

Recall that in Chapter 1, a *decision procedure* is defined as a procedure which always halts with a correct answer to a decision question. For propositional logic, most decision questions are decidable, such that questions regarding satisfiability, validity, or entailment. For linear temporal logic introduced in the previous chapter, these questions are also decidable. For first-order logic, these questions are undecidable in general. However, we restrict ourselves to some segments of firt-order logic, these questions may become decidable. In the literature, the selection of these segments is motivated by formal verification of programs as discussed in Chapter 10.

For ease of exposition, we consider only formulas in conjunctive normal form in each theory T that we examine. Conjunctive normal forms are conjunctions of literals. This restriction does not limit the scope of the decision procedures. For given arbitrary quantifier-free formula A, we will appeal to $DPLL(T)$, an extension of the DPLL procedure, where we only need to work on conjunctive normal forms. $DPLL(T)$ will be discussed in the last section of this chapter.

12.1 Equality with Uninterpreted Functions (EUF)

12.1.1 Uninterpreted Functions

In first-order logic, a function symbol can be interpreted as any function of the same arity. By default, every function symbol is an uninterpreted function. In some applications, it is convenient and efficient to keep in mind the definitions of some functions. For example, in program verification, $1 + 3$ can be replaced by 4 because $+$ is the addition of integers. In equational logic, the equality predicate "$=$" is assumed to satisfy the axioms of equality (Section 7.1.1). Functions like $+$ and $=$ with their intended meanings are called "interpreted functions". Thus, functions which can take any interpretation are called "uninterpreted functions".

Replacing interpreted functions by uninterpreted functions in a formula is a way of generalization (or abstraction) and can be useful in obtaining stronger results or simplifying proofs.

Example 12.1.1. We wish to prove the equivalence of the programs $P1$ and $P2$:

int $a[1..3]$	int $a[1..3]$
int $i := 2, x := a[3]$	int y
while $(i > 0)$	
$\quad x := a[i] + x; i := i - 1$	$y := a[1] + (a[2] + a[3])$
return x	**return** y
$P1$	$P2$

Since the body of the **while** loop in $P1$ executed twice, we may unroll the loop by introducing auxiliary variables x_0, x_1, x_2 to record the values of x before and after $x := a[i] + x$. The relationship between them can be expressed by

$$x_0 = a[3] \wedge x_1 = a[2] + x_0 \wedge x_2 = a[1] + x_1$$

Together with $y = a[1] + (a[2] + a[3])$, we would like to prove $x_2 = y$. Note here that x and y in the formulas are variables of the program and are regarded as identifiers in the first-order logic.

If $+$ is an interpreted function, we will have difficulty to decide the value of $a[2]+a[3]$, x_2, and y. If we replace $+$ by an uninterpreted function f, the verification becomes to prove the following theorem:

$$(x_0 = a[3] \wedge x_1 = f(a[2], x_0) \wedge x_2 = f(a[1], x_1) \wedge y = f(a[1], f(a[2], a[3]))) \rightarrow x_2 = y.$$

This can be shown to be true by rewriting, viewing each equality as a rewrite rule from left to right. Once the theorem is proved, the equivalence of $P1$ and $P2$ holds for any binary operation f on any type of $a[1..3]$ (e.g., double, or matrix).

On the other hand, if the generalized formula cannot be proved, it does not mean the equivalence of the programs does not hold. For instance, in $P2$, if we let $y := (a[1] + (a[2]) + a[3]$, then the two programs are still equivalent because $+$ is associative. However, $f(a[1], f(a[2] + a[3])) = f(f(a[1], a[2]), a[3])$ is no longer true, unless f is known to be associative. □

We would like to point out that x and y used in $P1$ and $P2$ are variables of the code; they are not *variables* of first-order logic. The formula for the equivalence of $P1$ and $P2$ is indeed ground (i.e., variable-free) because x and y are 0-arity functions in first-order logic. To avoid confusion, we call these 0-arity functions *identifiers*.

12.1.2 The Congruence Closure Problem

Recall that the axioms of equality introduced in Chapter 7 are the following five formulas, where the variables are assumed universally quantified.

reflexivity :	$x = x$;
symmetry :	$(x = y) \rightarrow (y = x)$;
transitivity :	$(x = y) \wedge (y = z) \rightarrow (x = z)$;
function congruence :	$x_i = y_i \rightarrow f(..., x_i, ...) = f(..., y_i, ...)$;
predicate congruence :	$x_i = y_i \wedge p(..., x_i, ...) \rightarrow p(..., y_i, ...)$.

A set A of literals is called *ground equality* if the only predicate in A is "=" and no literals contain variables. We often write $\neg(s = t)$ as $s \neq t$. A positive literal $s = t$ is called *equation* and a negative literal $s \neq t$ is called *unequality*. If constants c_1 and c_2 appear in A, we assume that $(c_1 \neq c_2) \in A$, so that we do not distinguish identifiers (i.e., functions of zero arity) and constants. If predicates other than $=$ are needed, we treat them as functions: replace positive literal $p(x)$ by $p(x) = \mathit{true}$ and negative literal $\neg p(x)$ by $p(x) = \mathit{false}$, and add $\mathit{true} \neq \mathit{false}$ into A.

Given a set A of ground equality, is A satisfiable with the axioms of equality? This decision problem is traditionally called *the congruence closure problem of equality with uninterpreted functions*. For instance, in Example 12.1.1, let A be

$$\{x_0 = a[3], x_1 = f(a[2], x_0), x_2 = f(a[1], x_1), y = f(a[1], f(a[2], a[3])), \neg(x_2 = y)\},$$

then A is unsatisfiable iff $x_2 = y$ is a logical consequence of the first four equations.

Since a function is uninterpreted by default in first-order logic, we prefer to call it *the congruence closure problem of ground equality*. Note that $=$ is the only interpreted function and the axioms of equality contain variables.

Some people also called it *the congruence closure problem of quantifier-free equality*. In the literature, the quantifier-free fragment of a theory T consists of the axioms of T and valid formulas of the form

$$\forall x_1, ..., x_n \, F(x_1, ..., x_n),$$

or of form

$$\exists x_1, ..., x_n \, F(x_1, ..., x_n),$$

where F is quantifier-free and $free(F) = \{x_1, ..., x_n\}$ and only one of the two forms is allowed in any given input. While such formulas have quantifiers, the point is that they do not have quantifier alternations: either all quantifiers are universal or all quantifiers are existential. For instance, clauses in resolution contain universally quantified variables and terms in unification contain existential variables. For the

congruence closure problem to be discussed, no variables are allowed, with the only exception of the variables appearing in the equality axioms. Thus, "quantifier-free" is not a right label for the congruence closure problem.

The conventional approach, proposed by Nelson and Oppen, as well as Shastak, consists of the following steps:

1. Divide a set A of ground equality into equations E (positive literals) and unequalities D (negative literals);

2. Compute the congruence closure $=_E$, which is the congruence closure generated by E (Definition 7.1.6). It is the minimum equivalence relation satisfying the congruence property.

3. If there exists $(s \neq t) \in D$ such that $s =_E t$, then return "unsatisfiable"; otherwise return "satisfiable".

The soundness of the above approach is stated as the following theorem:

Theorem 12.1.2. *Given a set A of ground equality, including both equations and unequalities, let $E \subseteq A$ be the set of all equations in A. If there exists $(s \neq t) \in A$ such that $s =_E t$, then A is unsatisfiable, otherwise A is satisfiable.*

The proof can be done by Herbrand models (Proposition 7.1.3) and is left as an exercise. Now we ready for the proof of the following theorem:

Theorem 12.1.3. *The congruence closure of ground equality is decidable.*

Proof. Given a set A of ground equality, including both equations and unequalities, let $E \subseteq A$ be the set of all equations in A. According to Theorem 7.2.22, there is a decision procedure for $=_E$. For each $s \neq t \in A$, we check if $s =_E t$: if yes, return "unsatisfiable". If every check says "no", return "satisfiable". Theorem 12.1.2 ensures the correctness of the answer. □

Example 7.2.21 shows how $A = \{f(a) \neq a, f^3a = a, f^5a = a\}$, where f^3a is $f(f(f(a)))$ and f^5a is $f(f(f(f(f(a)))))$, is unsatisfiable.

12.1.3 Nelson and Oppen's Algorithm

Congruence is an equivalence relation satisfying the congruence axiom. To store an equivalence relation in computer, the best data structure is *sets*: Equivalent elements are stored in the same set so that reflexive, symmetric, and transitive properties are easily maintained. Each set represents an equivalent class and the collection of equivalent classes is a partition of all the elements.

The Nelson and Oppen's algorithm is based on this idea: To compute $=_E$ for a set E of ground equations,

1. Place each subterm of E into its own congruence class;
2. For each $s = t \in E$, if s and t are not in the same class, the *merge* operation will (i) union the two classes containing s and t; and (ii) propagate $s = t$ by the congruence axiom.

The propagation of $s = t$ through the congruence axiom is done as follows: If s appears in $f(..., s, ...)$ and t appears in $f(..., t, ...)$, then the NelsonOppen algorithm will ensure that $f(..., s, ...)$ and $f(..., t, ...)$ should be in the same class by checking if their arguments are pair-wisely equivalent.

The NelsonOppen algorithm uses term graphs and the Union-Find data structure for sets, in a way very similar to the unification algorithm (Alg. 6.1.19).

The algorithm *NelsonOppen*(E) takes a set E of ground equations as input, create the term graph $G = (V, E)$ from all the terms in E, sharing subterms as much as possible. If a set D of unequalities is provided, the terms in D are also added into G. $G = (V, E)$ has the following properties:

1. G is a directed acyclic graph (DAG) with possibly multiple edges between two nodes;
2. Each node $v \in V$ has a label, denoted by *label*(v), which is a function symbol of E;
3. If *label*$(v) = f$ and *arity*$(f) = k$, then v has k ordered successors in G. The i^{th} successor can be accessed by *child*(v, i). Obviously, if f is an identifier (*arity*$(f) = 0$), v has no successors.
4. Each subterm t of E is represented by a unique node $v_t \in V$ such that (a) t is an identifier and *label*$(v_t) = t$; or (b) if $t = f(t_1, ..., t_k)$, then *label*$(v_t) = f$ and v_t has k successors $v_{t_1}, ..., v_{t_k}$, each v_{t_i} represents t_i for $1 \leq i \leq k$.
5. To implement the Union-Find algorithm, we add two data structures to each node $v \in V$:
   - *up*(v) defines the parent of v in a tree, which also represents an equivalence relation among the nodes. Initially, *up*$(v) = v$. That is, v is a root and each node is initially in its own equivalence class.
   - $w(v)$ records the *weight* of the tree representing the class of v, which is the size of the class. Initially, $w(v) = 1$, that is, the tree is a single node.

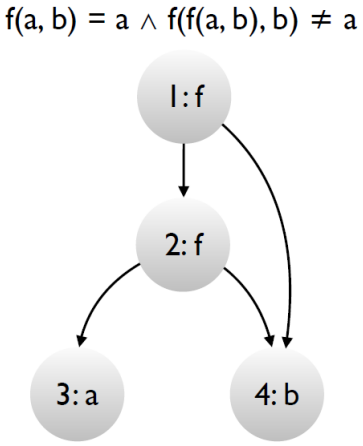


Figure 12.1.1: DAG $G = (V, E)$ for $A = \{f(a, b) = a, f(f(a, b), b) \neq a\}$, where $V = \{1, 2, 3, 4\}$, $use(1) = \emptyset, use(2) = \{1\}$, $use(3) = \{2\}$, $use(4) = \{1, 2\}$

- To implement the congruence propagation, we add a list to each node $v \in V$: $use(v)$ records the list of all predecessors (also called parents) of v in G; this should not be confused with $up(v)$, which is the parent of v in the tree.

Example 12.1.4. Let $A = \{f(a, b) = a, f(f(a, b), b) \neq a\}$. The DAG for the terms of A is shown in Figure 12.1.1. □

Once DAG G is created, for each equation $s = t$, we will call $merge(v_s, v_t)$, where v_s and v_t are the nodes of G representing s and t.

```
proc merge(v1, v2)
1     s1 := Find(v1); s2 := Find(v2)
2     if s1 = s2 return // already in the same class, exit
3     P1 := use(s1); P2 := use(s2)
4     Union(s1, s2) // Union two classes into one
5     for (t1, t2) ∈ P1 × P2 do // Congruence propagation
6         if Find(t1) ≠ Find(t2) ∧ congruent(t1, t2)
7             merge(t1, t2)
```

The pseudo-code for *congruent*, *Union* and *Find* are given below.

proc *congruent*(t_1, t_2) : *Boolean*

```
1    if label(t_1) ≠ label(t_2) return false
2    for i := 1 to arity(label(t_1)) do
3        if Find(child(t_1, i)) ≠ Find(child(t_2, i)) return false
4    return true

proc Union(s_1, s_2)
     // Initially, for any node s, w(s) = 1, the weight of a singleton tree.
1    if w(s_1) < w(s_2) x := s_1; s_1 := s_2; s_2 := x // switch s_1 and s_2
2    up(s_2) := s_1 // s_2's representative is s_1
3    use(s_1) := use(s_1) ∪ use(s_2); use(s_2) = ∅
4    w(s_1) := w(s_1) + w(s_2)

proc Find(v)
1    r := up(v) // look for the root of the tree containing v
2    while (r ≠ up(r)) r := up(r) // r is a root iff r = up(r)
3    if (r ≠ v) // compress the path from v to r
4       while (r ≠ v) up(v) := r; v := up(v)
5    return r
```

Algorithm 12.1.5. The algorithm *NelsonOppen*(A) takes a set A of ground equality, which is a mix of equations or unequalities, as input, create the term graph $G = (V, E)$ from all the terms in A, construct the congruence classes for equations in A and check the satisfiability of A.

```
proc NelsonOppen(A) : Boolean
1    G = createGraph(A) // up(v) = v for each v in G
2    for (s = t) ∈ A do merge(v_s, v_t) // v_s, v_t represent s, t in G
3    for (s ≠ t) ∈ A do
4        if Find(v_s) = Find(_t) return "unsatisfiable"
5    return "satisfiable"
```

Example 12.1.6. Continue from Example 12.1.4: To handle $f(a, b) = a$, *merge*$(2, 3)$ is called. The first line of *merge* calls *Find* twice with the result $s_1 = Find(2) = 2$ and $s_2 = Find(3) = 3$. Since $s_1 \neq s_2$, we get $P_1 = \{1\}$ and $P_2 = \{2\}$, preparing for congruence propagation.

At line 4 of *merge*, *Union*$(2, 3)$ is called. Because $w(2) = w(3) = 1$, the second part of line 1 is skipped. At line 2, $up(3)$ is changed to 2, meaning node 3 uses node 2 as its class name. At line 3, $w(2)$ is changed from 1 to 2, as the class tree increases its weight. At line 4, the parents of node 3 move to those of node 2.

Now back to line 5 of *merge*: Since $P_1 \times P_2 = \{\langle 1, 2\rangle\}$, $t_1 = 1$ and $t_2 = 2$ are the only case for t_1 and t_2. Since $Find(1) = 1$ and $Find(2) = 2$, $congruent(1, 2)$ is called and returns true, because $Find(child(1, 1)) = Find(2) = 2 = Find(3) = Find(child(2, 1))$ and $Find(child(1, 2)) = Find(child(2, 2)) = 4$. Now $merge(1, 2)$ is recursively called at line 7 and it will merge node 1 into $\{2, 3\}$ (i.e., $up(1) = 2$). The final congruence classes are $\{1, 2, 3\}$ and $\{4\}$.

For the unequality $f(f(a, b), b) \neq a$ of A, since $f(f(a, b), b)$ (represented by node 1) and a (node 3) are in the same class, $f(f(a, b), b) \neq a$ is false and A is unsatisfiable. □

Nelson and Oppen proved that the time complexity of *NelsonOppen*(A) is $O(n^2)$, where n is the number of symbols of A. Various improvements and extensions of *NelsonOppen*(A) have been proposed, including Shastak's algorithm. Shastak's algorithm improves Nelson and Oppen's algorithm in three ways:

1. At line 7 of *merge*, $merge(t_1, t_2)$ is replaced by $merge(Find(t_1), Find(t_2))$.

2. Introduce a data structure $sig(t)$ for each term t such that $sig(f(t_1, ..., t_n)) = f(Find(t_1), ..., Find(t_u))$. This data structure will reduce substantially the time spent at lines 5-6 of *merge*.

3. Allow the handing of equations in an "on-the-fly" manner so that equations can be added dynamically. This optimization widens the application of congruence closure algorithms.

Instead of presenting Shastak's algorithm in details, we will take Kapur's approach and present Shastak's algorithm in the context of the Knuth-Bendix completion procedure.

12.1.4 Ground Congruence by Knuth-Bendix Completion

According to Theorem 12.1.3, the Knuth-Bendix completion procedure for ground equations can be used as a decision procedure for the ground congruence problem. To make the completion procedure efficient, Kapur suggested two preprocessing steps which make equations or rewrite rules "flat" and "singleton".

Definition 12.1.7. *A term t is said to be* flat *if either t is an identifier or $t = f(a_1, ..., a_k)$ and $a_1, ..., a_k$ are identifiers. An equation $s = t$ (or a rewrite rule $s \to t$) is* flat *if both s and t are flat. $s = t$ (or $s \to t$) is* singleton *if t is an identifier. $s = t$ (or $s \to t$) is* unit *if both s and t are identifiers.*

The two preprocessing steps over E of ground equations will flatten terms if the terms in an equation are not flat and slicing an equation if no identifier appears either side of the equation. Flattening and slicing are done by introducing new identifiers, so that the generated rewrite rules will be flat and singleton. This is achieved by the following operations:

- **flattening**: If t in E is not flat, then t is $f(..., g(s), ...)$ and we replace t by $f(..., c, ...)$ and add $g(s) = c$ into E, where c is a new identifier.
- **slicing**: If $(s = t) \in E$, neither s nor t is an identifier, then we replace $s = t$ by $s = c$ and $t = c$ in E, where c is a new identifier.
- **switching**: If $(s = t) \in E$, s is an identifier and t is not, then we replace $s = t$ by $t = s$ in E.

Let $E_1 = \mathit{flattenSlicing}(E_0)$ denote the procedure which applies the above operations to E_0 until E_1 is flat and singleton.

Proposition 12.1.8. *Assuming the arity of any function symbol is a constant. If* $E_1 = \text{flattenSlicing}(E_0)$*, then* $n = O(|E_1|)$*, where* n *is the total of function symbols (including identifiers) appearing in* E_0*.*

Proof. Let n_1 be the total of function symbols (including identifiers) appearing in E_1. Then $n \leq n_1 \leq 3n$, because for the **flattening** rule, two copies of a new constant are added into E for each non-identifier subterm position other than the root position. Similarly, for the **slicing** rule, a new identifier is introduced for the root position of a non-identifier term. In other words, the total number of new identifier occurrences introduced by $\mathit{flattenSlicing}(E_0)$ cannot be more than twice of the total number of subterm positions in E_0.

Since E_1 is flat and singleton, if k is the arity of the root symbol appearing in the left side of an equation, then this equation has $k+2$ symbols. By the assumption, k is a constant, so $n_1 = O(|E_1|)$. Thus $n = O(|E_1|)$ because $n \leq n_1 \leq 3n$. □

Example 12.1.9. Let $E_0 = \{f(a, g(a)) = g(g(a))\}$ be the input to the modified Knuth-Bendix completion procedure. Then $E_1 = \mathit{flattenSlicing}(E_0)$, where E_1 contains the following equations:

$$\{g(a) = c_1, f(a, c_1) = c_3, g(a) = c_2, g(c_2) = c3\}$$

where new identifiers c_1 and c_2 are introduced by flattening, and c_3 by slicing. If we feed E_1 to the Knuth-Bendix completion procedure, the procedure ends with the following rewrite rules:

$$(1)\ g(a) \to c_1, \quad (2)\ f(a, c_1) \to c_3, \quad (3)\ g(c_1) \to c_3, \quad (4)\ c_2 \to c_1$$

This rewrite system is canonical (confluent and terminating), flat and singleton. It can serve as a decision procedure for $=_{E_0}$. □

Any rule in U whose left side is an identifier introduced by flattening and slicing can be removed, because this identifier occurs nowhere other than in this rule; thus there is no chance to use this rule. Thus, for the above example, the last rule, i.e., $c_2 \to c_1$ can be deleted.

Example 12.1.10. Let $E_0 = \{f^3a = a, f^5a = a\}$. Then $E_1 = \mathit{flattenSlicing}(E_0)$, where E_1 contains the following equations:

$$\{f(a) = c_1, f(c_1) = c_2, f(c_2) = a, f(c_2) = c_3, f(c_3) = c_4, f(c_4) = a\}$$

The first three equations come from $f^3a = a$; all the equation excluding $f(c_2) = a$ come from $f^5a = a$, as we reuse the first two equations.

Feeding E_1 to the Knuth-Bendix procedure, the first three equations will produce three rewrite rules:

$$\{(1)\ f(a) \to c_1,\ \ (2)\ f(c_1) \to c_2,\ \ (3)\ f(c_2) \to a\}$$

The fourth equation, $f(c_2) = c_3$, is rewritten to $a = c_3$, from which we obtain (4) $c_3 \to a$.

The fifth equation, $f(c_3) = c_4$, is rewritten by (4) and (1) to $c_1 = c_4$, from which we obtain (5) $c_4 \to c_1$.

The last equation, $f(c_4) = a$, is rewritten by (5) and (2) to $c_2 = a$, from which we obtain (6) $c_2 \to a$. (2) becomes $f(c_1) \to a$ by (6).

(6) reduces (3) to $f(a) = a$, then by (1) to $c_1 = a$, from which we obtain (7) $c_1 \to a$. (1) becomes $f(a) \to a$ by (7). (7) also reduces (2) to $f(a) = a$, then by (1) to $a = a$.

The final rewrite system is

$$\{(1)\ f(a) \to a,\ \ (4)\ c_3 \to a,\ \ (5)\ c_4 \to a,\ \ (6)\ c_2 \to a,\ \ (7)\ c_1 \to a\}$$

If we remove those rules in U whose left side is a new identifier, then only the first rule, i.e., $f(a) \to a$, is needed for the decision procedure of $=_{E0}$. □

During the execution of the Knuth-Bendix completion procedure (Proc. 7.2.15), let E be the set of remaining equations and R be the current set of rewrite rules, both E and R are ground, flat and singleton. R can be partitioned into U and N, i.e., $R = U \cup N$, where U are unit rules and N are non-unit rules. The above two examples illustrate some properties of (E, R):

1. If $R = U \cup N$ is inter-reduced (Def. 7.2.16), and $u \to v \in U$, i.e., it is a unit rule, then $u \to v$ will never be deleted by any other rule $l \to r$, because if $l \to r$ is older than $u \to v$, then R is not inter-reduced; if $l \to r$ is newer, see the next point.

2. When $s = t \in E$ is made into a unit rule $l \to r$, $\{l, r\}$ are normal forms of $\{s, t\}$ by R. This unit rule can rewrite some rules of N, but cannot rewrite the left side of any rule of U. If $l \to r$ could rewrite u, where $u \to v \in U$, then $l = u$ and l cannot a normal form of U. In other words, all rules of U will stay forever.

3. When $s = t \in E$ is made into a non-unit rule $s' \to t'$, s' and t' are normal forms of s and t by R, respectively. This non-unit rule cannot rewrite any rule of R, because if $s' \to t'$ can rewrite $l \to r \in R$, then it must be the case that $s' = l$ and this is a contradiction to the fact that s' is a normal form of R. Later, $s' \to t'$ may be removed from R and added into E if one of the arguments of s' is writable by another rule.

Based on the above properties and borrowing some ideas from Kapur's algorithm as well as Nieuwenhuis and Oliveras' algorithm, we modify the Knuth-Bendix procedure by using the following three data structures for each identifier c:

- $nf(c)$: the normal form of c by U; initially, $nf(c) = c$.
- $class(c)$: the set of all identifiers b such that $nf(b) = c$; initially, $class(c) = \{c\}$.
- $use(c)$: the set of rules in N which uses c in its left-hand side; initially $use(c) = \emptyset$.

Procedure 12.1.11. The procedure $KBG(E, \succ)$ takes a set E of ground, flat and singleton equations and a total order $\succ$ where $f \succ c$ for any function symbol f and identifier c, and generates a canonical rewrite system from E.

```
proc KBG(E)
1    U := N := ∅; initialize nf(), class(), and use()
2    while (E ≠ ∅) do
3       (s = t) := pickEquation(E); E := E − {(s = t)}
4       s := NF(s); t := NF(t) //normalize s and t by R = U ∪ N
5       if (s = t) continue // identity is discarded
6       else if identifier?(s) // if s is also an identifier
```

```
7          if |class(t) < |class(s)| x := s; s := t; t := x // switch s and t
8          U := U ∪ {s → t} // s → t is a unit rule
9          for c ∈ class(s) do nf(c) := t
10         class(t) := class(t) ∪ class(s); class(s) := ∅
11         for (l → r) ∈ use(s) do // inter-reduction
12             N := N − {l → r}; E := E ∪ {l = r}
13       else // s → t is non-unit
14         N := N ∪ {s → t} // suppose s = f(a_1, ..., a_k)
15         for i := 1 to k do use(a_i) := use(a_i) ∪ {s → t}
16     return U ∪ N

proc NF(t)
17     if identifier?(t) return nf(t)
18     else assume t = f(t_1, ..., t_k)
19       s := "f(NF(a_1), ..., NF(a_k))"
19       if s is flat and (s → b) ∈ N return nf(b)
20       else return s
```

Like the Knuth-Bendix completion procedure (Proc. 7.2.15), the above procedure takes one equation from E at each iteration of the main loop, normalizes the equation by $R = U \cup N$, and tries to make a rewrite rule from the normalized equation and add the rule into R, while keeping R inter-reduced. Some rules of R go back to E as the result of inter-reduction. The main loop terminates when E becomes empty. However, *KBG* has special features which make it efficient than the general completion procedure:

- Critical pair computation is omitted because inter-reduction is sufficient for ground equations.
- The normalization procedure $NF(t)$ uses $nf(c)$ instead of U for each identifier c; the rules of N can be used at most once because t is flat. In fact, $NF(t)$ can be done in $O(|t|)$ time since we can use a hash table or a trie structure to store N.
- When a unit rule $s \to t$ is created, we update $nf(c)$ by t for any $c \in class(s)$ (line 9) and merge $class(s)$ into $class(t)$ (line 10). Recall that $nf(c)$ is the normal form of c by U. Due to the existence of $nf(c)$, rewrite rules in U are superficial and never deleted nor simplified.

- The data structure $use(s)$ records the set of non-unit rules which use identifier s in its left-side (line 14). This data structure is useful at line 9 to remove rules $l \to r$ from N when l is rewritable by a new rule $s \to t$.
- When a non-unit rule is added into N (line 14), N as well as $R = U \cup N$ remains inter-reduced, so there is no need to rewrite N by the new rule.

The correctness of the above procedure comes directly from Theorem 7.2.22.

Theorem 12.1.12. *If E is a set of ground, flat and singleton equations, and n is the total number of symbols in E, then the procedure* KBG *runs in $O(n\ log(n))$.*

Proof. The main loop (lines 3–15) handles $s = t$ of E and each line of the main loop takes various times to execute. Line 4 takes $O(|s| + |t|)$ or $O(1)$ time if we consider the arity of any symbol is a constant. Line 9 takes $O(|class(s)|)$ time, lines 11-12 takes $O(|use(s)|)$ time. All other lines take constant time. If each equation of E is processed only once, then the total time would be $O(n)$, because the sums of $|s| + |t|$, $|class(s)|$ or $|use(s)|$ for all equations $s = t$ and identifies c are bound by $O(n)$. If no equations are processed more than m times, then the total time of *KBG* would be $O(nm)$.

How many times an equation can go back from N to E (line 12)? Note that the root cause of moving an equation from N to E at line 12 is that we have a new unit rule $s \to t$ (line 8) and s appears in the moved equations. Usining Nieuwenhuis and Oliveras' idea, when making a rewrite rule $s \to t$ from $s = t$, we ensure that $|class(s)| \leq |class(t)|$ (line 7), and then merge $class(s)$ into $class(t)$ (line 10). In other words, an equation moves from N to E only when (the left-side of) this equation contains an identifier whose class will be merged into another class. Since a smaller class cannot be merged into a larger class more than $log(n)$ times, no equation can be moved from N to E more than $log(n)$ times. So the total complexity of *KBG* is bound by $O(n\ log(n))$. □

Theorem 12.1.13. *The congruence closure problem for ground equality can be decided in $O(n\ log(n))$, where n is the input size, i.e., the total number of symbols appearing in the input.*

Proof. Given a set A of *ground equality*, let $E \subseteq A$ be all the equations of A. At first, we call *KBG*(*flattenSlicing*(E)) to obtain a decision procedure R for $=_E$. Next, for each unequality $(s \neq t) \in A$, we check if $NF(s) = NF(t)$ using R; if yes, return "unsatisfiable". The whole process takes $O(n\ log(n))$ time (Theorem 12.1.12) as $NF(s)$ and $NF(t)$ take $O(|s| + |t|)$ time. □

Kapur's idea of flattening and slicing has been used sucessfully to handle function symbols which are commutative and associative. Nieuwenhuis and Oliveras' algorithm can also produce automated proofs of $s =_E t$.

12.2 Linear Arithmetic

Linear arithmetic is a theory of first-order logic where formulas contain variables over a domain D, where D can be of type rational or integer, and usual arithmetic operators such as $+$, $-$, $\cdot$, $=$, $\leq$, $\geq$, etc., in addition to the conventional Boolean operators. The formulas are *linear* as the multiplication, i.e., $\cdot$, is not allowed over two variables. An atomic linear arithmetic formula looks like

$$a_1 \cdot x_1 + \cdots + a_n \cdot x_n \leq b, \quad \text{or} \quad a_1 x_1 + \cdots + a_n x_n \leq b,$$

where $\leq$ can be replaced by other relation operators, variables $x_1, ..., x_n$ are of type D, and constants $a_1, ..., a_n, b \in D$. Complex linear arithmetic formulas can be obtained from atomic formulas by Boolean operators as in first-order logic. The theory of linear arithmetic also defines the interpretation of linear arithmetic formulas: all arithmetic functions, predicates, and constants are interpreted by their well-known mathematical denitions and all Boolean connectives are interpreted as we are used to from rst-order logic.

Linear arithmetic is very expressive as numerous problems in scientific and industrial computation can be expressed in linear arithmetic. Linear arithmetic is relevant to automated reasoning because most programs use arithmetic variables (e.g., integers) and perform arithmetic operations on those variables. Therefore, software verication has to deal with arithmetic. Linear arithmetic has been thoroughly investigated in at least two directions: (i) optimization via linear programming (LP), integer programming (ILP), and mixed real integer programming (MILP). (ii) first-order quantifier elimination. In this chapter, we will focus on (i) only. In particular, we are interested in the satisability of linear arithmetic constraint problems in the context of the combination of theories, as they occur, e.g., in SMT (satisability modulo theories) solving. .

12.2.1 Simplex Method by Example

Dantzig's simplex method (or simplex algorithm) is a popular method for linear equational systems. Each equational system defines an optimization problem over the domain of reals by an objective function and a list of linear arithmetic equations. The simplex algorithm is regarded by many as one of the 10 algorithms with

the greatest influence on the development and practice of science and engineering in the 20th century.

Let us consider the following example of a linear equational system:

$$\begin{array}{lrcl} \text{Maximize} & 4x_1 + 3x_2 & & \\ \text{subject to} & & & \\ & x_1 + 2x_2 + x_3 & = & 16 \\ & x_1 + x_2 + x_4 & = & 9 \\ & 3x_1 + 2x_2 + x_5 & = & 24 \\ & x_1, x_2, x_3, x_4, x_5 & \geq & 0 \end{array} \tag{12.1}$$

For the objective function $x_0 = 4x_1 + 3x_2$, we may introduce a new variable x_0 and write it in standard form as $x_0 - 4x_1 - 3x_2 = 0$, then (12.1) becomes (12.2):

$$\begin{array}{lrrrrrrr} \text{Maximize} & x_0 & \text{where} & & & & & \\ (a) & x_0 & -4x_1 & -3x_2 & & & = & 0 \\ (b) & & x_1 & +2x_2 & +x_3 & & = & 16 \\ (c) & & x_1 & +x_2 & +x_4 & & = & 9 \\ (d) & & 3x_1 & +2x_2 & & +x_5 & = & 24 \\ & & x_1, & x_2, & x_3, & x_4, \quad x_5 & \geq & 0 \end{array} \tag{12.2}$$

Note that (12.2) has a simple matrix representation, which is used for the implementation of the simplex method.

x_0	x_1	x_2	x_3	x_4	x_5	rhs
1	−4	−3	0	0	0	0
0	1	2	1	0	0	16
0	1	1	0	1	0	9
0	3	2	0	0	1	24

Gien a linear equational system, a *basic variable* is a variable which appears only once in the system. In its standard form, the coefficient of a basic variable is 1. For the present example, x_0, x_3, x_4, and x_5 are basic variables; x_1 and x_2 are non-basic variables. It is easy to spot the basic variables in the matrix where the column corresponding to the variable has only one non-zero entry. The objective function x_0 is always basic. In a linear equational system, if the number of variables is more than the number of equations, it is easy to locate a basic variable for each equation by arithmetic operations.

In the simplex method, a basic variable and a non-basic variable can exchange their status if this exchange increases the objective function value. The simplex method repeats this kind of exchanges until an optimal value is obtained.

In the present example, x_0 is a function of x_1, x_2, x_3, x_4, x_5. The value of $(x_1, x_2, x_3, x_4, x_5)$ defines a point in the search space. The points considered by the

simplex method are those points where non-basic variables are zero and the values of the basic variables are decided by the equations. In the present example, we set non-basic variables $x_1 = x_2 = 0$. From the four equations of (12.2), we have $x_0 = 0$, $x_3 = 16$, $x_4 = 9$, and $x_5 = 24$. So the starting point considered by the simplex method is $(0, 0, 16, 9, 24)$.

The simplex method searches an optimal solution by repeatedly exchanging a basic variable and a non-basic variable. Next, let us see how to choose basic and non-basic variables for exchange.

From $(12.2)(a)$, $x_0 = 4x_1 + 3x_2$. In order to increase the value of x_0, we need to increase x_1 or x_2, one variable at a time. If x_1 increased by 1, x_0 gains 4. If x_2 is increased by 1, x_0 gains 3. Obviously, x_1 is preferred for the faster increment of x_0. Hence, x_1 is chosen for exchange.

Once the non-basic variable is chosen, what basic variable, x_3, x_4, or x_5, should we replace? To answer this question, consider $(12.2)(b) - (d)$ in the situation of $x_2 = 0$:

$$\begin{array}{rcl@{\quad}l@{\quad}rcl} x_1 + x_3 & = & 16 & & x_3 & = & 16 - x_1 \geq 0 \\ x_1 + x_4 & = & 9 & \text{which give us} & x_4 & = & 9 - x_1 \geq 0 \\ 3x_1 + x_5 & = & 24 & & x_5 & = & 24 - 3x_1 \geq 0 \end{array}$$

because $x_3, x_4, x_5 \geq 0$. From $16 - x_1 \geq 0$, $x_1 \leq 16$; from $9 - x_1 \geq 0$, $x_1 \leq 9$; from $24 - 3x_1 \geq 0$, $x_1 \leq 8$. Since x_1 must satisfy all the three inequalities, we choose $x_1 = min\{16, 9, 8\} = 8$. Since this value comes from $x_5 = 24 - 3x_1 \geq 0$, x_5 is chosen for becoming non-basic. The equation $3x_1 + 2x_2 + x_5 = 24$ is called the *pivot equation* and x_5 is called the *pivot variable*; the process of choosing a pivot equation is called *pivoting*.

From $x_1 = 8$ and $x_1 + x_3 = 16$, x_3 has a new value 8. From $x_1 + x_4 = 9$, $x_4 = 1$. So the next search point consider by the simplex method is $(8, 0, 8, 9, 0)$, where x_2 and x_5 are non-basic.

Now, we need to normalize (12.2) so that basic variables appear only once and x_0 is expressed as a function of non-basic variables. This can be done as follows: from the pivot equation $((12.2)(c))$, we have

$$x_1 + (2/3)x_2 + (1/3)x_5 = 8 \qquad (12.3)$$

Adding (12.3) 4 times to $(12.2)(a)$, we obtain

$$x_0 - (1/3)x_2 + (4/3)x_5 = 32$$

Subtracting (12.3) from $(12.2)(b)$, we obtain

$$(4/3)x_2 + x_3 - (1/3)x_5 = 8$$

Subtracting (12.3) from (12.2)(c), we obtain

$$(1/3)x_2 + x_4 - (1/3)x_5 = 1$$

Putting these equations together, we obtain a new equational system in standard form:

$$\begin{array}{lllllllllll} \text{Maximize} & x_0 & \text{where} & & & & & & & & \\ (a) & x_0 & & -(1/3)x_2 & & & & +(4/3)x_5 & = & 32 \\ (b) & & & (4/3)x_2 & +x_3 & & & -(1/3)x_5 & = & 8 \\ (c) & & & (1/3)x_2 & & +x_4 & & -(1/3)x_5 & = & 1 \\ (d) & & x_1 & +(2/3)x_2 & & & & +(1/3)x_5 & = & 8 \\ & & x_1, & & x_2, & x_3, & x_4, & & x_5 & \geq & 0 \end{array} \tag{12.4}$$

The corresponding matrix is given below:

x_0	x_1	x_2	x_3	x_4	x_5	rhs
1	0	$-1/3$	0	0	$4/3$	32
0	0	$4/3$	1	0	$-1/3$	8
0	0	$1/3$	0	1	$-1/3$	1
0	1	$2/3$	0	0	$1/3$	8

At the point $(8, 0, 8, 9, 0)$, the value of x_0 is 32. Is this the optimal value for x_0? From (12.4)(a), we have

$$x_0 = (1/3)x_2 - (4/3)x_5 + 32$$

Now $x_0 = 32$ when $x_2 = x_5 = 0$. If x_2 is increased by 1, x_0 gains 1/3. That is, there is a space for increment of x_0 by choosing x_2 as a new basic variable.

The process of selecting an existing basic variable for exchange is the same as before: Finding the pivot variable in a pivot equation. Assuming $x_5 = 0$, from (12.4)$(b) - (d)$, we have

$$\begin{array}{lcll} (4/3)x_2 + x_3 = 8 & & & x_3 = 8 - (4/3)x_2 \geq 0 \\ (1/3)x_2 + x_4 = 1 & \text{which give us} & & x_4 = 1 - (1/3)x_2 \geq 0 \\ x_1 + (2/3)x_2 = 8 & & & x_1 = 8 - (2/3)x_2 \geq 0 \end{array}$$

From $8 - (4/3)x_2 \geq 0$, $x_2 \leq 6$; from $1 - (1/3)x_2 \geq 0$, $x_2 \leq 3$; from $8 - (2/3)x_2 \geq 0$, $x_2 \leq 12$. Hence, we choose $x_2 = 3$ to satisfy all the three inequalities, and the pivot equation is $(1/3)x_2 + x_4 - (1/3)x_5 = 1$ ((12.4)(c)). Now, multiply (12.4)(c) by 3, we obtain

$$x_2 + 3x_4 - x_5 = 3 \tag{12.5}$$

Multiply (12.5) by 1/3 and add the result to (12.4)(a), we have

$$x_0 + x_4 + x_5 = 33 \qquad (12.6)$$

Multiply (12.5) by $-4/3$ and add the result to (12.4)(b), we have

$$x_3 - 4x_4 + x_5 = 4$$

Multiply (12.5) by $-2/3$ and add the result to (12.4)(d), we have

$$x_1 - 2x_4 + x_5 = 6$$

The matrix representation of these equations is given as follows.

x_0	x_1	x_2	x_3	x_4	x_5	rhs
1	0	0	0	1	1	33
0	0	0	1	−4	1	4
0	0	1	0	3	−1	3
0	1	0	0	−2	1	6

From (12.6), we have $x_0 = 33 - x_4 - x_5$. Since $x_4, x_5 \geq 0$, x_0 has the maximal value 33 when $x_4 = x_5 = 0$. The values of basic variables can be computed from the equations with $x_4 = x_5 = 0$: $x_1 = 6$, $x_2 = 3$, and $x_3 = 6$. That is, x_0 has the optimal value at point $(x_1, x_2, x_3, x_4, x_5) = (6, 3, 4, 0, 0)$. The simplex method stops with this result.

From this example, we can see that the simplex method is a local search procedure which starts from $(0, 0, 16, 9, 24)$, finds its best neighboring point $(8, 0, 8, 9, 0)$ through the pivot equation, then finds the next point $(6, 3, 4, 0, 0)$. The search stops when it reaches a local optimum. Fortunately, since all the constraints are linear, a local optimum is the global optimum.

12.2.2 The Simplex Algorithm

Suppose the linear equational system has m constraints and n variables, with the first $m + 1$ variables as basic variables. Its standard form is then:

$$\begin{array}{l} \text{Maximize } x_0 \text{ where} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (12.7) \\ \begin{array}{rcl} x_0 + \cdots + a_{0,m+1}x_{m+1} + \cdots + a_{0,n}x_n & = & b_0 \\ x_1 + \cdots + a_{1,m+1}x_{m+1} + \cdots + a_{1,n}x_n & = & b_1 \\ x_2 + \cdots + a_{2,m+1}x_{m+1} + \cdots + a_{2,n}x_n & = & b_2 \\ \ddots & \vdots & \\ x_m + a_{m,m+1}x_{m+1} + \cdots + a_{m,n}x_n & = & b_m \\ x_1, x_2, ..., x_n & \geq & 0 \end{array} \end{array}$$

Theorem 12.2.1. (optimality criterion) *For the linear equational system of* (12.7)*, if* $a_{0,j} \geq 0$ *for* $j = m+1, ..., n$*, then the maximal value of* x_0 *is* b_0 *and is attained at the point* $(b_1, b_2, ..., b_m, 0, ..., 0)$*.*

Proof. From the first row of (12.7), we have $x_0 = b_0 - (a_{0,m+1}x_{m+1} + \cdots + a_{0,n}x_n)$. Since $a_{0,j} \geq 0$ and $x_j \geq 0$ for $j = m+1, ..., n$, $a_{0,m+1}x_{m+1} + \cdots + a_{0,n}x_n \geq 0$. Hence the maximal value of x_0 is b_0 when $x_j = 0$ for $j = m+1, ..., n$. In this situation, $x_i = b_i$ for $i = 1, ..., m$. □

Theorem 12.2.2. (Unbounded criterion) *For the linear equational system of* (12.7)*, if there is an index* j*,* $m+1 \leq j \leq n$*, such that* $a_{0,j} < 0$ *and* $a_{i,j} \leq 0$ *for all* $i = 1, ..., m$ *then the optimal value of* x_0 *is* ∞*.*

Proof. If there is an index j, $m+1 \leq j \leq n$, $a_{0,j} < 0$, and $a_{i,j} \leq 0$ for all $i = 1, ..., m$, then we have infinite many points, where x_j can have any value, $x_i = b_i - a_{i,j}x_j$ for $i = 1, ..., m$, and $x_i = 0$ for $m+1 \leq i \leq n$ and $i \neq j$. Each of these points defines a feasible solution point for x_0, i.e., $x_0 = b_0 - a_{0,j}x_j$. Thus, x_0 goes to ∞ when x_j goes to ∞ because $a_{0,j} < 0$. □

Example 12.2.3. Suppose we wish to maximize x_0 with

$$\begin{array}{rrcl} x_0 & + 2x_3 - 3x_4 & = & 10 \\ x_1 & + 2x_3 - x_4 & = & 3 \\ x_2 & - x_3 - 2x_4 & = & 2 \end{array}$$

Let $x_3 = 0$, then $(x_1, x_2, x_3, x_4) = (x_4 + 3, 2x_4 + 2, 0, x_4)$ is a feasible solution point for $x_0 = 3x_4 + 10$, which goes to ∞ when x_4 does. □

The above two theorems give us two halting criteria for the simplex algorithm: one gives us the optimum solution and the other gives us an unbound solution. Let *haltCondition*(M) be the Boolean function which checks the conditions in the above two theorems. Then the outline of the simplex algorithm can be described below, following the illustration by the example in the previous subsection.

proc *Simplex*(S)
1 Convert S in standard form and store it in matrix M.
2 Let r be the row of M with basic variable x_0, which is to be maximized.
3 **while** $\neg$ *haltCondition*(M) **do**
4 select a non-basic variable v_1 in r with the minimal coefficient.
5 select a basic variable v_2 by pivoting with respect to v_1.
6 switch the roles of v_1 and v_2 and convert M in standard form.

The complexity of the simplex algorithm is exponential in the worst case. Since there are polynomial decision procedures for linear programming, this means that there are at least theoretically faster decision procedures. But in contrast to those polynomial decision procedures, the simplex algorithm has all properties necessary for an efficient linear programming solver: It produces minimal conflict explanations, handles backtracking e However, in practice, ciently, and is highly incremental. In practice, these properties are more important than the dierence between polynomial and exponential worst-case complexity [61]. It also helps that the simplex algorithm rarely reaches its st case in practice.

12.2.3 Linear Programming

The name of "linear programming" exists before the age of computers. Here "programming" means "table", as the data of a linear programming problem, or *linear program* for short, are usually stored in a table.

The standard form of a linear program is give as follows:

$$\begin{array}{lll} \text{Maximize} & a_{0,1}x_1 + \cdots + a_{0,n}x_n & \qquad (12.8) \\ \text{subject to} & & \\ & a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \leq b_1 & \\ & a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \leq b_2 & \\ & \cdots & \\ & a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n \leq b_m & \\ & x_1, x_2, ..., x_n \geq 0 & \end{array}$$

where each $a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n \leq b_i$ is a *atomic linear constraint* and we are looking for values of x_i, $1 \leq i \leq n$, such that the objective function is maximized.

Example 12.2.4. Below is a linear program with $n = 2$ and $m = 3$:

$$\begin{array}{lll} \text{Maximize} & 4x_1 + 3x_2 & \qquad (12.9) \\ \text{subject to} & & \\ & x_1 + 2x_2 \leq 16 & \\ & x_1 + x_2 \leq 9 & \\ & 3x_1 + 2x_2 \leq 24 & \\ & x_1, x_2 \geq 0 & \end{array}$$

Its solution will be discussed shortly. □

Many application problems can be specified as linear programs. To solve this problem using the simplex method, we introduce m new variables, called *slack*, one for each of the m constraints, to convert inequalities into equations.

Example 12.2.5. Continue from the previous example, we introduce three variables, x_3, x_4, and x_5, and convert (12.8) into a linear equational system.

$$\begin{array}{lll} \text{Maximize} & 4x_1 + 3x_2 & \qquad (12.9) \\ \text{subject to} & & \\ & x_1 + 2x_2 + x_3 = 16 & \\ & x_1 + x_2 + x_4 = 9 & \\ & 3x_1 + 2x_2 + x_5 = 24 & \\ & x_1, x_2, x_3, x_4, x_5 \geq 0 & \end{array}$$

(12.9) is identical to (12.1) in section 12.2.1 and we have seen how (12.1) is solved by the simplex method. Hence, the optimal solution of $4x_1 + 3x_2$ is 33 when $x_1 = 6$ and $x_2 = 3$. □

By definition, linear programming is an optimization problem. We can convert it into a decision problem by introducing a constant c, so that "Maximize $a_{0,1}x_1 + \cdots + a_{0,n}x_n$" becomes "Decide $a_{0,1}x_1 + \cdots + a_{0,n}x_n \geq c$". Another decision problem related to linear programming is to decide if the set of all constraints consistent. That is, can we find values of $x_1, x_2, ..., x_n$ such that all constraints are true? This problem can be easily solved if we convert linear programming into linear equational system through slack variables.

12.2.4 Integer Programming

The standard form of an integer program is the same as that of linear programming, with the additional constraints that the variables take only integers.

$$\begin{array}{lll} \text{Maximize} & a_{0,1}x_1 + \cdots + a_{0,n}x_n & \qquad (12.10) \\ \text{subject to} & & \\ & a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \leq b_1 & \\ & a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \leq b_2 & \\ & \cdots & \\ & a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n \leq b_m & \\ & x_1, x_2, ..., x_n \geq 0 x_1, x_2, ..., x_n \text{ are integers,} & \end{array}$$

where $a_{i,j}, b_i$ are assumed to be integers.

If the variables take only 0 or 1, integer programming becomes 0-1 linear programming. While linear programming can be solved in polynomial time, 0-1 linear programming is NP-complete.

Whereas the simplex method is effective for solving linear programs, there is no single technique for solving integer programs. Instead, a number of procedures

have been developed, and the performance of any particular technique appears to be highly problem-dependent. Methods to date can be classified broadly as following one of three approaches:

- Enumeration techniques: The well-known enumeration technique is the branch-and-bound procedure. This procedure is very much like the branch-and-bound procedure for maximum satisfiability (section 4.5.2) and especially effective for 0-1 linear programming.
- Cutting-plane techniques: The basic idea of the cutting plane method is to cut off parts of the feasible region of the linear programming search space, so that the optimal integer solution becomes an extreme point and therefore can be found by the simplex method. In practice, the branch-and-bound method outperforms mostly the cutting plane method. Cutting plane is the first algorithm developed for integer programming that could be proved to converge in a finite number of steps. Even though the technique is considered not efficient, it has provided insights into integer programming that have led to other, more efficient, algorithms.

In addition, several composite procedures have been proposed, which combine techniques using several of these approaches. In fact, there is a trend in computer systems for integer programming to include a number of approaches and possibly utilize them all when analyzing a given problem. The interested reader may consult a textbook on integer programming for the details of these techniques.

The constraints using other predicates can be easily converted into the standard form, as shown in the following table.

input	output
$A \geq B$	$B \leq A$
$A = B$	$A \leq B \wedge B \leq A$
$A > B$	$B + 1 \leq A$
$A < B$	$A + 1 \leq B$

input	output
$\neg(A > B)$	$A \leq B$
$\neg(A < B)$	$B \leq A$
$\neg(A \geq B)$	$A + 1 \leq B$
$\neg(A \leq B)$	$B + 1 \leq A$

Example 12.2.6. Consider the constraint $\neg(x_1 - 2x_2 < 3)$. Apply $\neg(A < B) \equiv B \leq A$, $3 \leq x_1 + 2x_2$. Moving 3 to the right and $x_1 - 2x_2$ to the left of $\leq$, we have $-x_1 + 2x_2 \leq -3$. □

The top two rows of the above table can be also used for preprocessing linear programming problems. Constraints like $x \geq a$ for $a \neq 0$ can be replaced by $y \geq 0$ with $x = y + a$.

12.2.5 Difference Arithmetic

12.3 Finite-length Linear Data Types

12.3.1 Bit Vectors

12.3.2 Arrays

12.3.3 Points

12.4 Satisfiability Modulo Theories (SMT)

Solvers for satisfiability modulo theories (SMT) check the satisfiability of first-order formulas containing operations from various theories such as the Booleans, bit-vectors, arithmetic, arrays, and recursive data types. SMT solvers are extensions of Boolean satisfiability solvers (SAT solvers) that check the satisfiability of formulas built from Boolean variables and operations. SMT solvers have a wide range of applications in hardware and software verification, extended static checking, constraint solving, planning, scheduling, test case generation, and computer security.

This section serves as a survey of SMT, demonstrating various concepts and techniques by examples, and present some of the key algorithms in the form of pseudo-code. Our survey uses the materials from Albert Oliveras and Cesare Tinelli's lectures on SMT.

Many theories of interest have (efficient) decision procedures for the satisfiability of sets (or conjunctions) of literals. In practice, we need to deal with the following three problems:

1. arbitrary Boolean combinations of literals
2. literals over more than one theory
3. formulas with quantifiers

The suggested solutions to the above problems will (1) DPLL(T), an extension of SAT solvers with theories; (2) Combination of theories; and (3) Elimination of quantifiers. These are the topics of the next three subsections.

12.4.1 DPLL(T): Extension of DPLL with Theory T

In Chapter 4, we have presented the DPLL algorithm in details and described several techniques for the efficient implementation of DPLL for propositional satisfiability. DPLL is an enumeration-based search method which tries to find a model for a set of propositional clauses. Here the model itself can be viewed as a set of literals which are true in the model.

Example 12.4.1. Consider the problem of deciding the satisfiability of the following EUF formula in conjunctive normal form:

$$g(a) = c \wedge (f(g(a)) \neg f(c) \vee g(a) = d) \wedge c \neq d$$

Let T be EUF. Since this is not a conjunction of literals, the methods proposed in Section 1 cannot be directly applied. The first step of DPLL(T) is *abstraction*: Using propositional variables for each atomic formula and obtain an instance of SAT solvers. That is, with p_1 for $g(a) = c$, p_2 for $f(g(a)) = f(c)$, p_3 for $g(a) = c$, and p_4 for $c = d$, we obtain the following propositional clauses:

$$\{p_1,\ \neg p_2 \vee p_3,\ \neg p_4\}$$

DPLL(T) will feed these clauses to a SAT solver which returns $\{p_1, \neg p_2, \neg p_4\}$ as the first model. The EUF formula corresponding to this model is

$$g(a) = c \wedge f(g(a)) \neg f(c) \wedge c \neq d$$

which can be proved to be unsatisfiable by an EUF solver. DPLL(T) will add the negation of the model as a new clause $(\neg p_1 \vee p_2 \vee p_4)$, together with the original clauses, into the SAT solver to ask for the next model. The SAT solver will return $\{p_1,\ p_3,\ \neg p_4\}$ as the second model. The EUF formula corresponding to this model is

$$g(a) = c \wedge g(a) = d \wedge c \neq d$$

which can be proved to be unsatisfiable, too. DPLL(T) then adds $(\neg p_1 \vee \neg p_3 \vee p_4)$ to the SAT solver, which will return "unsatisfiable". Finally, DPLL(T) claims that the original formula is unsatisfiable as the SAT solver runs out of the models. □

The above example illustrates several interesting aspects of DPLL(T).

1. Atomic formulas in theory T are abstracted to propositional variables. This abstraction is done once for all before a SAT solver is called.

2. The SAT solver is called multiple times, each time with an increment of input clauses, Thus, an incremental SAT solver is desired to support DPLL(T).

3. A theory solver is called multiple times, each with a conjunction of literals of the theory as input. Each input corresponds to a propositional model found by the SAT solver. The theory solver does not need to be incremental because the input changes unpredictably. That is, the interaction between a SAT solver and a theory solver goes back and forth in DPLL(T).

4. This approach of solving theory satisfiability via propositional satisfiability is said to be *lazy*: Theory information is used lazily when checking T-consistency of propositional models. In contrast, the *eager* approach converts the original theory satisfiability problem into an equisatisfiable propositional instance.

5. In this lazy approach, every tool does what it is good at:

   - SAT solver takes care of Boolean combinations;
   - Theory solver takes care of conjunction of literals in theory.

   High-level view gives the same algorithm as a CDCL SAT solver:

```
proc DPLL(T)
    S := abstraction(T)
    while (true)
        while (propagateGivesConflict(T, S))
            if (decisionLevel==0) return UNSAT
            else analyzeConflict(T, S)
        restartIfApplicable(S)
        if (!decide(S)) return SAT; // All variables assigned
```

abstraction(T): This procedure takes T, a quantifier-free CNF formulas, as input and replace each atomic formula (treating variables as identifiers) by a propositional variable so that T is abstracted as a set S of propositional clauses.

propagateGivesConflict(T, S):

```
proc propagateGivesConflict(T, S): Boolean
    while (good do
        // unit propagate
        if (BCPgivesConflict(S)) return true
        // check T-consistency of the model
        if (IsModelInconsistent(T, S)) return true
        // theory propagation
        good := TheoryPropagate(T, S)
    return false
```

The three procedures in *propagateGivesConflict*(T, S):

- *BCPgivesConflict*(S): This is a modification of *BCP* for propositional logic (Section 4.1). It returns *true* if a conflict is found.
- *IsModelInconsistent*(T, S): This procedure takes the current partial model from S and checks if the corresponding set of literals is consistent by theory solver for T. It returns *true* if the set of literals is inconsistent.
- *TheoryPropagate*(T, S): This procedure is a continuation of *IsModelInconsistent*(T, S). In addition to consistency checking, the theory solver will compute some new literals in theory as logical consequences of the set of literals. The corresponding propositional literals will be fed to S. This procedure is not needed for completeness. If the propagation is expensive, it can be omitted, or is called when a full propositional model is found.

analyzeConflict(T, S) is a modification of *analyzeConflict*(c) in section 4.2.

proc *analyzeConflict*(T, S)
  if the conflict comes from *BCPgivesConflict*(T, S)
    c := *conflictingClause*
  else c := *explainTheoryInconsistency*(T)
  while c contains more than one literal of last *decisionLevel*
    l := last literal assigned in c
    c := *resolution*$(c, reason(l))$
  undo(*secondMaxLevel*(c))
  insertNewClause(c, S)

The procedures *undo* and *insertNewClause* are the same as those in section 4.2 for conflict-directed clause learning. The procedure *explainTheoryInconsistency*(T) is a continuation of the procedure *IsModelInconsistent*(T, S) and perform the following steps:

1. Find a conjunction of literals $l_1 \wedge \cdots \wedge l_n$ such that they are part of the current partial the model when T-inconsistency was found and their corresponding literals in theory are T-inconsistent;
2. Return $\neg(l_1 \wedge \cdots \wedge l_n)$ as a propositional clause.

Example 12.4.2. Suppose T contains some clauses plus the following:

$$(b = c), \quad h(a) = h(c) \vee p, \quad a = b \vee \neg p \vee a = d, \quad a = b \vee a \neq d$$

The abstraction procedure will convert T into S with the following clauses:

$$(s), \quad (t \vee p), \quad (q \vee \neg p \vee r), \quad (q \vee \neg r)$$

where q stands for $a = b$, r for $a = d$, s for $b = c$, and t for $h(a) = h(c)$.

BCP(S) will make s (i.e., $b = c$) to be true. Suppose *Decide* chooses $\neg t$ ($h(a) \neq h(c)$) to be true. *BCP*(S) will make p to be true due to $(t \vee p)$.

T-Propagation will make $\neg q$ ($a \neq b$) to be true, since $h(a) \neq h(c)$ and $b = c$ are true. So *Explain*($a \neg b$) will be $\{h(a) \neq h(c),\ b = c\}$ and *Reason*($\neg q$ is $(\neq q \vee \neg s \vee t)$.

With $\neg q$ being true, *BCP*(S) will find that $(q \vee \neg p \vee r)$ becomes unit, so r ($a = d$) becomes true. Continuing *BCP*(S), $(q \vee \neg r)$ becomes conflicting.

With $(q \vee \neg r)$ being the conflicting clause, *conflictAnalysis*(T, S) will perform the following resolutions:

- Resolve on r with *Reason*(r) $= (q \vee \neg p \vee r)$ and $(q \vee \neg r)$, the resolvent is $(q \vee \neg p)$.
- Resolve on $\neq q$ with *Reason*($\neg q$) $= (\neg q \vee \neg s \vee t)$ and $(q \vee \neg p)$, the resolvent is $(\neg p \vee \neg s \vee t)$.
- Resolve on p with *Reason*(p) $= (t \vee p)$ and $(\neg p \vee \neg s \vee t)$, the resolvent is $(\neg s \vee t)$.

The new clause generated by *conflictAnalysis*(T, S) is $(\neg s \vee t)$. □

For certain theories, consistency checking requires case reasoning. Example: consider the theory of arrays and the set of literals read(write(A, i, x), j) 6= x read(write(A, i, x), j) 6= read(A, j)

A complete T-solver might need to reason by cases via internal case splitting and backtracking mechanisms. An alternative is to lift case splitting and backtracking from the T-Solver to the SAT engine. Basic idea: encode case splits as sets of clauses and send them as needed to the SAT engine for it to split on them. Possible benefits: All case-splitting is coordinated by the SAT engine Only have to implement case-splitting infrastructure in one place Can learn a wider class of lemmas (more details later)

12.4.2 Combination of Theories

In software verification, formulas like the following one arise:

$$a = b + 2 \wedge A = \text{write}(B, a + 1, 4) \wedge \text{read}(A, b + 3) = 2 \wedge f(a - 1) \neq f(b + 1)$$

To deal efficiently with this formula, we need the following theories:

- The theory of linear arithmetic (T_{LA})
- The theory of arrays (T_{arr})
- The theory of uninterpreted functions (T_{EUF})

Now the question is: Given T-solvers for the three individual theories, can we combine them to obtain one for $(T_{LA} \cup T_{arr} \cup T_{EUF})$? Under certain conditions the Nelson-Oppen combination method gives a positive answer.

12.4.2.1 Nelson-Oppen Combination

Definition 12.4.3. *A theory T is* stably-infinite *if every T-satisfiable quantifier-free formula has an infinite model.*

Definition 12.4.4. *A theory T is* convex *if for any set S of literals, $S \models a_1 = b_1 \vee \cdots a_n = b_n$, then $S \models a_i = b_i$ for some $1 \leq i \leq n$.*

proc *Nelson-Oppen*(S)
Purify S and split it into S_1 and S_2.
while (*true* **do**
if S_1 is T_1-unsatisfiable **return** *false*
if S_2 is T_2-unsatisfiable **return** *false*
if $S_1 \models (x = y)$ for $(x = y) \in E - S_2$
$S_2 = S_2 \cup \{x = y\}$
if $S_2 \models (x = y)$ for $(x = y) \in E - S_1$
$S_1 = S_1 \cup \{x = y\}$
return *true*

Theorem 12.4.5. *Given two signature-disjoint, stably-infinite and covex theories T_1 and T_2, and a set S of literals over the signature of $T_1 \cup T_2$, the $(T_1 \cup T_2)$-satisfiability of S can be checked with the Nelson-Oppen algorithm.*

12.5 Exercise Problems

Drill Questions

1. In Example 12.1.1, the rewrite rules are made from

$$\{(x_0 = a[3], x_1 = f(a[2], x_0), x_2 = f(a[1], x_1), y = f(a[1], f(a[2], a[3]))\}$$

by orienting them from left to right. Please show $x_2 = y$ by the rewrite rules obtained by orienting the above equations from right to left and list each step of rewriting.

2. Run *NelsonOppen*(A) by hand for $A = \{f^3a = a, f^5a = a\}$. Using Example 12.1.6 as model to describe the changes of $up(v)$, $h(v)$, $use(v)$ for any node v.

3. If you have a software tool to solve a linear program for maximizing an objective function, how to use this tool to solve a linear program where the goal is to minimize an objective function?

Exercises

1. Given the set E of ground equations from Example 12.1.10,

$$\{f(a) = c_1, f(c_1) = c_2, f(c_2) = a, f(c_2) = c_3, f(c_3) = c_4, f(c_4) = a\}$$

please run the Knuth-Bendix procedure on E_1 with the ordering $f > a > c_1 > c_2 > c_3 > c_4$, and list all the rewrite rules made from the procedure in the order they were created.

2. Let $E_0 = \{f(a, g(a)) = g(g(a))\}$ (Example 12.1.9). Please run *KBG(flattenSlicing*(E_0)*)* by hand and show the contents of $nf(c)$, $class(c)$, and $use(c)$ for each identifier c before the main loop of *KBG*.

3. The final matrix given in section 12.2.1 is

x_0	x_1	x_2	x_3	x_4	x_5	rhs
1	0	0	0	1	1	33
0	0	0	1	−4	1	4
0	0	1	0	3	−1	3
0	1	0	0	−2	1	6

Please write the corresponding linear equational system in standard form and show step by step how the initial matrix became the final matrix through operations on the matrices (such as multiplying one row by constant, or subtracting one row by another row). [2] [1]

REFERENCES

[1] J. Doe. A paper. *Another journal*, 2009.

[2] J. Smith. A title. *A Journal*, 2010.

Index

α-rule
  $\Box$, 376
  release, 389
β-rule
  $\Diamond$, 376
  until, 389

alphabet, 171
argument, 6
arity, 172
assertion, 328
  middle, 335
atom
  atomic formula, 173
axiom, 20
  consistent, 20
  progress, 397

Backus–Naur form, 24
basic variable, 415
BCP
  boolean constraint propagation, 113
BDD
  binary decision diagram, 54
Boolean algebra, 19
Boolean functions, 30
Boolean variables, 4

Church–Turing thesis, 26
clause, 47
  binary clause, 99
  definite, 115, 281
  empty clause, 99
  fact, 115, 280
  hard, 158
  Horn, 115
  negative, 99
  positive, 99
  query, 281
  rule, 115, 281
  soft, 158
  unit clause, 99
closure
  congruence, 246
  equivalence, 246
  transitive, 246
CNF
  2CNF, 125
  conjunctive normal form, 47
completeness, 21
congruence, 244
  class, 244
consistency, 20
constructor, 259
correctness
  partial, 339
  total, 339
countable, 13

decision procedure, 21
  semi-decision procedure, 21
deletion
  pure, 112
  subsumption, 112
  tautology, 112
  unit, 113
DNF

disjunctive normal form, 49
domain, 176

entailment, 41
equality crossing, 260
equivalence
LTL, 370
MTL, 363
exit condition, 334

fair strategy, 22, 98
fallacy, 6
deductive, 6
logical, 6
flattening, 275
form
canonical, 44
normal, 44
formula
X-, 374
closed, 175
dual, 47
first-order, 173
instance, 33
model, 36
MTL, 360
satisfiable, 36
valid, 37
function
defined, 259
domain, 12
interpreted, 259
range, 12

ground, 173
ground equality, 403
ground equation, 257
group, 245

Herbrand
base, 199
interpretations, 199
model, 200
universe, 199
Hilbert system, 92
Hoare triple, 329
hyper-resolution, 234
negative, 234

implication
contrapositive, 39
converse, 39
independence, 20
independent, 20
induction
structural rule, 259
INF
ite normal form, 52
inference rule
sound, 21
inference graph, 97
inference rule, 21
axiom rule, 90
sound, 90
inference system, 21
injective, 12
interpretation, 34
full, 118
partial, 118
invariance
basic rule, 396
invariant, 395
bounding, 353
essential, 353
linear, 395
local, 395
loop, 334
polyhedral, 396
reaffirmed, 395

Kripke model, 363
Kripke frame, 361

leaf, 376
    closed, 374
    expandable, 374
    open, 374
literal, 46
    decision, 128
    first-order, 191
    implied, 128
    negative, 46
    positive, 46
local search, 152
logic
    equivalence, 39
    formal logic, 2
    Hoare, 328
    logical connectives, 3
    logical operator, 3
    philosophical logic, 2
    philosophy of logic, 2
    propositional logic, 4
logical consequence, 41, 370
loop, 330
    invariant, 334

marked dead, 380
MaxSAT
    hybrid, 158
    partial and weighted, 158
    solution, 158
    weighted, 158
maxterm, 47
minterm, 49
model, 19, 182, 369
modus ponens, 21
multiset, 223

next
    X-formula, 374
    X-rule, 375
NNF
    negation normal form, 46

operator
    minimum, 57
    sufficient, 57
order
    minimal, 12
    partial order, 11
    simplification, 223
    total order, 11
    well-founded, 12

pivot equation, 416
PNF
    prenex normal form, 191
    PrenexNF, 194
postcondition, 329
precedence, 224
precondition, 329
    weakest, 333
predicate, 170
predicate calculus, 170
probability logic, 4
product
    term, 49
Prolog
    fact, 280
    query, 281
    rule, 281
proof, 21, 90
proof procedure, 20
proposition, 3
propositional variable, 4
prover
    refutation prover, 83
    tautology prover, 83

theorem prover, 83

quantifier
existential, 171
universal, 171

reachable, 376
reason, 119
reflexing, 266
relation, 11
release, 386
strong, 388
resolution
binary, 99
core, 100
cut, 97
hyper-resolution, 233
input resolution, 101
linear resolution, 101
negative resolution, 101
ordered resolution, 101
positive resolution, 101
proof, 100
saturation, 109
set-of-support, 101
unit resolution, 101
resolvent, 99

SAT
decoder, 147
encoder, 147
MAX-SAT, 152
satisfiable, 182
equally, 103
LTL, 369
MTL, 363
self-denotation, 199
sentence, 175
sequent, 94
set, 9
Skolem
function, 193
soundness, 21
weighted clauses, 161
stable, 223
state
sequence, 368
statement
composed, 3
simple, 3
STeP, 390
style
deduction, 84
enumeration, 84
reduction, 84
saturation, 108
substitution, 205
affected variable, 205
idempotent, 206
renaming, 205
subsumption
back, 222
forward, 222
suffix
immediate, 368
surjective, 12

tableau
LTL, 373
tautology, 37
temporal logic, 359
term, 172
graph, 212
theorem, 20, 42
inductive, 258
theorem proving, 37
theory, 42
convex, 428
equality, 246

stably-infinite, 428
truth table, 30
truth values, 3
Tseitin transformation, 104

unequality, 403
unifiable, 206
unification
clash, 208
occur check, 208
unifier, 206
mgu, 208
more general, 208
unit deletion, 222
unit propagation, 113
universal set, 10
universe, 176
until, 385
weak, 388

valid
LTL, 369
MTL, 363
variable
bounded, 175
free, 175
scope, 175
slack, 420
verification condition, 342

WCNF
weighted CNF, 157
well-founded
induction, 223