The Calculus of Computation

Aaron R. Bradley · Zohar Manna

The Calculus of Computation

Decision Procedures with Applications to Verification

With 60 Figures

Springer

Authors

Aaron R. Bradley
Zohar Manna

Gates Building, Room 481
Stanford University
Stanford, CA 94305
USA

arbrad@cs.stanford.edu
manna@cs.stanford.edu

Library of Congress Control Number: 2007932679

ACM Computing Classification (1998): B.8, D.1, D.2, E.1, F.1, F.3, F.4, G.2, I.1, I.2

ISBN 978-3-540-74112-1 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Typesetting by the authors
Production: LE-TEX Jelonek, Schmidt & Vöckler GbR, Leipzig
Cover design: KünkelLopka Werbeagentur, Heidelberg

Printed on acid-free paper 45/3180/YL - 5 4 3 2 1 0

To my wife,

Sarah

A.R.B.

To my grandchildren,

Itai
Maya
Ori

Z.M.

Preface

Logic is the calculus of computation. Forty-five years ago, John McCarthy predicted in *A Basis for a Mathematical Theory of Computation* that "the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last". The field of *computational logic* emerged over the past few decades in partial fulfillment of that vision. Focusing on producing efficient and powerful algorithms for deciding the satisfiability of formulae in logical theories and fragments, it continues to push the frontiers of general computer science.

This book is about computational logic and its applications to *program verification*. Program verification is the task of analyzing the correctness of a program. It encompasses the formal specification of what a program should do and the formal proof that the program meets this specification. The reasoning power that computational logic offers revolutionized the field of verification. Ongoing research will make verification standard practice in software and hardware engineering in the next few decades. This acceptance into everyday engineering cannot come too soon: software and hardware are becoming ever more ubiquitous and thus ever more the source of failure.

We wrote this book with an undergraduate and beginning graduate audience in mind. However, any computer scientist or engineer who would like to enter the field of computational logic or apply its products should find this book useful.

Content

The book has two parts. Part I, *Foundations*, presents first-order logic, induction, and program verification. The methods are general. For example, Chapter 2 presents a complete proof system for first-order logic, while Chapter 5 describes a relatively complete verification methodology. Part II, *Algorithmic Reasoning*, focuses on specialized algorithms for reasoning about fragments of first-order logic and for deducing facts about programs. Part II trades generality for decidability and efficiency.

The first three chapters of Part I introduce first-order logic. Chapters 1 and 2 begin our presentation with a review of propositional and predicate logic. Much of the material will be familiar to the reader who previously studied logic. However, Chapter 3 on first-order theories will be new to many readers. It axiomatically defines the various first-order theories and fragments that we study and apply throughout the rest of the book. Chapter 4 reviews induction, introducing some forms of induction that may be new to the reader. Induction provides the mathematical basis for analyzing program correctness.

Chapter 5 turns to the primary motivating application of computational logic in this book, the task of verifying programs. It discusses *specification*, in which the programmer formalizes in logic the (sometimes surprisingly vague) understanding that he has about what functions should do; *partial correctness*, which requires proving that a program or function meets a given specification if it halts; and *total correctness*, which requires proving additionally that a program or function always halts. The presentation uses the simple programming language pi and is supported by the verifying compiler πVC (see **The πVC System**, below, for more information on πVC). Chapter 6 suggests strategies for applying the verification methodology.

Part II on *Algorithmic Reasoning* begins in Chapter 7 with quantifier-elimination methods for limited integer and rational arithmetic. It describes an algorithm for reducing a quantified formula in integer or rational arithmetic to an equivalent formula without quantifiers.

Chapter 8 begins a sequence of chapters on decision procedures for quantifier-free and other fragments of theories. These fragments of first-order theories are interesting for three reasons. First, they are sometimes decidable when the full theory is not (see Chapters 9, 10, and 11). Second, they are sometimes efficiently decidable when the full theory is not (compare Chapters 7 and 8). Finally, they are often useful; for example, proving the verification conditions that arise in the examples of Chapters 5 and 6 requires just the fragments of theories studied in Chapters 8–11. The simplex method for linear programming is presented in Chapter 8 as a decision procedure for deciding satisfiability in rational and real arithmetic without multiplication.

Chapters 9 and 11 turn to decision procedures for non-arithmetical theories. Chapter 9 discusses the classic congruence closure algorithm for equality with uninterpreted functions and extends it to reason about data structures like lists, trees, and arrays. These decision procedures are for quantifier-free fragments only. Chapter 11 presents decision procedures for larger fragments of theories that formalize array-like data structures.

Decision procedures are most useful when they are combined. For example, in program verification one must reason about arithmetic and data structures simultaneously. Chapter 10 presents the Nelson-Oppen method for combining decision procedures for quantifier-free fragments. The decision procedures of Chapters 8, 9, and 11 are all combinable using the Nelson-Oppen method.

Chapter 12 presents a methodology for constructing *invariant generation procedures*. These procedures reason inductively about programs to aid in

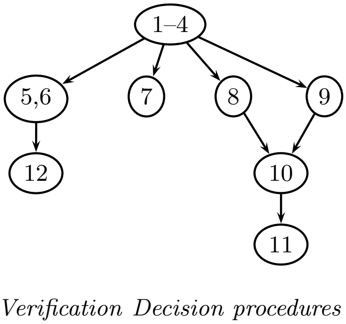

Fig. 0.1. The chapter dependency graph

verification. They relieve some of the burden on the programmer to provide program annotations for verification purposes. For now, developing a static analysis is one of the easiest ways of bringing formal methods into general usage, as a typical static analysis requires little or no input from the programmer. The chapter presents a general methodology and two instances of the method for deducing arithmetical properties of programs.

Finally, Chapter 13 suggests directions for further reading and research.

Teaching

This book can be used in various ways and taught at multiple levels. Figure 0.1 presents a dependency graph for the chapters. There are two main tracks: the *verification track*, which focuses on Chapters 1–4, 5, 6, and 12; and the *decision procedures track*, which focuses on Chapters 1–4 and 7–11. Within the decision procedures track, the reader can focus on the *quantifier-free decision procedures track*, which skips Chapters 7 and 11. The reader interested in quickly obtaining an understanding of modern combination decision procedures would prefer this final track.

We have annotated several sections with a $^\bigstar$ to indicate that they provide additional depth that is unnecessary for understanding subsequent material. Additionally, all proofs may be skipped without preventing a general understanding of the material.

Each chapter ends with a set of exercises. Some require just a mechanical understanding of the material, while others require a conceptual understanding or ask the reader to think beyond what is presented in the book. These latter exercises are annotated with a $^\bigstar$. For certain audiences, additional exercises might include implementing decision procedures or invariant generation procedures and exploring certain topics in greater depth (see Chapter 13).

In our courses, we assign program verification exercises from Chapters 5 and 6 throughout the term to give students time to develop this important skill. Learning to verify programs is about as difficult for students as learning to program in the first place. Specifying and verifying programs also strengthens the students' facility with logic.

Bibliographic Remarks

Each chapter ends with a section entitled **Bibliographic Remarks** in which we attempt to provide a brief account of the historical context and development of the chapter's material. We have undoubtedly missed some important contributions, for which we apologize. We welcome corrections, comments, and historical anecdotes.

The πVC System

We implemented a *verifying compiler* called πVC to accompany this text. It allows users to write and verify annotated programs in the `pi` programming language. The system and a set of examples, including the programs listed in this book, are available for download from `http://theory.stanford.edu/~arbrad/pivc`. We plan to update this website regularly and welcome readers' comments, questions, and suggestions about πVC and the text.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. CSR-0615449 and CNS-0411363 and by Navy/ONR contract N00014-03-1-0939. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Navy/ONR. The first author received additional support from a Sang Samuel Wang Stanford Graduate Fellowship.

We thank the following people for their comments throughout the writing of this book: Miquel Bertran, Andrew Bradley, Susan Bradley, Chang-Seo Park, Caryn Sedloff, Henny Sipma, Matteo Slanina, Sarah Solter, Fabio Somenzi, Tomás Uribe, the students of CS156, and Alfred Hofmann and the reviewers and editors at Springer. Their suggestions helped us to improve the presentation substantially. Remaining errors and shortcomings are our responsibility.

Stanford University,
June 2007

Aaron R. Bradley
Zohar Manna

Contents

Part I Foundations

Part II Algorithmic Reasoning

Part I

Foundations

Everything is vague to a degree you do not realize till you have tried to make it precise.

— Bertrand Russell
Philosophy of Logical Atomism, 1918

Modern design and implementation of software and hardware systems lacks precision. Design documents written in a natural language admit misinterpretation. Informal arguments about why a system works miss crucial weaknesses. The resulting systems are fragile. Part I of this book presents an alternative approach to system design and implementation based on using a formal language to specify and reason about software systems.

Chapters 1 and 2 introduce the (first-order) predicate calculus. Chapter 1 presents the propositional calculus, and Chapter 2 presents the full predicate calculus. A central task is determining whether formulae of the calculus are valid. Chapter 3 formalizes common data types of software in the predicate calculus. It also introduces the concepts of decidability and complexity of deciding validity of formulae.

The final three chapters of Part I discuss applications of the predicate calculus. Chapter 4 formalizes mathematical induction in the predicate calculus, in the process introducing several forms of induction that may be new to the reader. Chapters 5 and 6 then apply the predicate calculus and mathematical induction to the specification and verification of software. Specification consists of asserting facts about software. Verification applies mathematical induction to prove that each assertion evaluates to true when program control reaches it; and to prove that program control eventually reaches specific program locations.

Part I thus provides the mathematical foundations for precise engineering. Part II will investigate algorithmic aspects of applying these foundations.

1

Propositional Logic

> *A deduction is speech in which, certain things having been supposed, something different from the things supposed results of necessity because of their being so.*
>
> — Aristotle
> *Prior Analytics*, 4th century BC

A calculus is a set of symbols and a system of rules for manipulating the symbols. In an interesting calculus, the symbols and rules have meaning in some domain that matters. For example, the differential calculus defines rules for manipulating the integral symbol over a polynomial to compute the area under the curve that the polynomial defines. Area has meaning outside of the calculus; the calculus provides the tool for computing such quantities. The domain of the differential calculus, loosely speaking, consists of real numbers and functions over those numbers.

Computer scientists are interested in a different domain and thus require a different calculus. The behavior of programs, or computation, is a computer scientist's chief concern. What is an appropriate domain for studying computation? The basic entity of the domain is *state*: roughly, the assignment of values (for example, Booleans, integers, or addresses) to variables. Pairs of states comprise *transitions*. A *computation* is a sequence of states, each adjacent pair of which is a transition. A program defines the form of its states, the set of transitions between states, and the set of computations that it can produce. A program's set of computations characterizes the program itself as precisely as its source code. Chapter 5 studies these ideas in depth.

With a domain in mind, a computer scientist can now ask questions. Does this program that accepts an array of integers produce a sorted array? In other words, does each of the program's computations have a state in which a sorted array is returned? Does this program ever access unallocated memory? Does this function always halt? To answer such questions, we need a calculus to reason about computations.

This chapter and the next introduce the calculus that will be the basis for studying computation in this book. In this chapter, we cover **propositional logic** (**PL**); in the next chapter, we build on the presentation to define **first-order logic** (**FOL**). PL and FOL are also known as **propositional calculus** and **predicate calculus**, respectively, because they are calculi for reasoning about propositions ("the sky is blue", "this comment references itself") and predicates ("x is blue", "y references z"), respectively. Propositions are either true or false, while predicates evaluate to true or false depending on the values given to their parameters (x, y, and z).

Just as differential calculus has a set of symbols, a set of rules, and a mapping to reality that provides its meaning, propositional logic has its own symbols, rules of inference, and meaning. Sections 1.1 and 1.2 introduce the *syntax* and *semantics* (meaning) of PL formulae. Then Section 1.3 discusses two concepts that are fundamental throughout this book, *satisfiability* (Is this formula ever true?) and *validity* (Is this formula always true?), and the rules for computing whether a PL formula is satisfiable or valid. Rules for manipulating PL formulae, some of which preserve satisfiability and validity, are discussed in Section 1.5 and applied in Section 1.6.

1.1 Syntax

In this section, we introduce the syntax of PL. The **syntax** of a logical language consists of a set of symbols and rules for combining them to form "sentences" (in this case, **formulae**) of the language.

The basic elements of PL are the **truth symbols** $\top$ ("true") and $\bot$ ("false") and the **propositional variables**, usually denoted by P, Q, R, P_1, $P_2, \ldots$. A countably infinite set of propositional variable symbols exists. **Logical connectives**, also called **Boolean connectives**, provide the expressive power of PL. A **formula** is simply $\top$, $\bot$, or a propositional variable P; or the application of one of the following connectives to formulae F, F_1, or F_2:

- $\neg F$: negation, pronounced "not";
- $F_1 \wedge F_2$: conjunction, pronounced "and";
- $F_1 \vee F_2$: disjunction, pronounced "or";
- $F_1 \rightarrow F_2$: implication, pronounced "implies";
- $F_1 \leftrightarrow F_2$: iff, pronounced "if and only if".

Each connective has an **arity** (the number of arguments that it takes): negation is **unary** (it takes one argument), while the other connectives are **binary** (they take two arguments). The left and right arguments of $\rightarrow$ are called the **antecedent** and **consequent**, respectively.

Some common terminology is useful. An **atom** is a truth symbol $\top$, $\bot$ or propositional variable P, Q, ... A **literal** is an atom α or its negation $\neg\alpha$. A **formula** is a literal or the application of a logical connective to a formula or formulae.

Formula G is a **subformula** of formula F if it occurs syntactically within G. More precisely,

- the only subformula of P is P;
- the subformulae of $\neg F$ are $\neg F$ and the subformulae of F;
- and the subformulae of $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \rightarrow F_2$, $F_1 \leftrightarrow F_2$ are the formula itself and the subformulae of F_1 and F_2.

Notice that every formula is a subformula of itself. The **strict subformulae** of a formula are all its subformulae except itself.

Example 1.1. Consider the formula

$$F:\ (P \wedge Q)\ \rightarrow\ (P \vee \neg Q)\ .$$

It contains two propositional variables, P and Q. Each instance of P and Q is an atom and a literal. $\neg Q$ is a literal, but not an atom. F has six distinct subformulae:

$$F\ ,\quad P \vee \neg Q\ ,\quad \neg Q\ ,\quad P \wedge Q\ ,\quad P\ ,\quad Q\ .$$

Its strict subformulae are all of its subformulae except F itself. ■

Parentheses are cumbersome. We define the relative precedence of the logical connectives from highest to lowest as follows: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. Additionally, let $\rightarrow$ and $\leftrightarrow$ associate to the right, so that $P \rightarrow Q \rightarrow R$ is the same formula as $P \rightarrow (Q \rightarrow R)$.

Example 1.2. Abbreviate F of Example 1.1 as

$$F':\ P \wedge Q\ \rightarrow\ P \vee \neg Q\ .$$

Also,

$$P_1 \wedge \neg P_2 \wedge \top \vee \neg P_1 \wedge P_2$$

stands for

$$(P_1 \wedge ((\neg P_2) \wedge \top)) \vee ((\neg P_1) \wedge P_2)\ .$$

Finally,

$$P_1\ \rightarrow\ P_2\ \rightarrow\ P_3$$

abbreviates

$$P_1\ \rightarrow\ (P_2\ \rightarrow\ P_3)\ .$$

■

1.2 Semantics

So far, we have considered the syntax of PL. The **semantics** of a logic provides its meaning. What exactly is meaning? In PL, meaning is given by the **truth values** true and false, where true $\neq$ false. Our objective is to define how to give meaning to formulae.

The first step in defining the semantics of PL is to provide a mechanism for evaluating the propositional variables. An **interpretation** I assigns to every propositional variable exactly one truth value. For example,

$$I : \{P \mapsto \mathsf{true},\ Q \mapsto \mathsf{false},\ \ldots\}$$

is an interpretation assigning true to P and false to Q, where ... elides the (countably infinitely many) assignments that are not relevant to us. That is, I assigns to every propositional variable available to us (and there are countably infinitely many) a value. We usually do not write the elision. Clearly, many interpretations exist.

Now given a PL formula F and an interpretation I, the truth value of F can be computed. The simplest manner of computing the truth value of F is via a **truth table**. Let us first examine truth tables that indicate how to evaluate each logical connective in terms of its arguments. First, a propositional variable gets its truth value immediately from I. Now consider the possible evaluations of F: it is either true or false. How is $\neg F$ evaluated? The following table summarizes the possibilities, where 0 corresponds to the value false, and 1 corresponds to true:

F	$\neg F$
0	1
1	0

The other connective can be defined similarly given values of F_1 and F_2:

F_1	F_2	$F_1 \land F_2$	$F_1 \lor F_2$	$F_1 \rightarrow F_2$	$F_1 \leftrightarrow F_2$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

In particular, $F_1 \rightarrow F_2$ is false iff F_1 is true and F_2 is false. (Throughout the book, we use the word "iff" to abbreviate the phrase "if and only if"; one can also read it as "precisely when".)

Example 1.3. Consider the formula

$$F : P \land Q \ \rightarrow \ P \lor \neg Q$$

and the interpretation

$$I : \{P \mapsto \mathsf{true},\ Q \mapsto \mathsf{false}\}\ .$$

To evaluate the truth value of F under I, construct the following table:

P	Q	$\neg Q$	$P \wedge Q$	$P \vee \neg Q$	F
1	0	1	0	1	1

The top row is given by the subformulae of F. I provides values for the first two columns; then the semantics of PL provide the values for the remainder of the table. Hence, F evaluates to true under I. ■

This tabular notation is convenient, but it is unsuitable for the predicate logic of Chapter 2. Instead, we introduce an **inductive definition** of PL's semantics that will extend to Chapter 2. An inductive definition defines the meaning of basic elements first, which in the case of PL are atoms. Then it assumes that the meaning of a set of elements is fixed and defines a more complex element in terms of these elements. For example, in PL, $F_1 \wedge F_2$ is a more complex formula than either of the formulae F_1 or F_2.

Recall that we want to compute whether F has value true under interpretation I. We write $I \models F$ if F evaluates to true under I and $I \not\models F$ if F evaluates to false. To start our inductive definition, define the meaning of truth symbols:

$$\begin{array}{lll} I & \models & \top \\ I & \not\models & \bot \end{array}$$

Under any interpretation I, $\top$ has value true, and $\bot$ has value false. Next, define the truth value of propositional variables:

$$I \ \models\ P \qquad \text{iff } I[P] = \mathsf{true}$$

P has value true iff the interpretation I assigns P to have value true.

Since an interpretation assigns a truth value to every propositional variable, I assigns false to P when I does not assign true to P. Thus, we can instead define the truth values of propositional variables as follows:

$$I \ \not\models\ P \qquad \text{iff } I[P] = \mathsf{false}$$

Since $\mathsf{true} \neq \mathsf{false}$, both definitions yield the same (unique) truth values.

Having completed the base cases of our inductive definition, we turn to the inductive step. Assume that formulae F, F_1, and F_2 have truth values. From these formulae, evaluate the semantics of more complex formulae:

$$\begin{array}{llll} I & \models & \neg F & \text{iff } I \not\models F \\ I & \models & F_1 \wedge F_2 & \text{iff } I \models F_1 \text{ and } I \models F_2 \\ I & \models & F_1 \vee F_2 & \text{iff } I \models F_1 \text{ or } I \models F_2 \\ I & \models & F_1 \rightarrow F_2 & \text{iff, if } I \models F_1 \text{ then } I \models F_2 \\ I & \models & F_1 \leftrightarrow F_2 & \text{iff } I \models F_1 \text{ and } I \models F_2 \text{, or } I \not\models F_1 \text{ and } I \not\models F_2 \end{array}$$

In studying these definitions, it is useful to recall the earlier definitions given by the truth tables, which are free of English ambiguities.

For implication, consider also the equivalent formulation

$$I \not\models F_1 \rightarrow F_2 \qquad \text{iff } I \models F_1 \text{ and } I \not\models F_2$$

The formula $F_1 \rightarrow F_2$ has truth value true under I when either F_1 is false or F_2 is true. It is false only when F_1 is true and F_2 is false. Our inductive definition of the semantics of PL is complete.

Example 1.4. Consider the formula

$$F : P \wedge Q \rightarrow P \vee \neg Q$$

and the interpretation

$$I : \{P \mapsto \mathsf{true},\ Q \mapsto \mathsf{false}\} .$$

Compute the truth value of F as follows:

$$\begin{array}{lllll}
1. & I & \models & P & \text{since } I[P] = \mathsf{true} \\
2. & I & \not\models & Q & \text{since } I[Q] = \mathsf{false} \\
3. & I & \models & \neg Q & \text{by 2 and semantics of } \neg \\
4. & I & \not\models & P \wedge Q & \text{by 2 and semantics of } \wedge \\
5. & I & \models & P \vee \neg Q & \text{by 1 and semantics of } \vee \\
6. & I & \models & F & \text{by 4 and semantics of } \rightarrow
\end{array}$$

We considered the distinct subformulae of F according to the **subformula ordering**: F_1 precedes F_2 if F_1 is a subformula of F_2. In that order, we computed the truth value of F from its simplest subformulae to its most complex subformula (F itself).

The final line of the calculation deserves some explanation. According to the semantics for implication,

$$I \models F_1 \rightarrow F_2 \qquad \text{iff, if } I \models F_1 \text{ then } I \models F_2$$

the implication $F_1 \rightarrow F_2$ has value true when $I \not\models F_1$. Thus, line 5 is unnecessary for establishing the truth value of F. ■

1.3 Satisfiability and Validity

We now consider a fundamental characterization of PL formulae.

A formula F is **satisfiable** iff there exists an interpretation I such that $I \models F$. A formula F is **valid** iff for all interpretations I, $I \models F$. Determining satisfiability and validity of formulae are important tasks in logic.

Satisfiability and validity are dual concepts, and switching from one to the other is easy. F is valid iff $\neg F$ is unsatisfiable. For suppose that F is valid;

then for any interpretation I, $I \models F$. By the semantics of negation, $I \not\models \neg F$, so $\neg F$ is unsatisfiable. Conversely, suppose that $\neg F$ is unsatisfiable. For any interpretation I, $I \not\models \neg F$, so that $I \models F$ by the semantics of negation. Thus, F is valid.

Because of this duality between satisfiability and validity, we are free to focus on either one or the other in the text, depending on which is more convenient for the discussion. The reader should realize that statements about one are also statements about the other.

In this section, we present several methods of determining validity and satisfiability of PL formulae.

1.3.1 Truth Tables

Our first approach to checking the validity of a PL formula is the **truth-table method**. We exhibit this method by example.

Example 1.5. Consider the formula

$$F :\ P \wedge Q \ \rightarrow\ P \vee \neg Q\ .$$

Is it valid? Construct a table in which the first row is a list of the subformulae of F ordered according to the subformula ordering. Fill columns of propositional variables with all possible combinations of truth values. Then apply the semantics of PL to fill the rest of the table:

P	Q	$P \wedge Q$	$\neg Q$	$P \vee \neg Q$	F
0	0	0	1	1	1
0	1	0	0	0	1
1	0	0	1	1	1
1	1	1	0	1	1

The final column, which represents the truth value of F under the possible interpretations, is filled entirely with true. F is valid. ■

Example 1.6. Consider the formula

$$F :\ P \vee Q \ \rightarrow\ P \wedge Q\ .$$

Construct the truth table:

P	Q	$P \vee Q$	$P \wedge Q$	F
0	0	0	0	1
0	1	1	0	0
1	0	1	0	0
1	1	1	1	1

Because the second and third rows show that F can be false, F is invalid. ■

1.3.2 Semantic Arguments

Our next approach to validity checking is the **semantic argument method**. While more complicated than the truth-table method, we introduce it and emphasize it throughout the remainder of the chapter because it is our only method of evaluating the satisfiability and validity of formulae in Chapter 2.

A proof based on the semantic method begins by assuming that the given formula F is invalid: hence, there is a **falsifying interpretation** I such that $I \not\models F$. The proof proceeds by applying the semantic definitions of the logical connectives in the form of **proof rules**. A proof rule has one or more **premises** (assumed facts) and one or more **deductions** (deduced facts). An application of a proof rule requires matching the premises to facts already existing in the semantic argument and then forming the deductions. The proof rules are the following:

- According to the semantics of negation, from $I \models \neg F$, deduce $I \not\models F$; and from $I \not\models \neg F$, deduce $I \models F$:

$$\begin{array}{l} I \models \neg F \\ \hline I \not\models F \end{array} \qquad \begin{array}{l} I \not\models \neg F \\ \hline I \models F \end{array}$$

- According to the semantics of conjunction, from $I \models F \wedge G$, deduce both $I \models F$ and $I \models G$; and from $I \not\models F \wedge G$, deduce $I \not\models F$ or $I \not\models G$. The latter deduction results in a fork in the proof; each case must be considered separately.

$$\begin{array}{l} I \models F \wedge G \\ \hline I \models F \\ I \models G \end{array} \qquad \begin{array}{c} I \not\models F \wedge G \\ \hline I \not\models F \quad \mid \quad I \not\models G \end{array}$$

- According to the semantics of disjunction, from $I \models F \vee G$, deduce $I \models F$ or $I \models G$; and from $I \not\models F \vee G$, deduce both $I \not\models F$ and $I \not\models G$. The former deduction requires a case analysis in the proof.

$$\begin{array}{c} I \models F \vee G \\ \hline I \models F \quad \mid \quad I \models G \end{array} \qquad \begin{array}{l} I \not\models F \vee G \\ \hline I \not\models F \\ I \not\models G \end{array}$$

- According to the semantics of implication, from $I \models F \rightarrow G$, deduce $I \not\models F$ or $I \models G$; and from $I \not\models F \rightarrow G$, deduce both $I \models F$ and $I \not\models G$. The former deduction requires a case analysis in the proof.

$$\begin{array}{c} I \models F \rightarrow G \\ \hline I \not\models F \quad \mid \quad I \models G \end{array} \qquad \begin{array}{l} I \not\models F \rightarrow G \\ \hline I \models F \\ I \not\models G \end{array}$$

- According to the semantics of iff, from $I \models F \leftrightarrow G$, deduce $I \models F \land G$ or $I \not\models F \lor G$; and from $I \not\models F \leftrightarrow G$, deduce $I \models F \land \neg G$ or $I \models \neg F \land G$. Both deductions require considering multiple cases.

$$\frac{I \;\models\; F \leftrightarrow G}{I \;\models\; F \land G \quad | \quad I \;\not\models\; F \lor G} \qquad \frac{I \;\not\models\; F \leftrightarrow G}{I \;\models\; F \land \neg G \quad | \quad I \;\models\; \neg F \land G}$$

- Finally, a contradiction occurs when following the above proof rules results in the claim that an interpretation I both satisfies a formula F and does not satisfy F.

$$\frac{\begin{array}{l} I \;\models\; F \\ I \;\not\models\; F \end{array}}{I \;\models\; \bot}$$

Before explaining proofs in more detail, let us see several examples.

Example 1.7. To prove that the formula

$$F:\; P \land Q \;\rightarrow\; P \lor \neg Q$$

is valid, assume that it is invalid and derive a contradiction. Thus, assume that there is a falsifying interpretation I of F (such that $I \not\models F$). Then,

1. $I \;\not\models\; P \land Q \;\rightarrow\; P \lor \neg Q$ assumption
2. $I \;\models\; P \land Q$ by 1 and semantics of $\rightarrow$
3. $I \;\not\models\; P \lor \neg Q$ by 1 and semantics of $\rightarrow$
4. $I \;\models\; P$ by 2 and semantics of $\land$
5. $I \;\models\; Q$ by 2 and semantics of $\land$
6. $I \;\not\models\; P$ by 3 and semantics of $\lor$
7. $I \;\not\models\; \neg Q$ by 3 and semantics of $\lor$
8. $I \;\models\; Q$ by 7 and semantics of $\neg$

Lines 4 and 6 contradict each other, so that our assumption must be wrong: F is actually valid.

We can end the proof as soon as we have a contradiction. For example,

1. $I \;\not\models\; P \land Q \;\rightarrow\; P \lor \neg Q$ assumption
2. $I \;\models\; P \land Q$ by 1 and semantics of $\rightarrow$
3. $I \;\not\models\; P \lor \neg Q$ by 1 and semantics of $\rightarrow$
4. $I \;\models\; P$ by 2 and semantics of $\land$
5. $I \;\not\models\; P$ by 3 and semantics of $\lor$

This argument is sufficient because a contradiction already exists. In other words, the discovered contradiction closes the one branch of the proof. We sometimes note the contradiction explicitly in the proof:

6. $I \;\models\; \bot$ 4 and 5 are contradictory

■

Example 1.8. To prove that the formula

$$F:\ (P \rightarrow Q) \land (Q \rightarrow R)\ \rightarrow\ (P \rightarrow R)$$

is valid, assume otherwise and derive a contradiction:

1.	$I \not\models F$	assumption
2.	$I \models (P \rightarrow Q)\ \land\ (Q \rightarrow R)$	by 1 and semantics of →
3.	$I \not\models P \rightarrow R$	by 1 and semantics of →
4.	$I \models P$	by 3 and semantics of →
5.	$I \not\models R$	by 3 and semantics of →
6.	$I \models P \rightarrow Q$	by 2 and semantics of ∧
7.	$I \models Q \rightarrow R$	by 2 and semantics of ∧

There are two cases to consider from 6. In the first case,

8*a*.	$I \not\models P$	by 6 and semantics of →
9*a*.	$I \models \bot$	4 and 8*a* are contradictory

In the second case,

8*b*.	$I \models Q$	by 6 and semantics of →

Now there are two more cases from 7. In the first case,

9*ba*.	$I \not\models Q$	by 7 and semantics of →
10*ba*.	$I \models \bot$	8*b* and 9*ba* are contradictory

In the second case,

9*bb*.	$I \models R$	by 7 and semantics of →
10*bb*.	$I \models \bot$	5 and 9*bb* are contradictory

All three branches of the proof are closed: F is valid. ■

We introduce vocabulary for discussing semantic proofs. The reader need not memorize these terms now; just refer to them as they are used. A **line** $L : I \models F$ or $L : I \not\models F$ is a single statement in the proof, sometimes labeled as in the examples. A line L is a **direct descendant** of a **parent** M if L is directly below M in the proof. L is a **descendant** of M if M is L itself, if L is a direct descendant of M, or if the parent of L is a descendant of M (in other words, *descendant* is the reflexive and transitive closure of *direct descendant*). M is an **ancestor** of L if L is a descendant of M. Several proof rules — the second conjunction rule, the first disjunction rule, the first implication rule, and both rules for iff — produce a fork in the argument, as the last example shows. A proof thus evolves as a tree rather than linearly. A **branch** of the tree is a sequence of lines descending from the root. A branch is **closed** if it contains a contradiction, either explicitly as $I \models \bot$ or implicitly as $I \models G$

and $I \not\models G$ for some formula G. Otherwise, the branch is **open**. A semantic argument is **finished** when no more proof rules are applicable. It is a proof of the validity of F if every branch is closed; otherwise, each open branch describes a falsifying interpretation of F.

While the given proof rules are (theoretically) sufficient, **derived** proof rules can make proofs more concise.

Example 1.9. The derived rule of **modus ponens** simplifies the proof of Example 1.8. The rule is the following:

$$\begin{array}{l} I \models F \\ I \models F \rightarrow G \\ \hline I \models G \end{array}$$

In words, from $I \models F$ and $I \models F \rightarrow G$, deduce $I \models G$.

Using this rule, let us simplify the proof of the validity of

$$F:\ (P \rightarrow Q) \land (Q \rightarrow R) \ \rightarrow\ (P \rightarrow R)\ .$$

We assume that it is invalid and try to derive a contradiction.

1.	$I \not\models F$	assumption
2.	$I \models (P \rightarrow Q)\ \land\ (Q \rightarrow R)$	by 1 and semantics of →
3.	$I \not\models P \rightarrow R$	by 1 and semantics of →
4.	$I \models P$	by 3 and semantics of →
5.	$I \not\models R$	by 3 and semantics of →
6.	$I \models P \rightarrow Q$	by 2 and semantics of $\land$
7.	$I \models Q \rightarrow R$	by 2 and semantics of $\land$
8.	$I \models Q$	by 4, 6, and *modus ponens*
9.	$I \models R$	by 8, 7, and *modus ponens*
10.	$I \models \bot$	5 and 9 are contradictory

This proof has only one branch. ■

The truth-table and semantic methods can be used to check satisfiability. For example, the truth table of Example 1.6 can be extended to show that

$$\neg F:\ \neg(P \lor Q\ \rightarrow\ P \land Q)$$

is satisfiable:

P	Q	$P \lor Q$	$P \land Q$	F	$\neg F$
0	0	0	0	1	0
0	1	1	0	0	1
1	0	1	0	0	1
1	1	1	1	1	0

The second and third rows represent satisfying interpretations of $\neg F$. Additionally, the semantic argument in the following example shows that

$$G:\ \neg(P \lor Q\ \rightarrow\ P \land Q)$$

is satisfied by the discovered interpretation I, and thus that G is satisfiable.

Example 1.10. To prove that the formula

$$F:\ P \lor Q\ \rightarrow\ P \land Q$$

is valid, assume that F is invalid; then there is an interpretation I such that $I \models \neg F$:

$$\begin{array}{lll} 1. & I\ \not\models\ P \lor Q\ \rightarrow\ P \land Q & \text{assumption} \\ 2. & I\ \models\ P \lor Q & \text{by 1 and semantics of } \rightarrow \\ 3. & I\ \not\models\ P \land Q & \text{by 1 and semantics of } \rightarrow \end{array}$$

We have two choices to make. By 2 and the semantics of disjunction, either P or Q must be true. By 3 and the semantics of conjunction, either P or Q must be false. So there are two options: either P is true and Q is false, or P is false and Q is true. We choose P to be true and Q to be false. Then,

$$\begin{array}{lll} 4a. & I\ \models\ P & \text{by 2 and semantics of } \lor \\ 5a. & I\ \not\models\ Q & \text{by 3 and semantics of } \land \end{array}$$

The only subformulae of P and Q are themselves, so the table is complete. Yet we did not derive a contradiction. In fact, we found the interpretation

$$I:\ \{P \mapsto \mathsf{true},\ Q \mapsto \mathsf{false}\}$$

for which $I \models \neg F$. Therefore, F is actually invalid. The interpretation I : $\{P \mapsto \mathsf{true},\ Q \mapsto \mathsf{false}\}$ is a falsifying interpretation.

If our choice had resulted in a contradiction, then we would have had to try the other choice for P and Q, in which P is false and Q is true. In general, we stop either when we have found an interpretation or when we have closed every branch. ■

1.4 Equivalence and Implication

Just as satisfiability and validity are important properties of PL formulae, **equivalence** and **implication** are important properties of pairs of formulae. Two formulae F_1 and F_2 are **equivalent** if they evaluate to the same truth value under all interpretations I. That is, for all interpretations I, $I \models F_1$ iff $I \models F_2$. Another way to state the equivalence of F_1 and F_2 is to assert the validity of the formula $F_1 \leftrightarrow F_2$. We write $F_1 \Leftrightarrow F_2$ when F_1 and F_2 are equivalent. $F_1 \Leftrightarrow F_2$ is *not* a formula; it simply abbreviates the statement "F_1 and F_2 are equivalent."

We use the last characterization to prove that two formulae are equivalent.

Example 1.11. To prove that

$$P \;\Leftrightarrow\; \neg\neg P\ ,$$

we prove that

$$P \;\leftrightarrow\; \neg\neg P$$

is valid via a truth table:

P	$\neg P$	$\neg\neg P$	$P \;\leftrightarrow\; \neg\neg P$
0	1	0	1
1	0	1	1

■

Example 1.12. To prove

$$P \rightarrow Q \;\Leftrightarrow\; \neg P \vee Q\ ,$$

we prove that

$$F:\ P \rightarrow Q \;\leftrightarrow\; \neg P \vee Q$$

is valid via a truth table:

P	Q	$P \rightarrow Q$	$\neg P$	$\neg P \vee Q$	F
0	0	1	1	1	1
0	1	1	1	1	1
1	0	0	0	0	1
1	1	1	0	1	1

■

Formula F_1 **implies** formula F_2 if $I \models F_2$ for every interpretation I such that $I \models F_1$. Another way to state that F_1 implies F_2 is to assert the validity of the formula $F_1 \rightarrow F_2$. We write $F_1 \Rightarrow F_2$ when F_1 implies F_2. Do not confuse the *implication* $F_1 \Rightarrow F_2$, which asserts the validity of $F_1 \rightarrow F_2$, with the *PL formula* $F_1 \rightarrow F_2$, which is constructed using the logical operator $\rightarrow$. $F_1 \Rightarrow F_2$ is *not* a formula.

As with equivalences, we use the validity characterization to prove implications.

Example 1.13. To prove that

$$R \wedge (\neg R \vee P) \;\Rightarrow\; P\ ,$$

we prove that

$$F : \; R \wedge (\neg R \vee P) \; \rightarrow \; P$$

is valid via a semantic argument. Suppose F is not valid; then there exists an interpretation I such that $I \not\models F$:

$$\begin{array}{lll}
1. & I \not\models F & \text{assumption} \\
2. & I \models R \wedge (\neg R \vee P) & \text{by 1 and semantics of } \rightarrow \\
3. & I \not\models P & \text{by 1 and semantics of } \rightarrow \\
4. & I \models R & \text{by 2 and semantics of } \wedge \\
5. & I \models \neg R \vee P & \text{by 2 and semantics of } \wedge
\end{array}$$

There are two cases to consider. In the first case,

$$\begin{array}{lll}
6a. & I \models \neg R & \text{by 5 and semantics of } \vee \\
7a. & I \models \bot & \text{4 and } 6a \text{ are contradictory}
\end{array}$$

In the second case,

$$\begin{array}{lll}
6b. & I \models P & \text{by 5 and semantics of } \vee \\
7b. & I \models \bot & \text{3 and } 6b \text{ are contradictory}
\end{array}$$

Thus, our assumption that $I \not\models F$ is wrong, and F is valid. ■

1.5 Substitution

Substitution is a syntactic operation on formulae with significant semantic consequences. It allows us to prove the validity of entire sets of formulae via **formula templates**. It is also an essential tool for manipulating formulae throughout the text.

A **substitution** σ is a mapping from formulae to formulae:

$$\sigma : \; \{F_1 \mapsto G_1, \; \ldots, \; F_n \mapsto G_n\} \; .$$

The **domain** of σ, $\mathsf{domain}(\sigma)$, is

$$\mathsf{domain}(\sigma) : \; \{F_1, \ldots, F_n\} \; ,$$

while the **range** $\mathsf{range}(\sigma)$ is

$$\mathsf{range}(\sigma) : \; \{G_1, \ldots, G_n\} \; .$$

The application of a substitution σ to a formula F, $F\sigma$, replaces each occurrence of a formula F_i in the domain of σ with its corresponding formula G_i in the range of σ. Replacements occur all at once. We remove any ambiguity by establishing that when both subformulae F_j and F_k are in the domain of σ, and F_k is a strict subformula of F_j, then the larger subformula F_j is replaced by the corresponding formula G_j. An example clarifies this statement.

Example 1.14. Consider formula

$$F:\ P \wedge Q \ \rightarrow\ P \vee \neg Q$$

and substitution

$$\sigma:\ \{P \ \mapsto\ R,\quad P \wedge Q \ \mapsto\ P \rightarrow Q\}\ .$$

Then

$$F\sigma:\ (P \rightarrow Q)\ \rightarrow\ R \vee \neg Q\ ,$$

where the antecedent $P \wedge Q$ of F is replaced by $P \rightarrow Q$, and the P of the consequent is replaced by R. Moreover,

$$F\sigma\ \neq\ R \wedge Q\ \rightarrow\ R \vee \neg Q$$

by our convention. ■

A **variable substitution** is a substitution in which the domain consists only of propositional variables.

One notation is useful when working with substitutions. When we write $F[F_1, \ldots, F_n]$, we mean that formula F can have formulae F_i, $i = 1, \ldots, n$, as subformulae. If σ is $\{F_1 \mapsto G_1, \ldots, F_n \mapsto G_n\}$, then

$$F[F_1, \ldots, F_n]\sigma:\ F[G_1, \ldots, G_n]\ .$$

In the formula of Example 1.14, writing

$$F[P,\ P \wedge Q]\sigma:\ F[R,\ P \rightarrow Q]$$

emphasizes that subformulae P and $P \wedge Q$ of F are replaced by formulae R and $P \rightarrow Q$, respectively.

Two interesting semantic consequences can be derived from substitution. Proposition 1.15 states that substituting subformulae F_i of F with corresponding equivalent subformulae G_i results in an equivalent formula F'.

Proposition 1.15 (Substitution of Equivalent Formulae). *Consider substitution*

$$\sigma:\ \{F_1 \mapsto G_1, \ldots, F_n \mapsto G_n\}$$

such that for each i, $F_i \Leftrightarrow G_i$. Then $F \Leftrightarrow F\sigma$.

Example 1.16. Consider applying substitution

$$\sigma:\ \{P \rightarrow Q\ \mapsto\ \neg P \vee Q\}$$

to

$$F : \; (P \rightarrow Q) \; \rightarrow \; R \, .$$

Since $P \rightarrow Q \;\Leftrightarrow\; \neg P \vee Q$, the formula

$$F\sigma : \; (\neg P \vee Q) \; \rightarrow \; R$$

is equivalent to F. ■

Proposition 1.17 asserts that proving the validity of a PL formula F actually proves the validity of an infinite set of formulae: those formulae that can be derived from F via variable substitutions.

Proposition 1.17 (Valid Template). *If F is valid and $G = F\sigma$ for some variable substitution σ, then G is valid.*

Example 1.18. In Example 1.12, we proved that $P \rightarrow Q$ is equivalent to $\neg P \vee Q$:

$$F : \; (P \rightarrow Q) \; \leftrightarrow \; (\neg P \vee Q)$$

is valid. The validity of F implies that every formula of the form $F_1 \rightarrow F_2$ is equivalent to $\neg F_1 \vee F_2$, for arbitrary subformulae F_1 and F_2. ■

Finally, it is often useful to compute the **composition** of substitutions. Given substitutions σ_1 and σ_2, the idea is to compute substitution σ such that $F\sigma_1\sigma_2 = F\sigma$ for any F. Compute $\sigma_1\sigma_2$ as follows:

1. apply σ_2 to each formula of the range of σ_1, and add the results to σ;
2. if F_i of $F_i \mapsto G_i$ appears in the domain of σ_2 but *not* in the domain of σ_1, then add $F_i \mapsto G_i$ to σ.

Example 1.19. Compute the composition of substitutions

$$\sigma_1\sigma_2 : \; \{P \;\mapsto\; R, \; P \wedge Q \;\mapsto\; P \rightarrow Q\}\{P \;\mapsto\; S, \; S \;\mapsto\; Q\}$$

as follows:

$$\begin{aligned} &\{P \;\mapsto\; R\sigma_2, \; P \wedge Q \;\mapsto\; (P \rightarrow Q)\sigma_2, \; S \;\mapsto\; Q\} \\ &\quad = \{P \;\mapsto\; R, \; P \wedge Q \;\mapsto\; S \rightarrow Q, \; S \;\mapsto\; Q\} \end{aligned}$$

■

1.6 Normal Forms

A **normal form** of formulae is a syntactic restriction such that for every formula of the logic, there is an equivalent formula in the normal form. Three normal forms are particularly important for PL.

Negation normal form (**NNF**) requires that $\neg$, $\wedge$, and $\vee$ be the only connectives and that negations appear only in literals. Transforming a formula F to equivalent formula F' in NNF can be computed recursively using the following list of template equivalences:

$$\begin{aligned} \neg\neg F_1 &\Leftrightarrow F_1 \\ \neg\top &\Leftrightarrow \bot \\ \neg\bot &\Leftrightarrow \top \\ \neg(F_1 \wedge F_2) &\Leftrightarrow \neg F_1 \vee \neg F_2 \\ \neg(F_1 \vee F_2) &\Leftrightarrow \neg F_1 \wedge \neg F_2 \\ F_1 \rightarrow F_2 &\Leftrightarrow \neg F_1 \vee F_2 \\ F_1 \leftrightarrow F_2 &\Leftrightarrow (F_1 \rightarrow F_2) \ \wedge \ (F_2 \rightarrow F_1) \end{aligned}$$

When implementing the transformation, the equivalences should be applied left-to-right. The equivalences

$$\neg(F_1 \wedge F_2) \Leftrightarrow \neg F_1 \vee \neg F_2 \qquad\qquad \neg(F_1 \vee F_2) \Leftrightarrow \neg F_1 \wedge \neg F_2$$

are known as **De Morgan's Law**.

Propositions 1.15 and 1.17 justify that the result of applying the template equivalences to a formula produces an equivalent formula. The transitivity of equivalence justifies that this equivalence holds over any number of transformations: if $F \Leftrightarrow G$ and $G \Leftrightarrow H$, then $F \Leftrightarrow H$.

Example 1.20. To convert the formula

$$F : \ \neg(P \ \rightarrow \ \neg(P \wedge Q))$$

into NNF, apply the template equivalence

$$F_1 \rightarrow F_2 \ \Leftrightarrow \ \neg F_1 \vee F_2 \tag{1.1}$$

to produce

$$F' : \ \neg(\neg P \vee \neg(P \wedge Q)) \ .$$

Let us understand this "application" of the template equivalence in detail. First, apply variable substitution

$$\sigma_1 : \ \{F_1 \mapsto P, \ F_2 \mapsto \neg(P \wedge Q)\}$$

to the valid template formula of equivalence (1.1):

$$(F_1 \rightarrow F_2 \ \leftrightarrow \ \neg F_1 \vee F_2)\sigma_1 : \ P \ \rightarrow \ \neg(P \wedge Q) \ \leftrightarrow \ \neg P \vee \neg(P \wedge Q) \ .$$

Proposition 1.17 implies that the result is valid. Then construct substitution

$$\sigma_2 : \ \{P \ \rightarrow \ \neg(P \wedge Q) \ \mapsto \ \neg P \vee \neg(P \wedge Q)\} \ ,$$

and apply Proposition 1.15 to $F\sigma_2$ to yield that

$$F' : \neg(\neg P \vee \neg(P \wedge Q))$$

is equivalent to F. Subsequently, we shall not provide these details.

Continuing with the conversion to NNF, apply De Morgan's law

$$\neg(F_1 \vee F_2) \iff \neg F_1 \wedge \neg F_2$$

to produce

$$F'' : \neg\neg P \wedge \neg\neg(P \wedge Q)\ .$$

Apply

$$\neg\neg F_1 \iff F_1$$

twice to produce

$$F''' : P \wedge P \wedge Q\ ,$$

which is in NNF and equivalent to F. ■

A formula is in **disjunctive normal form** (**DNF**) if it is a disjunction of conjunctions of literals:

$$\bigvee_i \bigwedge_j \ell_{i,j} \quad \text{for literals } \ell_{i,j}\ .$$

To convert a formula F into an equivalent formula in DNF, transform F into NNF and then use the following table of template equivalences:

$$\begin{array}{lcl}(F_1 \vee F_2) \wedge F_3 & \iff & (F_1 \wedge F_3) \vee (F_2 \wedge F_3) \\ F_1 \wedge (F_2 \vee F_3) & \iff & (F_1 \wedge F_2) \vee (F_1 \wedge F_3)\end{array}$$

Again, when implementing the transformation, the equivalences should be applied left-to-right. The equivalences simply say that conjunction distributes over disjunction.

Example 1.21. To convert

$$F : (Q_1 \vee \neg\neg Q_2)\ \wedge\ (\neg R_1 \rightarrow R_2)$$

into DNF, first transform it into NNF

$$F' : (Q_1 \vee Q_2)\ \wedge\ (R_1 \vee R_2)\ ,$$

and then apply distributivity to obtain

$$F'' : (Q_1 \wedge (R_1 \vee R_2))\ \vee\ (Q_2 \wedge (R_1 \vee R_2))\ ,$$

and then distributivity twice again to produce

$$F''' : (Q_1 \wedge R_1)\ \vee\ (Q_1 \wedge R_2)\ \vee\ (Q_2 \wedge R_1)\ \vee\ (Q_2 \wedge R_2)\ .$$

F''' is in DNF and is equivalent to F. ■

The dual of DNF is **conjunctive normal form** (**CNF**). A formula in CNF is a conjunction of disjunctions of literals:

$$\bigwedge_i \bigvee_j \ell_{i,j} \quad \text{for literals } \ell_{i,j} \ .$$

Each inner block of disjunctions is called a **clause**. To convert a formula F into an equivalent formula in CNF, transform F into NNF and then use the following table of template equivalences:

$$\begin{array}{rcl} (F_1 \wedge F_2) \vee F_3 & \Leftrightarrow & (F_1 \vee F_3) \wedge (F_2 \vee F_3) \\ F_1 \vee (F_2 \wedge F_3) & \Leftrightarrow & (F_1 \vee F_2) \wedge (F_1 \vee F_3) \end{array}$$

Example 1.22. To convert

$$F : \ (Q_1 \wedge \neg\neg Q_2) \ \vee \ (\neg R_1 \rightarrow R_2)$$

into CNF, first transform F into NNF:

$$F' : \ (Q_1 \wedge Q_2) \ \vee \ (R_1 \vee R_2) \ .$$

Then apply distributivity to obtain

$$F'' : \ (Q_1 \vee R_1 \vee R_2) \ \wedge \ (Q_2 \vee R_1 \vee R_2) \ ,$$

which is in CNF and equivalent to F. ■

1.7 Decision Procedures for Satisfiability

Section 1.3 introduced the truth-table and semantic argument methods for determining the satisfiability of PL formulae. In this section, we study algorithms for *deciding* satisfiability (see Section 2.6 for a formal discussion of decidability). A **decision procedure** for satisfiability of PL formulae reports, after some finite amount of computation, whether a given PL formula F is satisfiable.

1.7.1 Simple Decision Procedures

The truth-table method immediately suggests a decision procedure: construct the full table, which has 2^n rows when F has n variables, and report whether the final column, representing F, has value 1 in any row.

The semantic argument method also suggests a decision procedure. The basic idea is to make sure that a proof rule is only applied to each line in the argument at most once. Because each deduction is simpler in construction than its premise, the constructed proof is of finite size (see Chapter 4 for a formal approach to proving this point). When the semantic argument is finished, report whether any branch is still open.

This simple description leaves out many details. Most importantly, when many lines exist to which one can apply proof rules, which line should be considered next? Different implementations of this decision, called **proof tactics**, result in different proof shapes and sizes. For example, one basic tactic is to apply proof rules with only one deduction before proof rules with multiple deductions to delay forks in the proof as long as possible.

Subsequent sections consider more sophisticated procedures that are the basis for modern satisfiability solvers.

1.7.2 Reconsidering the Truth-Table Method

In the naive decision procedure based on the truth-table method, the entire table is constructed. Actually, only one row need be considered at a time, making for a space efficient procedure. This idea is implemented in the following recursive algorithm for deciding the satisfiability of a PL formula F:

```
let rec SAT $F$ =
  if $F = \top$ then true
  else if $F = \bot$ then false
  else
    let $P$ = CHOOSE $\mathsf{vars}(F)$ in
    $(\text{SAT } F\{P \mapsto \top\}) \lor (\text{SAT } F\{P \mapsto \bot\})$
```

The notation "let rec SAT F =" declares SAT as a recursive function that takes one argument, a formula F. The notation "let P = CHOOSE $\mathsf{vars}(F)$ in" means that P's value in the subsequent text is the variable returned by the CHOOSE function. When applying the substitutions $F\{P \mapsto \top\}$ or $F\{P \mapsto \bot\}$, the template equivalences of Exercise 1.2 should be applied to simplify the result. Then the comparisons $F = \top$ and $F = \bot$ can be implemented as purely syntactic operations.

At each recursive step, if F is not yet $\top$ or $\bot$, a variable is chosen on which to branch. Each possibility for P is attempted if necessary. This algorithm returns true immediately upon finding a satisfying interpretation. Otherwise, if F is unsatisfiable, it eventually returns $\bot$. SAT may save branching on certain variables by simplifying intermediate formulae.

Example 1.23. Consider the formula

$$F: \ (P \to Q) \land P \land \neg Q \ .$$

To compute SAT F, choose a variable, say P, and recurse on the first case,

$$F\{P \mapsto \top\}: \ (\top \ \to \ Q) \land \top \land \neg Q \ ,$$

which simplifies to

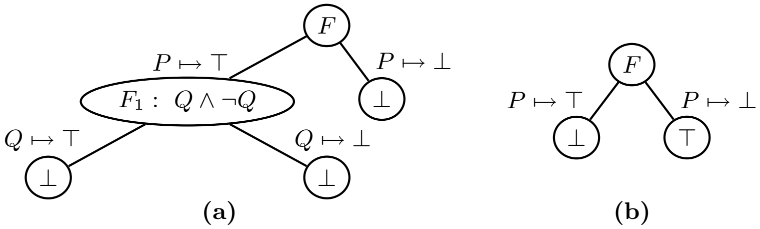

Fig. 1.1. Visualizing runs of SAT

$$F_1 :\; Q \wedge \neg Q \;.$$

Now try each of

$$F_1\{Q \mapsto \top\} \quad \text{and} \quad F_1\{Q \mapsto \bot\} \;.$$

Both simplify to $\bot$, so this branch ends without finding a satisfying interpretation.

Now try the other branch for P in F:

$$F\{P \mapsto \bot\} :\; (\bot \;\rightarrow\; Q) \wedge \bot \wedge \neg Q \;,$$

which simplifies to $\bot$. Thus, this branch also ends without finding a satisfying interpretation. Thus, F is unsatisfiable.

The run of SAT on F is visualized in Figure 1.1(**a**). ■

Example 1.24. Consider the formula

$$F :\; (P \rightarrow Q) \wedge \neg P \;.$$

To compute SAT F, choose a variable, say P, and recurse on the first case,

$$F\{P \mapsto \top\} :\; (\top \;\rightarrow\; Q) \wedge \neg\top \;,$$

which simplifies to $\bot$. Therefore, try

$$F\{P \mapsto \bot\} :\; (\bot \;\rightarrow\; Q) \wedge \neg\bot$$

instead, which simplifies to $\top$. Arbitrarily assigning a value to Q produces the following satisfying interpretation:

$$I :\; \{P \mapsto \mathsf{false},\; Q \mapsto \mathsf{true}\} \;.$$

The run of SAT on F is visualized in Figure 1.1(**b**). ■

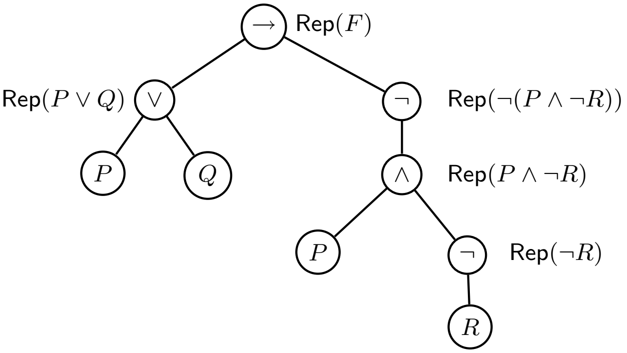

Fig. 1.2. Parse tree of $F:\ P \vee Q\ \rightarrow\ \neg(P \wedge \neg R)$ with representatives for subformulae

1.7.3 Conversion to an Equisatisfiable Formula in CNF

The next two decision procedures operate on PL formulae in CNF. The transformation suggested in Section 1.6 produces an equivalent formula that can be exponentially larger than the original formula: consider converting a formula in DNF into CNF. However, to decide the satisfiability of F, we need only examine a formula F' such that F and F' are **equisatisfiable**. F and F' are equisatisfiable when F is satisfiable iff F' is satisfiable.

We define a method for converting PL formula F to equisatisfiable PL formula F' in CNF that is at most a constant factor larger than F. The main idea is to introduce new propositional variables to represent the subformulae of F. The constructed formula F' includes extra clauses that assert that these new variables are equivalent to the subformulae that they represent.

Figure 1.2 visualizes the idea of the procedure. Each node of the "parse tree" of F represents a subformula G of F. With each node G is associated a representative propositional variable Rep(G). In the constructed formula F', each representative Rep(G) is asserted to be equivalent to the subformula G that it represents in such a way that the conjunction of all such assertions is in CNF. Finally, the representative Rep(F) of F is asserted to be true.

To obtain a small formula in CNF, each assertion of equivalence between Rep(G) and G refers at most to the children of G in the parse tree. How is this possible when a subformula may be arbitrarily large? The main trick is to refer to the representatives of G's children rather than the children themselves.

Let the "representative" function Rep : PL $\rightarrow \mathcal{V} \cup \{\top, \bot\}$ map PL formulae to propositional variables $\mathcal{V}$, $\top$, or $\bot$. In the general case, it is intended to map a formula F to its representative propositional variable P_F such that the truth value of P_F is the same as that of F. In other words, P_F provides a compact way of referring to F.

Let the "encoding" function En : PL $\rightarrow$ PL map PL formulae to PL formulae. En is intended to map a PL formula F to a PL formula F' in CNF that asserts that F's representative, P_F, is equivalent to F: "Rep(F) $\leftrightarrow F$".

As the base cases for defining Rep and En, define their behavior on $\top$, $\bot$, and propositional variables P:

$$\begin{array}{ll} \mathsf{Rep}(\top) = \top & \mathsf{En}(\top) = \top \\ \mathsf{Rep}(\bot) = \bot & \mathsf{En}(\bot) = \top \\ \mathsf{Rep}(P) = P & \mathsf{En}(P) = \top \end{array}$$

The representative of $\top$ is $\top$ itself, and the representative of $\bot$ is $\bot$ itself. Thus, $\mathsf{Rep}(\top) \leftrightarrow \top$ and $\mathsf{Rep}(\bot) \leftrightarrow \bot$ are both trivially valid, so $\mathsf{En}(\top)$ and $\mathsf{En}(\bot)$ are both $\top$. Finally, the representative of a propositional variable P is P itself; and again, $\mathsf{Rep}(P) \leftrightarrow P$ is trivially valid so that $\mathsf{En}(P)$ is $\top$.

For the inductive case, F is a formula other than an atom, so define its representative as a unique propositional variable P_F:

$$\mathsf{Rep}(F) = P_F \ .$$

En then asserts the equivalence of F and P_F as a CNF formula. On conjunction, define

```
En($F_1 \wedge F_2$) =
  let $P$ = Rep($F_1 \wedge F_2$) in
  $(\neg P \vee \mathsf{Rep}(F_1)) \ \wedge \ (\neg P \vee \mathsf{Rep}(F_2)) \ \wedge \ (\neg \mathsf{Rep}(F_1) \vee \neg \mathsf{Rep}(F_2) \vee P)$
```

The returned formula

$$(\neg P \vee \mathsf{Rep}(F_1)) \ \wedge \ (\neg P \vee \mathsf{Rep}(F_2)) \ \wedge \ (\neg \mathsf{Rep}(F_1) \vee \neg \mathsf{Rep}(F_2) \vee P)$$

is in CNF and is equivalent to

$$\mathsf{Rep}(F_1 \wedge F_2) \ \leftrightarrow \ \mathsf{Rep}(F_1) \wedge \mathsf{Rep}(F_2) \ .$$

In detail, the first two clauses

$$(\neg P \vee \mathsf{Rep}(F_1)) \ \wedge \ (\neg P \vee \mathsf{Rep}(F_2))$$

together assert

$$P \ \rightarrow \ \mathsf{Rep}(F_1) \wedge \mathsf{Rep}(F_2)$$

(since, for example, $\neg P \vee \mathsf{Rep}(F_1)$ is equivalent to $P \rightarrow \mathsf{Rep}(F_1)$), while the final clause asserts

$$\mathsf{Rep}(F_1) \wedge \mathsf{Rep}(F_2) \ \rightarrow \ P \ .$$

Notice the application of Rep to F_1 and F_2. As mentioned above, it is the trick to producing a small CNF formula.

On negation, $\mathsf{En}(\neg F)$ returns a formula equivalent to $\mathsf{Rep}(\neg F) \ \leftrightarrow \ \neg \mathsf{Rep}(F)$:

```
En($\neg F$) =
  let $P$ = Rep($\neg F$) in
  $(\neg P \vee \neg \mathsf{Rep}(F)) \ \wedge \ (P \vee \mathsf{Rep}(F))$
```

En is defined for $\vee$, $\rightarrow$, and $\leftrightarrow$ as well:

$$\begin{array}{l}
\mathsf{En}(F_1 \vee F_2) = \\
\quad \texttt{let } P = \mathsf{Rep}(F_1 \vee F_2) \texttt{ in} \\
\quad (\neg P \vee \mathsf{Rep}(F_1) \vee \mathsf{Rep}(F_2)) \ \wedge\ (\neg \mathsf{Rep}(F_1) \vee P) \ \wedge\ (\neg \mathsf{Rep}(F_2) \vee P)
\end{array}$$

$$\begin{array}{l}
\mathsf{En}(F_1 \rightarrow F_2) = \\
\quad \texttt{let } P = \mathsf{Rep}(F_1 \rightarrow F_2) \texttt{ in} \\
\quad (\neg P \vee \neg \mathsf{Rep}(F_1) \vee \mathsf{Rep}(F_2)) \ \wedge\ (\mathsf{Rep}(F_1) \vee P) \ \wedge\ (\neg \mathsf{Rep}(F_2) \vee P)
\end{array}$$

$$\begin{array}{l}
\mathsf{En}(F_1 \leftrightarrow F_2) = \\
\quad \texttt{let } P = \mathsf{Rep}(F_1 \leftrightarrow F_2) \texttt{ in} \\
\quad (\neg P \vee \neg \mathsf{Rep}(F_1) \vee \mathsf{Rep}(F_2)) \ \wedge\ (\neg P \vee \mathsf{Rep}(F_1) \vee \neg \mathsf{Rep}(F_2)) \\
\quad\ \ \wedge\ (P \vee \neg \mathsf{Rep}(F_1) \vee \neg \mathsf{Rep}(F_2)) \ \wedge\ (P \vee \mathsf{Rep}(F_1) \vee \mathsf{Rep}(F_2))
\end{array}$$

Having defined En, let us construct the full CNF formula that is equisatisfiable to F. If S_F is the set of all subformulae of F (including F itself), then

$$F' : \ \mathsf{Rep}(F) \ \wedge \bigwedge_{G \in S_F} \mathsf{En}(G)$$

is in CNF and is equisatisfiable to F. The second main conjunct asserts the equivalences between all subformulae of F and their corresponding representatives. $\mathsf{Rep}(F)$ asserts that F's representative, and thus F itself (according to the second conjunct), is true.

If F has size n, where each instance of a logical connective or a propositional variable contributes one unit of size, then F' has size at most $30n + 2$. The size of F' is thus linear in the size of F. The number of symbols in the formula returned by $\mathsf{En}(F_1 \leftrightarrow F_2)$, which incurs the largest expansion, is 29. Up to one additional conjunction is also required per symbol of F. Finally, two extra symbols are required for asserting that $\mathsf{Rep}(F)$ is true.

Example 1.25. Consider formula

$$F : \ (Q_1 \wedge Q_2) \ \vee \ (R_1 \wedge R_2) \ ,$$

which is in DNF. To convert it to CNF, we collect its subformulae

$$S_F : \ \{Q_1,\ Q_2,\ Q_1 \wedge Q_2,\ R_1,\ R_2,\ R_1 \wedge R_2,\ F\}$$

and compute

$$\begin{aligned}
\mathsf{En}(Q_1) &= \top \\
\mathsf{En}(Q_2) &= \top \\
\mathsf{En}(Q_1 \wedge Q_2) &= (\neg P_{(Q_1 \wedge Q_2)} \vee Q_1) \ \wedge\ (\neg P_{(Q_1 \wedge Q_2)} \vee Q_2) \\
&\qquad \wedge\ (\neg Q_1 \vee \neg Q_2 \vee P_{(Q_1 \wedge Q_2)})
\end{aligned}$$

$$\begin{aligned}
\mathsf{En}(R_1) &= \top \\
\mathsf{En}(R_2) &= \top \\
\mathsf{En}(R_1 \land R_2) &= (\neg P_{(R_1 \land R_2)} \lor R_1) \;\land\; (\neg P_{(R_1 \land R_2)} \lor R_2) \\
&\quad \land\; (\neg R_1 \lor \neg R_2 \lor P_{(R_1 \land R_2)})
\end{aligned}$$

$$\begin{aligned}
\mathsf{En}(F) = {} & (\neg P_{(F)} \lor P_{(Q_1 \land Q_2)} \lor P_{(R_1 \land R_2)}) \\
& \land\; (\neg P_{(Q_1 \land Q_2)} \lor P_{(F)}) \\
& \land\; (\neg P_{(R_1 \land R_2)} \lor P_{(F)})
\end{aligned}$$

Then

$$F' :\; P_{(F)} \;\land \bigwedge_{G \in S_F} \mathsf{En}(G)$$

is equisatisfiable to F and is in CNF. ■

1.7.4 The Resolution Procedure

The next decision procedure that we consider is based on **resolution** and applies only to PL formulae in CNF. Therefore, the procedure of Section 1.7.3 must first be applied to the given PL formula if it is not already in CNF.

Resolution follows from the following observation of any PL formula F in CNF: to satisfy clauses $C_1[P]$ and $C_2[\neg P]$ that share variable P but disagree on its value, either the rest of C_1 or the rest of C_2 must be satisfied. Why? If P is true, then a literal other than $\neg P$ in C_2 must be satisfied; while if P is false, then a literal other than P in C_1 must be satisfied. Therefore, the clause $C_1[\bot] \lor C_2[\bot]$, simplified according to the template equivalences of Exercise 1.2, can be added as a conjunction to F to produce an equivalent formula still in CNF.

Clausal resolution is stated as the following proof rule:

$$\frac{C_1[P] \qquad C_2[\neg P]}{C_1[\bot] \lor C_2[\bot]}$$

From the two clauses of the premise, deduce the new clause, called the **resolvent**.

If ever $\bot$ is deduced via resolution, F must be unsatisfiable since $F \land \bot$ is unsatisfiable. Otherwise, if every possible resolution produces a clause that is already known, then F must be satisfiable.

Example 1.26. The CNF of $(P \to Q) \land P \land \neg Q$ is the following:

$$F :\; (\neg P \lor Q) \;\land\; P \;\land\; \neg Q\ .$$

From resolution

$$\frac{(\neg P \lor Q) \qquad P}{Q}\ ,$$

construct

$$F_1 : \; (\neg P \vee Q) \;\wedge\; P \;\wedge\; \neg Q \;\wedge\; Q \; .$$

From resolution

$$\frac{\neg Q \qquad Q}{\bot} \; ,$$

deduce that F, and thus the original formula, is unsatisfiable. ■

Example 1.27. Consider the formula

$$F : \; (\neg P \vee Q) \;\wedge\; \neg Q \; .$$

The one possible resolution

$$\frac{(\neg P \vee Q) \qquad \neg Q}{\neg P}$$

yields

$$F_1 : \; (\neg P \vee Q) \;\wedge\; \neg Q \;\wedge\; \neg P \; .$$

Since no further resolutions are possible, F is satisfiable. Indeed,

$$I : \; \{P \mapsto \mathsf{false}, \; Q \mapsto \mathsf{false}\}$$

is a satisfying interpretation. A CNF formula that does not contain the clause $\bot$ and to which no more resolutions can be applied represents all possible satisfying interpretations. ■

1.7.5 DPLL

Modern satisfiability procedures for propositional logic are based on the Davis-Putnam-Logemann-Loveland algorithm (**DPLL**), which combines the space-efficient procedure of Section 1.7.2 with a restricted form of resolution. We review in this section the basic algorithm. Much research in the past decade has advanced the state-of-the-art considerably.

Like the resolution procedure, DPLL operates on PL formulae in CNF. But again, as the procedure decides satisfiability, we can apply the conversion procedure of Section 1.7.3 to produce a small equisatisfiable CNF formula.

As in the procedure SAT, DPLL attempts to construct an interpretation of F; failing to do so, it reports that the given formula is unsatisfiable. Rather than relying solely on enumerating possibilities, however, DPLL applies a restricted form of resolution to gain some deductive power. The process of applying this restricted resolution as much as possible is called **Boolean constraint propagation** (**BCP**).

BCP is based on **unit resolution**. Unit resolution operates on two clauses. One clause, called the **unit clause**, consists of a single literal ℓ ($\ell = P$ or $\ell = \neg P$ for some propositional variable P). The second clause contains the negation of ℓ: $C[\neg\ell]$. Then unit resolution is the deduction

$$\frac{\ell \qquad C[\neg\ell]}{C[\bot]} \; .$$

Unlike with full resolution, the literals of the resolvent are a subset of the literals of the second clause. Hence, the resolvent replaces the second clause.

Example 1.28. In the formula

$$F : \; (P) \;\wedge\; (\neg P \vee Q) \;\wedge\; (R \vee \neg Q \vee S) \; ,$$

(P) is a unit clause. Therefore, applying unit resolution

$$\frac{P \qquad (\neg P \vee Q)}{Q}$$

produces

$$F' : \; (Q) \;\wedge\; (R \vee \neg Q \vee S) \; .$$

Applying unit resolution again

$$\frac{Q \qquad R \vee \neg Q \vee S}{R \vee S}$$

produces

$$F'' : \; (R \vee S) \; ,$$

ending this round of BCP. ■

The implementation of DPLL is structurally similar to SAT, except that it begins by applying BCP:

```
let rec DPLL $F$ =
  let $F'$ = BCP $F$ in
  if $F' = \top$ then true
  else if $F' = \bot$ then false
  else
    let $P$ = CHOOSE vars($F'$) in
    (DPLL $F'\{P \mapsto \top\}$) ∨ (DPLL $F'\{P \mapsto \bot\}$)
```

As in SAT, intermediate formulae are simplified according to the template equivalences of Exercise 1.2.

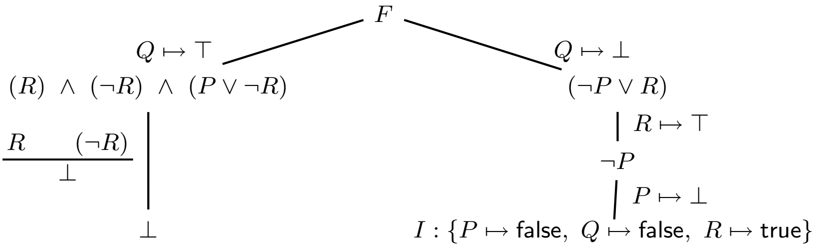

Fig. 1.3. Visualization of Example 1.30

One easy optimization is the following: if variable P appears only positively or only negatively in F, it should not be chosen by CHOOSE $\mathsf{vars}(F')$. P appears only positively when every P-literal is just P; P appears only negatively when every P-literal is $\neg P$. In both cases, F is equisatisfiable to the formula F' constructed by removing all clauses containing an instance of P. Therefore, these clauses do not contribute to BCP. When only such variables remain, the formula must be satisfiable: a full interpretation can be constructed by setting each variable's value based on whether it appears only positively (true) or only negatively (false).

The values to which propositional variables are set on the path to a solution can be recorded so that DPLL can return a satisfying interpretation if one exists, rather than just `true`.

Example 1.29. Consider the formula

$$F : \ (P) \ \wedge \ (\neg P \vee Q) \ \wedge \ (R \vee \neg Q \vee S) \ .$$

On the first level of recursion, DPLL recognizes the unit clause (P) and applies the BCP steps from Example 1.28, resulting in the formula

$$F'' : \ R \vee S \ .$$

The unit resolutions correspond to the partial interpretation

$$\{P \mapsto \mathsf{true},\ Q \mapsto \mathsf{true}\} \ .$$

Only positively occurring variables remain, so F is satisfiable. In particular,

$$\{P \mapsto \mathsf{true},\ Q \mapsto \mathsf{true},\ R \mapsto \mathsf{true},\ S \mapsto \mathsf{true}\}$$

is a satisfying interpretation of F.

Branching was not required in this example. ■

Example 1.30. Consider the formula

$$F : \ (\neg P \vee Q \vee R) \ \wedge \ (\neg Q \vee R) \ \wedge \ (\neg Q \vee \neg R) \ \wedge \ (P \vee \neg Q \vee \neg R) \ .$$

On the first level of recursion, DPLL must branch. Branching on Q or R will result in unit clauses; choose Q.

Then

$$F\{Q \mapsto \top\} : \ (R) \ \wedge \ (\neg R) \ \wedge \ (P \vee \neg R) \ .$$

The unit resolution

$$\frac{R \qquad (\neg R)}{\bot}$$

finishes this branch.

On the other branch,

$$F\{Q \mapsto \bot\} : \ (\neg P \vee R) \ .$$

P appears only negatively, and R appears only positively, so the formula is satisfiable. In particular, F is satisfied by interpretation

$$I : \ \{P \mapsto \mathsf{false}, \ Q \mapsto \mathsf{false}, \ R \mapsto \mathsf{true}\} \ .$$

This run of DPLL is visualized in Figure 1.3. ■

1.8 Summary

This chapter introduces propositional logic (PL). It covers:

- Its *syntax*. How one constructs a PL formula. Propositional variables, atoms, literals, logical connectives.
- Its *semantics*. What a PL formula means. Truth values true and false. Interpretations. Truth-table definition, inductive definition.
- *Satisfiability* and *validity*. Whether a PL formula evaluates to true under any or all interpretations. Duality of satisfiability and validity, truth-table method, semantic argument method.
- *Equivalence* and *implication*. Whether two formulae always evaluate to the same truth value under every interpretation. Whether under any interpretation, if one formula evaluates to true, the other also evaluates to true. Reduction to validity.
- *Substitution*, which is a tool for manipulating formulae and making general claims. Substitution of equivalent formulae. Valid templates.
- *Normal forms*. A normal form is a set of syntactically restricted formulae such that every PL formula is equivalent to some member of the set.
- *Decision procedures for satisfiability*. Truth-table method, SAT, resolution procedure, DPLL. Transformation to equisatisfiable CNF formula.

PL is an important logic with applications in software and hardware design and analysis, knowledge representation, combinatorial optimization, and complexity theory, to name a few. Although relatively simple, the Boolean structure that is central to PL is often a main source of complexity in applications of the algorithmic reasoning that is the focus of Part II. Exercise 8.1 explores this point in more depth.

Besides being an important logic in its own right, PL serves to introduce the main concepts that are important throughout the book, in particular syntax, semantics, and satisfiability and validity. Chapter 2 presents first-order logic by building on the concepts of this chapter.

Bibliographic Remarks

For a complete and concise presentation of propositional logic, see Smullyan's text *First-Order Logic* [87]. The semantic argument method is similar to Smullyan's tableau method.

The DPLL algorithm is based on work by Davis and Putnam, presented in [26], and by Davis, Logemann, and Loveland, presented in [25].

Exercises

1.1 (PL validity & satisfiability). For each of the following PL formulae, identify whether it is valid or not. If it is valid, prove it with a truth table or semantic argument; otherwise, identify a falsifying interpretation. Recall our conventions for operator precedence and associativity from Section 1.1.

(*a*) $P \land Q \rightarrow P \rightarrow Q$
(*b*) $(P \rightarrow Q) \lor P \land \neg Q$
(*c*) $(P \rightarrow Q \rightarrow R) \rightarrow P \rightarrow R$
(*d*) $(P \rightarrow Q \lor R) \rightarrow P \rightarrow R$
(*e*) $\neg(P \land Q) \rightarrow R \rightarrow \neg R \rightarrow Q$
(*f*) $P \land Q \lor \neg P \lor (\neg Q \rightarrow \neg P)$
(*g*) $(P \rightarrow Q \rightarrow R) \rightarrow \neg R \rightarrow \neg Q \rightarrow \neg P$
(*h*) $(\neg R \rightarrow \neg Q \rightarrow \neg P) \rightarrow P \rightarrow Q \rightarrow R$

1.2 (Template equivalences). Use the truth table or semantic argument method to prove the following template equivalences.

(*a*) $\top \Leftrightarrow \neg\bot$
(*b*) $\bot \Leftrightarrow \neg\top$
(*c*) $\neg\neg F \Leftrightarrow F$
(*d*) $F \land \top \Leftrightarrow F$
(*e*) $F \land \bot \Leftrightarrow \bot$
(*f*) $F \land F \Leftrightarrow F$

(*g*) $F \lor \top \;\Leftrightarrow\; \top$

(*h*) $F \lor \bot \;\Leftrightarrow\; F$

(*i*) $F \lor F \;\Leftrightarrow\; F$

(*j*) $F \rightarrow \top \;\Leftrightarrow\; \top$

(*k*) $F \rightarrow \bot \;\Leftrightarrow\; \neg F$

(*l*) $\top \rightarrow F \;\Leftrightarrow\; F$

(*m*) $\bot \rightarrow F \;\Leftrightarrow\; \top$

(*n*) $\top \;\leftrightarrow\; F \;\Leftrightarrow\; F$

(*o*) $\bot \;\leftrightarrow\; F \;\Leftrightarrow\; \neg F$

(*p*) $\neg(F_1 \land F_2) \;\Leftrightarrow\; \neg F_1 \lor \neg F_2$

(*q*) $\neg(F_1 \lor F_2) \;\Leftrightarrow\; \neg F_1 \land \neg F_2$

(*r*) $F_1 \rightarrow F_2 \;\Leftrightarrow\; \neg F_1 \lor F_2$

(*s*) $F_1 \rightarrow F_2 \;\Leftrightarrow\; \neg F_2 \rightarrow \neg F_1$

(*t*) $\neg(F_1 \rightarrow F_2) \;\Leftrightarrow\; F_1 \land \neg F_2$

(*u*) $(F_1 \lor F_2) \;\land\; F_3 \;\Leftrightarrow\; (F_1 \land F_3) \;\lor\; (F_2 \land F_3)$

(*v*) $(F_1 \land F_2) \;\lor\; F_3 \;\Leftrightarrow\; (F_1 \lor F_3) \;\land\; (F_2 \lor F_3)$

(*w*) $(F_1 \rightarrow F_3) \;\land\; (F_2 \rightarrow F_3) \;\Leftrightarrow\; F_1 \lor F_2 \;\rightarrow\; F_3$

(*x*) $(F_1 \rightarrow F_2) \;\land\; (F_1 \rightarrow F_3) \;\Leftrightarrow\; F_1 \;\rightarrow\; F_2 \land F_3$

(*y*) $F_1 \;\rightarrow\; F_2 \;\rightarrow\; F_3 \;\Leftrightarrow\; F_1 \land F_2 \;\rightarrow\; F_3$

(*z*) $(F_1 \leftrightarrow F_2) \;\land\; (F_2 \leftrightarrow F_3) \;\Rightarrow\; (F_1 \leftrightarrow F_3)$

1.3 (Redundant logical connectives). Given $\top$, $\land$, and $\neg$, prove that $\bot$, $\lor$, $\rightarrow$, and $\leftrightarrow$ are redundant logical connectives. That is, show that each of $\bot$, $F_1 \lor F_2$, $F_1 \rightarrow F_2$, and $F_1 \leftrightarrow F_2$ is equivalent to a formula that uses only F_1, F_2, $\top$, $\lor$, and $\neg$.

1.4 (The nand connective). Let the logical connective $\overline{\land}$ (pronounced "nand") be defined according to the following truth table:

F_1	F_2	$F_1 \overline{\land} F_2$
0	0	1
0	1	1
1	0	1
1	1	0

Show that all standard logical connectives can be defined in terms of $\overline{\land}$.

1.5 (Normal forms). Convert the following PL formulae to NNF, DNF, and CNF via the transformations of Section 1.6.

(*a*) $\neg(P \;\rightarrow\; Q)$

(*b*) $\neg(\neg(P \land Q) \;\rightarrow\; \neg R)$

(*c*) $(Q \land R \;\rightarrow\; (P \lor \neg Q)) \;\land\; (P \lor R)$

(*d*) $\neg(Q \rightarrow R) \;\land\; P \;\land\; (Q \;\lor\; \neg(P \land R))$

1.6 (Graph coloring). A solution to a **graph coloring** problem is an assignment of colors to vertices such that no two adjacent vertices have the same color. Formally, a finite graph $G = \langle V, E \rangle$ consists of vertices $V = \{v_1, \ldots, v_n\}$ and edges $E = \{\langle v_{i_1}, w_{i_1} \rangle, \ldots, \langle v_{i_k}, w_{i_k} \rangle\}$. The finite set of colors is given by $C = \{c_1, \ldots, c_m\}$. A problem instance is given by a graph and a set of colors: the problem is to assign each vertex $v \in V$ a $\mathsf{color}(v) \in C$ such that for every edge $\langle v, w \rangle \in E$, $\mathsf{color}(v) \neq \mathsf{color}(w)$. Clearly, not all instances have solutions.

Show how to encode an instance of a graph coloring problem into a PL formula F. F should be satisfiable iff a graph coloring exists.

(*a*) Describe a set of constraints in PL asserting that every vertex is colored. Since the sets of vertices, edges, and colors are all finite, use notation such as "$\mathsf{color}(v) = c$" to indicate that vertex v has color c. Realize that such an assertion is encodeable as a single propositional variable P_v^c.

(*b*) Describe a set of constraints in PL asserting that every vertex has at most one color.

(*c*) Describe a set of constraints in PL asserting that no two connected vertices have the same color.

(*d*) Identify a significant optimization in this encoding. *Hint*: Can any constraints be dropped? Why?

(*e*) If the constraints are not already in CNF, specify them in CNF now. For N vertices, K edges, and M colors, how many variables does the optimized encoding require? How many clauses?

1.7 (CNF). Example 1.25 constructs a CNF formula that is equisatisfiable to a given small formula in DNF.

(*a*) If distribution of disjunction over conjunction (described in Section 1.6) were used, how many clauses would the resulting formula have?

(*b*) Consider the formulae

$$F_n : \bigvee_{i=1}^{n} (Q_i \wedge R_i)$$

for positive integers n. As a function of n, how many clauses are in

(*i*) the formula F' constructed based on distribution of disjunction over conjunction?

(*ii*) the formula

$$F' : \mathsf{Rep}(F_n) \wedge \bigwedge_{G \in S_{F_n}} \mathsf{En}(G) \ ?$$

(*iii*) For which n is the distribution approach better?

1.8 (DPLL). Describe the execution of DPLL on the following formulae.

(*a*) $(P \vee \neg Q \vee \neg R) \wedge (Q \vee \neg P \vee R) \wedge (R \vee \neg Q)$

(*b*) $(P \vee Q \vee R) \wedge (\neg P \vee \neg Q \vee \neg R) \wedge (\neg P \vee Q \vee R) \wedge (\neg Q \vee R) \wedge (Q \vee \neg R)$

2

First-Order Logic

One task we logicians are interested in is that of analyzing the notion of "proof" — to make it as rigorous as any other notion in mathematics.

— Raymond Smullyan
The Lady or the Tiger?, 1982

This chapter extends the machinery of propositional logic to **first-order logic** (**FOL**), also called both **predicate logic** and the first-order **predicate calculus**. While first-order logic enjoys a degree of expressiveness that makes it suitable for reasoning about computation, it does not admit completely automated reasoning.

FOL extends PL with predicates, functions, and quantifiers. As in our discussion of PL, we first introduce the syntax of FOL and its semantics. We then build on the semantic argument method of PL to provide a method of proving first-order validity.

Section 2.6 reviews *decidability* and *complexity*. A decidable problem has an algorithm, which is a procedure that always finishes with a correct answer on every instance of the problem. While validity of PL formulae is decidable, validity of FOL formulae is not. Complexity is the study of the intrinsic hardness of a decidable problem.

The optional Section 2.7 proves the soundness and completeness of the semantic argument method. It also presents two classic theorems that are applied in Chapter 10.

2.1 Syntax

All formulae of PL evaluate to true or false. FOL is not so simple. In FOL, **terms** evaluate to values other than truth values such as integers, people, or cards of a deck. However, we are getting ahead of ourselves: just as in PL, the syntax of FOL is independent of its meaning. The most basic terms are **variables** x, y, z, x_1, x_2, ... and **constants** a, b, c, a_1, a_2, ...

More complicated terms are constructed using **functions**. An n-ary function f takes n terms as arguments. Notationally, we represent generic FOL functions by symbols f, g, h, f_1, f_2, ... A constant can also be viewed as a 0-ary function.

Example 2.1. The following are all terms:

- a, a constant (or 0-ary function);
- x, a variable;
- $f(a)$, a unary function f applied to a constant;
- $g(x, b)$, a binary function g applied to a variable x and a constant b;
- $f(g(x, f(b)))$.

■

The propositional variables of PL are generalized to **predicates**, denoted p, q, r, p_1, p_2, ... An n-ary predicate takes n terms as arguments. A **FOL propositional variable** is a 0-ary predicate, which we write P, Q, R, P_1, P_2, ...

Countably infinitely many constant, function, and predicate symbols are available.

An **atom** is $\top$, $\bot$, or an n-ary predicate applied to n terms. A **literal** is an atom or its negation.

Example 2.2. The following are all literals:

- P, a propositional variable (or 0-ary predicate);
- $p(f(x), g(x, f(x)))$, a binary predicate applied to two terms;
- $\neg p(f(x), g(x, f(x)))$.

The first two literals are also atoms. ■

A **FOL formula** is a literal, the application of a logical connective $\neg$, $\wedge$, $\vee$, $\rightarrow$, or $\leftrightarrow$ to a formula or formulae, or the application of a **quantifier** to a formula. There are two FOL quantifiers:

- the **existential quantifier** $\exists x.\ F[x]$, read "there exists an x such that $F[x]$";
- and the **universal quantifier** $\forall x.\ F[x]$, read "for all x, $F[x]$".

In $\forall x.\ F[x]$, x is the **quantified variable**, and $F[x]$ is the **scope** of the quantifier $\forall x$. For convenience, we sometimes refer informally to the scope of the quantifier $\forall x$ as the scope of the quantified variable x itself. The case is similar for $\exists x.\ F[x]$. Also, x in $F[x]$ is **bound** (by the quantifier). By convention, the period in ". $F[x]$" indicates that the scope of the quantified variable x extends as far as possible. We often abbreviate $\forall x.\ \forall y.\ F[x, y]$ by $\forall x, y.\ F[x, y]$.

Example 2.3. In

$$\forall x.\ \underbrace{p(f(x), x)\ \rightarrow\ (\exists y.\ \underbrace{p(f(g(x,y)), g(x,y)))}_{G}\ \land\ q(x, f(x))}_{F}\ ,$$

the scope of x is F, and the scope of y is G. This formula is read: "for all x, if $p(f(x), x)$ then there exists a y such that $p(f(g(x,y)), g(x,y))$ and $q(x, f(x))$". ■

A variable is **free** in formula $F[x]$ if there is an occurrence of x that is not bound by any quantifier. Denote by $\mathsf{free}(F)$ the set of free variables of a formula F. A variable is **bound** in formula $F[x]$ if there is an occurrence of x in the scope of a binding quantifier $\forall x$ or $\exists x$. Denote by $\mathsf{bound}(F)$ the set of bound variables of a formula F. In general, it is possible that $\mathsf{free}(F) \cap \mathsf{bound}(F) \neq \emptyset$, as a variable x can have both free and bound occurrences.

A formula F is **closed** if it does not contain any free variables.

Example 2.4. In

$$F:\ \forall x.\ p(f(x), y)\ \rightarrow\ \forall y.\ p(f(x), y)\ ,$$

x only occurs bound, while y appears both free (in the antecedent) and bound (in the consequent). Thus, $\mathsf{free}(F) = \{y\}$ and $\mathsf{bound}(F) = \{x, y\}$. ■

If $\mathsf{free}(F) = \{x_1, \ldots, x_n\}$, then its **universal closure** is

$$\forall x_1.\ \ldots \forall x_n.\ F\ ,$$

and its **existential closure** is

$$\exists x_1.\ \ldots \exists x_n.\ F\ .$$

We usually write the universal and existential closures as $\forall *.\ F$ and $\exists *.\ F$, respectively.

The **subformulae** of a FOL formula are defined according to an extension of the PL definition of subformula:

- the only subformula of $p(t_1, \ldots, t_n)$, where the t_i are terms, is $p(t_1, \ldots, t_n)$;
- the subformulae of $\neg F$ are $\neg F$ and the subformulae of F;
- the subformulae of $F_1 \land F_2$, $F_1 \lor F_2$, $F_1 \rightarrow F_2$, $F_1 \leftrightarrow F_2$ are the formula itself and the subformulae of F_1 and F_2;
- and the subformulae of $\exists x.\ F$ and $\forall x.\ F$ are the formula itself and the subformulae of F.

The **strict subformulae** of a formula excludes the formula itself.

The **subterms** of a FOL term are defined as follows:

- the only subterm of constant a or variable x is a or x itself, respectively;

- and the subterms of $f(t_1, \ldots, t_n)$ are the term itself and the subterms of $t_1, \ldots, t_n$.

The **strict subterms** of a term excludes the term itself.

Example 2.5. In

$$F : \forall x.\ p(f(x), y) \ \rightarrow \ \forall y.\ p(f(x), y) \ ,$$

the subformulae of F are

$$F \ , \ p(f(x), y) \ \rightarrow \ \forall y.\ p(f(x), y) \ , \ \forall y.\ p(f(x), y) \ , \ p(f(x), y) \ .$$

The subterms of $g(f(x), f(h(f(x))))$ are

$$g(f(x), f(h(f(x)))) \ , \ f(x) \ , \ f(h(f(x))) \ , \ h(f(x)) \ , \ x \ .$$

$f(x)$ occurs twice in $g(f(x), f(h(f(x))))$. ■

Example 2.6. Before discussing the formal semantics for FOL, we suggest translations of English sentences into FOL. The names of the constants, functions, and predicates are chosen to provide some intuition for the meaning of the FOL formulae.

- Every dog has its day.

$$\forall x.\ \mathit{dog}(x) \ \rightarrow \ \exists y.\ \mathit{day}(y) \ \land \ \mathit{itsDay}(x, y)$$

- Some dogs have more days than others.

$$\exists x, y.\ \mathit{dog}(x) \ \land \ \mathit{dog}(y) \ \land \ \#\mathit{days}(x) > \#\mathit{days}(y)$$

- All cats have more days than dogs.

$$\forall x, y.\ \mathit{dog}(x) \ \land \ \mathit{cat}(y) \ \rightarrow \ \#\mathit{days}(y) > \#\mathit{days}(x)$$

- Fido is a dog. Furrball is a cat. Fido has fewer days than does Furrball.

$$\mathit{dog}(\mathit{Fido}) \ \land \ \mathit{cat}(\mathit{Furrball}) \ \land \ \#\mathit{days}(\mathit{Fido}) < \#\mathit{days}(\mathit{Furrball})$$

- The length of one side of a triangle is less than the sum of the lengths of the other two sides.

$$\forall x, y, z.\ \mathit{triangle}(x, y, z) \ \rightarrow \ \mathit{length}(x) < \mathit{length}(y) + \mathit{length}(z)$$

- Fermat's Last Theorem.

$$\begin{array}{l} \forall n.\ \mathit{integer}(n) \ \land \ n > 2 \\ \quad \rightarrow \ \forall x, y, z. \\ \qquad \mathit{integer}(x) \ \land \ \mathit{integer}(y) \ \land \ \mathit{integer}(z) \ \land \ x > 0 \ \land \ y > 0 \ \land \ z > 0 \\ \qquad\quad \rightarrow \ x^n + y^n \neq z^n \end{array}$$

■

2.2 Semantics

Having defined the syntax of FOL, we now define its **semantics**. Formulae of FOL evaluate to the truth values true and false as in PL. However, terms of FOL formulae evaluate to values from a specified domain. We extend the concept of interpretations to this more complex setting and then define the semantics of FOL in terms of interpretations.

First, we define a FOL **interpretation** I. The **domain** D_I of an interpretation I is a nonempty set of values or objects, such as integers, real numbers, dogs, people, or merely abstract objects. $|D_I|$ denotes the **cardinality**, or size, of D_I. Domains can be finite, such as the 52 cards of a deck of cards; countably infinite, such as the integers; or uncountably infinite, such as the reals. But all domains are nonempty.

The **assignment** α_I of interpretation I maps constant, function, and predicate symbols to elements, functions, and predicates over D_I. It also maps variables to elements of D_I:

- each variable symbol x is assigned a value x_I from D_I;
- each n-ary function symbol f is assigned an n-ary function

  $$f_I : D_I^n \to D_I$$

  that maps n elements of D_I to an element of D_I;
- each n-ary predicate symbol p is assigned an n-ary predicate

  $$p_I : D_I^n \to \{\mathsf{true},\ \mathsf{false}\}$$

  that maps n elements of D_I to a truth value.

In particular, each constant (0-ary function symbol) is assigned a value from D_I, and each propositional variable (0-ary predicate symbol) is assigned a truth value.

An interpretation $I : (D_I, \alpha_I)$ is thus a pair consisting of a domain and an assignment.

Example 2.7. The formula

$$F :\ x + y > z \ \to\ y > z - x$$

contains the binary function symbols $+$ and $-$, the binary predicate symbol $>$, and the variables x, y, and z. Again, $+$, $-$, and $>$ are just symbols: we choose these names to provide intuition for the intended meaning of the formulae. We could just as easily have written

$$F' :\ p(f(x,y), z) \ \to\ p(y, g(z,x)) \ .$$

We construct a “standard” interpretation. The domain is the integers, $\mathbb{Z}$:

$$D_I = \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \ .$$

To $+$ and $-$ we assign standard addition $+_{\mathbb{Z}}$ and subtraction $-_{\mathbb{Z}}$ of integers, respectively. To $>$ we assign the standard greater-than relation $>_{\mathbb{Z}}$ of integers. Finally, to x, y, and z, we assign the values 13, 42, and 1, respectively. We ignore the countably infinitely many other constant, function, and predicate symbols that do not appear in F. We thus have interpretation $I : (\mathbb{Z}, \alpha_I)$, where

$$\alpha_I : \{+ \mapsto +_{\mathbb{Z}},\ - \mapsto -_{\mathbb{Z}},\ > \mapsto >_{\mathbb{Z}},\ x \mapsto 13,\ y \mapsto 42,\ z \mapsto 1,\ \ldots\} .$$

The elision reminds us that, as always, α_I provides values for the countably infinitely many other constant, function, and predicate symbols. Usually, we do not write the elision. ■

Given a FOL formula F and interpretation $I : (D_I, \alpha_I)$, we want to compute if F evaluates to true under interpretation I, $I \models F$, or if F evaluates to false under interpretation I, $I \not\models F$. We define the semantics inductively as in PL. To start, define the meaning of truth symbols:

$$\begin{array}{lll} I & \models & \top \\ I & \not\models & \bot \end{array}$$

Next, consider more complicated atoms. α_I gives meaning $\alpha_I[x]$, $\alpha_I[c]$, and $\alpha_I[f]$ to variables x, constants c, and functions f. Evaluate arbitrary terms recursively:

$$\alpha_I[f(t_1, \ldots, t_n)] \ = \ \alpha_I[f](\alpha_I[t_1], \ldots, \alpha_I[t_n]) \ ,$$

for terms $t_1, \ldots, t_n$. That is, define the value of $f(t_1, \ldots, t_n)$ under α_I by evaluating the function $\alpha_I[f]$ over the terms $\alpha_I[t_1]$, $\ldots$, $\alpha_I[t_n]$. Similarly, evaluate arbitrary atoms recursively:

$$\alpha_I[p(t_1, \ldots, t_n)] \ = \ \alpha_I[p](\alpha_I[t_1], \ldots, \alpha_I[t_n]) \ .$$

Then

$$I \ \models \ p(t_1, \ldots, t_n) \qquad \text{iff } \alpha_I[p(t_1, \ldots, t_n)] = \mathsf{true}$$

Having completed the base cases of our inductive definition, we turn to the inductive step. Assume that formulae F_1 and F_2 have fixed truth values. From these formulae, evaluate the semantics of more complex formulae. The logical connectives are handled in FOL in precisely the same way as in PL:

$$\begin{array}{llll} I & \models & \neg F & \text{iff } I \not\models F \\ I & \models & F_1 \wedge F_2 & \text{iff } I \models F_1 \text{ and } I \models F_2 \\ I & \models & F_1 \vee F_2 & \text{iff } I \models F_1 \text{ or } I \models F_2 \\ I & \models & F_1 \rightarrow F_2 & \text{iff, if } I \models F_1 \text{ then } I \models F_2 \\ I & \models & F_1 \leftrightarrow F_2 & \text{iff } I \models F_1 \text{ and } I \models F_2\text{, or } I \not\models F_1 \text{ and } I \not\models F_2 \end{array}$$

Example 2.8. Recall the formula

$$F:\ x+y>z\ \rightarrow\ y>z-x$$

of Example 2.7 and the interpretation $I : (\mathbb{Z}, \alpha_I)$, where

$$\alpha_I:\ \{+\mapsto +_{\mathbb{Z}},\ -\mapsto -_{\mathbb{Z}},\ >\mapsto >_{\mathbb{Z}},\ x\mapsto 13_{\mathbb{Z}},\ y\mapsto 42_{\mathbb{Z}},\ z\mapsto 1_{\mathbb{Z}}\}\ .$$

Compute the truth value of F under I as follows:

$$\begin{array}{llll}
1. & I \models x+y>z & & \text{since } \alpha_I[x+y>z] = 13_{\mathbb{Z}} +_{\mathbb{Z}} 42 >_{\mathbb{Z}} 1_{\mathbb{Z}} \\
2. & I \models y>z-x & & \text{since } \alpha_I[y>z-x] = 42_{\mathbb{Z}} >_{\mathbb{Z}} 1_{\mathbb{Z}} -_{\mathbb{Z}} 13_{\mathbb{Z}} \\
3. & I \models F & & \text{by 1, 2, and the semantics of } \rightarrow
\end{array}$$

■

For the quantifiers, let x be a variable. Define an **x-variant** of an interpretation $I : (D_I, \alpha_I)$ as an interpretation $J : (D_J, \alpha_J)$ such that

- $D_I = D_J$;
- and $\alpha_I[y] = \alpha_J[y]$ for all constant, free variable, function, and predicate symbols y, except possibly x.

That is, I and J agree on everything except possibly the value of variable x. Denote by $J :\ I \triangleleft \{x \mapsto \mathsf{v}\}$ the x-variant of I in which $\alpha_J[x] = \mathsf{v}$ for some $\mathsf{v} \in D_I$. Then

$$\begin{array}{lll}
I \models \forall x.\ F & & \text{iff for all } \mathsf{v} \in D_I,\ I \triangleleft \{x \mapsto \mathsf{v}\} \models F \\
I \models \exists x.\ F & & \text{iff there exists } \mathsf{v} \in D_I \text{ such that } I \triangleleft \{x \mapsto \mathsf{v}\} \models F
\end{array}$$

In words, I is an interpretation of $\forall x.\ F$ iff all x-variants of I are interpretations of F. I is an interpretation of $\exists x.\ F$ iff some x-variant of I is an interpretation of F.

Example 2.9. Consider the formula

$$F:\ \exists x.\ f(x) = g(x)$$

and the interpretation $I : (D : \{\circ, \bullet\},\ \alpha_I)$ in which

$$\alpha_I:\ \{f(\circ)\mapsto \circ,\ f(\bullet)\mapsto \bullet,\ g(\circ)\mapsto \bullet,\ g(\bullet)\mapsto \circ\}\ .$$

Compute the truth value of F under I as follows:

$$\begin{array}{lrlll}
1. & I \triangleleft \{x \mapsto \mathsf{v}\} & \not\models & f(x) = g(x) & \text{for } \mathsf{v} \in D \\
2. & I & \not\models & \exists x.\ f(x) = g(x) & \text{since } \mathsf{v} \in D \text{ is arbitrary}
\end{array}$$

In the first line, basic reasoning about the interpretation I reveals that f and g always disagree. The second line follows from the first by the semantics of existential quantification. ■

2.3 Satisfiability and Validity

A formula F is said to be **satisfiable** iff there exists an interpretation I such that $I \models F$. A formula F is said to be **valid** iff for all interpretations I, $I \models F$. Determining satisfiability and validity of formulae are important tasks in FOL. Recall that satisfiability and validity are dual: F is valid iff $\neg F$ is unsatisfiable.

Technically, satisfiability and validity only apply to closed FOL formulae, which do not have free variables. However, let us agree on a convention: if we say that a formula F such that $\mathsf{free}(F) \neq \emptyset$ is valid, we mean that its universal closure $\forall * .\ F$ is valid; and if we say that it is satisfiable, we mean that its existential closure $\exists * .\ F$ is satisfiable. Duality still holds: a formula F with free variables is valid ($\forall * .\ F$ is valid) iff its negation is unsatisfiable ($\exists * .\ \neg F$ is unsatisfiable). Henceforth, we freely discuss the validity and satisfiability of formulae with free variables.

For arguing the validity of FOL formulae, we extend the semantic argument method from PL to FOL. Most of the concepts carry over to the FOL case without change. In addition to the rules for the logical connectives of PL (see Section 1.3), we have the following rules for the quantifiers.

- According to the semantics of universal quantification, from $I \models \forall x.\ F$, deduce $I \triangleleft \{x \mapsto \mathsf{v}\} \models F$ for any $\mathsf{v} \in D_I$.

  $$\frac{I \ \models\ \forall x.\ F}{I \triangleleft \{x \mapsto \mathsf{v}\} \ \models\ F} \qquad \text{for any } \mathsf{v} \in D_I$$

  In practice, we usually apply this rule using a domain element v that was introduced earlier in the proof.
- Similarly, from the semantics of existential quantification, from $I \not\models \exists x.\ F$, deduce $I \triangleleft \{x \mapsto \mathsf{v}\} \not\models F$ for any $\mathsf{v} \in D_I$.

  $$\frac{I \ \not\models\ \exists x.\ F}{I \triangleleft \{x \mapsto \mathsf{v}\} \ \not\models\ F} \qquad \text{for any } \mathsf{v} \in D_I$$

  Again, we usually apply this rule using a domain element v that was introduced earlier in the proof.
- According to the semantics of existential quantification, from $I \models \exists x.\ F$, deduce $I \triangleleft \{x \mapsto \mathsf{v}\} \models F$ for some $\mathsf{v} \in D_I$ that has *not* been previously used in the proof.

  $$\frac{I \ \models\ \exists x.\ F}{I \triangleleft \{x \mapsto \mathsf{v}\} \ \models\ F} \qquad \text{for a } \textit{fresh } \mathsf{v} \in D_I$$

- Similarly, from the semantics of universal quantification, from $I \not\models \forall x.\ F$, deduce $I \triangleleft \{x \mapsto \mathsf{v}\} \not\models F$ for some $\mathsf{v} \in D_I$ that has *not* been previously used in the proof.

  $$\frac{I \ \not\models\ \forall x.\ F}{I \triangleleft \{x \mapsto \mathsf{v}\} \ \not\models\ F} \qquad \text{for a } \textit{fresh } \mathsf{v} \in D_I$$

The restriction in the latter two rules corresponds to our intuition: if all we know is that $\exists x.\ F$, then we certainly do not know which value in particular satisfies F. Hence, we choose a new value v that does not appear previously in the proof: it was never introduced before by a quantification rule. Moreover, α_I does not already assign it to some constant, $\alpha_I[a]$, or to some function application, $\alpha_I[f(t_1, \ldots, t_n)]$.

Notice the similarity between the first two and between the final two rules. The first two rules handle a case that is universal in character. Consider the second rule: if there does not exist an x such that F, then for all values, F does not hold. The final two rules are existential in character.

Lastly, the contradiction rule is modified for the FOL case.

- A contradiction exists if two variants of the original interpretation I disagree on the truth value of an n-ary predicate p for a given tuple of domain values.

$$\begin{array}{lll}
J : I \lhd \cdots & \models & p(s_1, \ldots, s_n) \\
K : I \lhd \cdots & \not\models & p(t_1, \ldots, t_n) \\
\hline
I & \models & \bot
\end{array}
\quad \text{for } i \in \{1, \ldots, n\},\ \alpha_J[s_i] = \alpha_K[t_i]$$

The intuition behind the contradiction rule is the following. The variants J and K are constructed only through the rules for quantification. Hence, the truth value of p on the given tuple of domain values is already established by I. Therefore, the disagreement between J and K on the truth value of p indicates a problem with I.

None of these rules cause branching, but several of the rules for the logical connectives do. Thus, a proof in general is a tree. A branch is closed if it contains a contradiction according to the (first-order) contradiction rule; it is open otherwise. All branches are closed in a finished proof of a valid formula. We exhibit the proof method through several examples.

Example 2.10. We prove that

$$F :\ (\forall x.\ p(x)) \ \rightarrow\ (\forall y.\ p(y))$$

is valid. Suppose not; then there is an interpretation I such that $I \not\models F$:

1.	I	$\not\models$	F	assumption
2.	I	$\models$	$\forall x.\ p(x)$	1 and semantics of →
3.	I	$\not\models$	$\forall y.\ p(y)$	1 and semantics of →
4.	$I \lhd \{y \mapsto \mathsf{v}\}$	$\not\models$	$p(y)$	3 and semantics of $\forall$, for some $\mathsf{v} \in D_I$
5.	$I \lhd \{x \mapsto \mathsf{v}\}$	$\models$	$p(x)$	2 and semantics of $\forall$

Lines 2 and 3 state the case in which line 1 holds: the antecedent and consequent of F are respectively true and false under I. Line 4 states that because of 3, there must be a value $\mathsf{v} \in D_I$ such that $I \lhd \{y \mapsto \mathsf{v}\} \not\models p(y)$. Line 5 uses this same value v and the semantics of $\forall$ with 2 to derive a contradiction: under I, $p(\mathsf{v})$ is false by 4 and true by 5. Thus, F is valid. ■

For concision we shorten, for example, "semantics of ∀" to "∀" in the explanation column of our arguments.

Example 2.11. Consider the following relation between universal and existential quantification:

$$F : \ (\forall x.\ p(x)) \ \leftrightarrow \ (\neg\exists x.\ \neg p(x)) \ .$$

Is it valid? Suppose not. Then there is an interpretation I such that $I \not\models F$. Consider the forward ($\rightarrow$) and backward ($\leftarrow$) directions of $\leftrightarrow$ as separate cases. In the first case,

1.	I	$\models$	$\forall x.\ p(x)$	assumption
2.	I	$\not\models$	$\neg\exists x.\ \neg p(x)$	assumption
3.	I	$\models$	$\exists x.\ \neg p(x)$	2 and $\neg$
4.	$I \triangleleft \{x \mapsto \mathsf{v}\}$	$\models$	$\neg p(x)$	3 and $\exists$, for some $\mathsf{v} \in D_I$
5.	$I \triangleleft \{x \mapsto \mathsf{v}\}$	$\models$	$p(x)$	1 and $\forall$

Lines 4 and 5 are contradictory. In line 5, we use the value introduced in line 4 with the semantics of ∀ and line 1. We are allowed to choose this same value v precisely because line 1 states that *for all* x, $p(x)$.

For the second case,

1.	I	$\not\models$	$\forall x.\ p(x)$	assumption
2.	I	$\models$	$\neg\exists x.\ \neg p(x)$	assumption
3.	$I \triangleleft \{x \mapsto \mathsf{v}\}$	$\not\models$	$p(x)$	1 and $\forall$, for some $\mathsf{v} \in D_I$
4.	I	$\not\models$	$\exists x.\ \neg p(x)$	2 and $\neg$
5.	$I \triangleleft \{x \mapsto \mathsf{v}\}$	$\not\models$	$\neg p(x)$	4 and $\exists$
6.	$I \triangleleft \{x \mapsto \mathsf{v}\}$	$\models$	$p(x)$	5 and $\neg$

Lines 3 and 6 are contradictory. Line 4 says that $\exists x.\ \neg p(x)$ is false under I. Thus, by the semantics of ∃, no value w from D_I is such that $p(\mathsf{w})$ is true. In particular, line 5 identifies v, introduced in line 3.

As both cases end in contradictions for arbitrary interpretation I, F is valid. ■

It is sometimes useful to reference known values, as the following simple example illustrates.

Example 2.12. To prove that

$$F : \ p(a) \ \rightarrow \ \exists x.\ p(x)$$

is valid, assume otherwise and derive a contradiction.

1.	I	$\not\models$	F	assumption
2.	I	$\models$	$p(a)$	1 and $\rightarrow$
3.	I	$\not\models$	$\exists x.\ p(x)$	1 and $\rightarrow$
4.	$I \triangleleft \{x \mapsto \alpha_I[a]\}$	$\not\models$	$p(x)$	3 and $\exists$
5.	I	$\models$	$\bot$	2, 4

In line 4, we used the value assigned to a to instantiate the quantified variable of line 3, which has universal character. Because lines 2 and 4 are contradictory, F is valid. ■

To show that a formula F is invalid, it suffices to find an interpretation I such that $I \models \neg F$.

Example 2.13. Consider the formula

$$F: (\forall x.\ p(x,x)) \rightarrow (\exists x.\ \forall y.\ p(x,y)) .$$

To show that it is invalid, we find an interpretation I such that

$$I \models \neg((\forall x.\ p(x,x)) \rightarrow (\exists x.\ \forall y.\ p(x,y))) ,$$

or, according to the semantics of →,

$$I \models (\forall x.\ p(x,x)) \land \neg(\exists x.\ \forall y.\ p(x,y)) .$$

Choose

$$D_I = \{0,1\}$$

and

$$p_I = \{(0,0),\ (1,1)\} .$$

We use a common notation for defining relations: $p_I(a,b)$ is true iff $(a,b) \in p_I$. Here, $p_I(0,0)$ is true, and $p_I(1,0)$ is false.

Both $\forall x.\ p(x,x)$ and $\neg(\exists x.\ \forall y.\ p(x,y))$ evaluate to true under I, so

$$I \models (\forall x.\ p(x,x)) \land \neg(\exists x.\ \forall y.\ p(x,y)) ,$$

which shows that F is invalid. Interpretation I is a falsifying interpretation of F. ■

We apply the semantic argument method to more examples in Section 3.1.

Equivalence ($F_1 \Leftrightarrow F_2$) and implication ($F_1 \Rightarrow F_2$) extend directly from PL to FOL. Equivalence of and implication between two formulae can be argued using the semantic argument method. See, for example, Example 2.11.

2.4 Substitution

Substitution for FOL is more complex than substitution for PL because of quantification. We introduce two types of substitution in this section with the goal of generalizing Propositions 1.15 and 1.17 to the FOL setting. As in PL, substitution allows us to consider the validity of entire sets of formulae simultaneously.

First, we define the **renaming** of a quantified variable. If variable x is quantified in F so that F has the form $F[\forall x.\ G[x]]$, then the renaming of x to fresh variable x' produces the formula $F[\forall x'.\ G[x']]$. A "fresh variable" is any variable that does not occur in F. By the semantics of universal quantification, the original and final formulae are equivalent. The case is similar for existential quantification. Often, we simply say that a variable is renamed to mean that its bound occurrences are renamed to a fresh variable. Free occurrences of variables are never renamed.

Example 2.14. Renaming the bound variable x to fresh variable x' in

$$F:\ p(x)\ \wedge\ \forall x.\ q(x,y)$$

produces

$$F':\ p(x)\ \wedge\ \forall x'.\ q(x',y)\ .$$

Renaming y does not cause any change because y does not occur bound. ■

A **substitution** is a map from FOL formulae to FOL formulae:

$$\sigma:\ \{F_1 \mapsto G_1,\ \ldots,\ F_n \mapsto G_n\}\ .$$

As in PL, $\mathsf{domain}(\sigma) = \{F_1, \ldots, F_n\}$ and $\mathsf{range}(\sigma) = \{G_1, \ldots, G_n\}$. To compute the application of σ to F, $F\sigma$, replace each occurrence of F_i in F by G_i simultaneously. When both subformulae F_j and F_k are in the domain of σ and F_k is a strict subformula of F_j, replace occurrences of F_j by G_j.

Example 2.15. Consider formula

$$F:\ (\forall x.\ p(x,y))\ \rightarrow\ q(f(y),x)$$

and substitution

$$\sigma:\ \{x \mapsto g(x),\ y \mapsto f(x),\ q(f(y),x) \mapsto \exists x.\ h(x,y)\}\ .$$

Then

$$F\sigma:\ (\forall x.\ p(g(x),f(x)))\ \rightarrow\ \exists x.\ h(x,y)\ .$$

Notice how there are more bound occurrences of x in $F\sigma$ than in F. ■

Use care when substitutions include quantifiers in the domain. Substituting for a quantified subformula $\forall x.\ F$ requires that all of $\forall x.\ F$ be replaced.

Example 2.16. Consider formula

$$F:\ \exists y.\ p(x,y)\ \wedge\ p(y,x)$$

and substitution

$$\sigma:\ \{\exists y.\ p(x,y) \mapsto p(x,a)\}\ ,$$

where a is a constant. $F\sigma = F$ because the scope of the quantifier $\exists y$ in F is $p(x,y) \wedge p(y,x)$, not just $p(x,y)$. ■

2.4.1 Safe Substitution

A restricted application of substitution has a useful semantic property.

Define for a substitution σ its set of free variables:

$$V_\sigma = \bigcup_i (\mathsf{free}(F_i) \cup \mathsf{free}(G_i)) \ .$$

V_σ consists of the free variables of all formulae F_i and G_i of the domain and range of σ. Compute the **safe substitution** $F\sigma$ of formula F as follows:

1. For each quantified variable x in F such that $x \in V_\sigma$, rename x to a fresh variable to produce F'.
2. Compute $F'\sigma$.

Example 2.17. Consider again formula

$$F : \ (\forall x.\ p(x, y)) \ \rightarrow \ q(f(y), x)$$

and substitution

$$\sigma : \ \{x \mapsto g(x), \ y \mapsto f(x), \ q(f(y), x) \mapsto \exists x.\ h(x, y)\} \ .$$

To compute the safe substitution $F\sigma$, first compute

$$\begin{aligned} V_\sigma &= \mathsf{free}(x) \ \cup \ \mathsf{free}(g(x)) \ \cup \ \mathsf{free}(y) \ \cup \ \mathsf{free}(f(x)) \\ &\quad \cup \mathsf{free}(q(f(y), x)) \ \cup \ \mathsf{free}(\exists x.h(x, y)) \\ &= \{x\} \cup \{x\} \cup \{y\} \cup \{x\} \cup \{x, y\} \cup \{y\} \\ &= \{x, y\} \end{aligned}$$

Then

1. $x \in V_\sigma$, so rename bound occurrences in F:

   $$F' : \ (\forall x'.\ p(x', y)) \ \rightarrow \ q(f(y), x) \ .$$

   x also occurs free in F.
2. $F'\sigma : \ (\forall x'.\ p(x', f(x))) \ \rightarrow \ \exists x.\ h(x, y)$.

■

The first step of computing a safe substitution becomes trivial if each quantified variable has a unique name.

Example 2.18. Consider formula

$$F : \ (\forall z.\ p(z, y)) \ \rightarrow \ q(f(y), x) \ ,$$

in which the quantified variable has a different name than any free variable of F or the substitution

$$\sigma : \{x \mapsto g(x),\ y \mapsto f(y),\ q(f(y), x) \mapsto \exists w.\ h(w, y)\}\ .$$

Compared to Example 2.17, the quantified variable z in F and the quantified variable w of σ have different names than any other variable of F or σ. The safe substitution is the unrestricted substitution

$$F\sigma :\ (\forall z.\ p(z, f(y))) \ \rightarrow\ \exists w.\ h(w, y)\ .$$

■

Proposition 2.19 (Substitution of Equivalent Formulae). *Consider substitution*

$$\sigma :\ \{F_1 \mapsto G_1,\ \ldots,\ F_n \mapsto G_n\}$$

such that for each i, $F_i \Leftrightarrow G_i$. Then $F \Leftrightarrow F\sigma$ when $F\sigma$ is computed as a safe substitution.

The language of Propositions 2.19 and 1.15 are almost identical.

2.4.2 Schema Substitution

Example 2.11 proves an interesting relation between universal and existential quantification ($\forall x.\ p(x) \Leftrightarrow \neg\exists x.\ \neg p(x)$), but the result is not general. We would like to prove that for any FOL formula F,

$$H :\ (\forall x.\ F) \ \leftrightarrow\ (\neg\exists x.\ \neg F)$$

is valid. H is a **formula schema** (plural: **schemata**). Formula schema and **schema substitutions** provide the desired generality.

A **formula schema** H contains at least one **placeholder** $F_1, F_2, \ldots$. For example, F is a placeholder in the formula schema H above. A formula schema can also have **side conditions** that specify that certain variables do not occur free in the placeholders.

Consider a substitution σ mapping placeholders to FOL formulae. A **schema substitution** is an (unrestricted) application of σ to a formula schema. It is legal only if σ obeys the side conditions of the formula schema.

Example 2.20. Recall from Example 2.11 that

$$(\forall x.\ p(x)) \ \leftrightarrow\ (\neg\exists x.\ \neg p(x))$$

is valid. It can act as a formula schema. First, rewrite the formula using placeholders:

$$H :\ (\forall x.\ F) \ \leftrightarrow\ (\neg\exists x.\ \neg F)\ .$$

H does not have any side conditions. Next, to prove the validity of

$$G: \ (\forall x.\ \exists y.\ q(x,y)) \ \leftrightarrow \ (\neg\exists x.\ \neg\exists y.\ q(x,y)) \ ,$$

show that G is derivable from H via a schema substitution:

$$\sigma: \ \{F \mapsto \exists y.\ q(x,y)\} \ .$$

Then $H\sigma = G$ ($H\sigma$ is syntactically identical to G), so that by Proposition 2.25 below, G is valid. ■

A formula schema H is **valid** if $H\sigma$ is valid for every schema substitution σ that obeys the side conditions of H. Apply the semantic method to prove the validity of a formula schema.

Example 2.21. To prove the validity of the formula schema

$$H: \ (\forall x.\ F_1 \wedge F_2) \ \leftrightarrow \ (\forall x.\ F_1) \ \wedge \ (\forall x.\ F_2) \ ,$$

consider the two directions. First, assume that $I \not\models (\forall x.\ F_1 \wedge F_2) \ \rightarrow \ (\forall x.\ F_1) \wedge (\forall x.\ F_2)$:

$$\begin{array}{llll}
1. & I & \models & \forall x.\ F_1 \wedge F_2 & \text{assumption} \\
2. & I & \not\models & (\forall x.\ F_1) \ \wedge \ (\forall x.\ F_2) & \text{assumption} \\
3. & I & \models & (\exists x.\ \neg F_1) \ \vee \ (\exists x.\ \neg F_2) & 2,\ \neg
\end{array}$$

There are two cases to consider:

$$\begin{array}{lrlll}
4a. & I & \models & \exists x.\ \neg F_1 & 3,\ \vee \\
5a. & I \triangleleft \{x \mapsto \mathsf{v}\} & \models & \neg F_1 & 4a,\ \exists,\ \text{for some } \mathsf{v} \in D_I \\
6a. & I \triangleleft \{x \mapsto \mathsf{v}\} & \models & F_1 \wedge F_2 & 1,\ \forall \\
7a. & I \triangleleft \{x \mapsto \mathsf{v}\} & \models & F_1 & 6a,\ \wedge
\end{array}$$

ending in a contradiction. The second disjunctive case is similar.

For the second main case, assume that $I \not\models (\forall x.\ F_1) \ \wedge \ (\forall x.\ F_2) \ \rightarrow (\forall x.\ F_1 \wedge F_2)$:

$$\begin{array}{lrlll}
1. & I & \not\models & \forall x.\ F_1 \wedge F_2 & \text{assumption} \\
2. & I & \models & (\forall x.\ F_1) \ \wedge \ (\forall x.\ F_2) & \text{assumption} \\
3. & I & \models & \exists x.\ \neg F_1 \vee \neg F_2 & 1,\ \neg \\
4. & I \triangleleft \{x \mapsto \mathsf{v}\} & \models & \neg F_1 \vee \neg F_2 & 3,\ \exists,\ \text{for some } \mathsf{v} \in D_I \\
5. & I & \models & \forall x.\ F_1 & 2,\ \wedge \\
6. & I & \models & \forall x.\ F_2 & 2,\ \wedge \\
7. & I \triangleleft \{x \mapsto \mathsf{v}\} & \models & F_1 & 5,\ \forall \\
8. & I \triangleleft \{x \mapsto \mathsf{v}\} & \models & F_2 & 6,\ \forall
\end{array}$$

Again, there are two cases to consider:

$$\begin{array}{lrlll}
9a. & I \triangleleft \{x \mapsto \mathsf{v}\} & \models & \neg F_1 & 4,\ \vee
\end{array}$$

ending in a contradiction. The second disjunctive case is similar. Thus, H is a valid formula schema. ■

We now consider formula schemata with side conditions.

Example 2.22. Consider the formula schema with side condition

$$H : \ (\forall x.\ F) \ \leftrightarrow \ F \quad \text{provided } x \notin \mathsf{free}(F)\ .$$

If we disregard the side condition, then H is an invalid formula schema as, for example,

$$G_1 : \ (\forall x.\ p(x)) \ \leftrightarrow \ p(x)\ ,$$

obtained from H by schema substitution

$$\sigma : \ \{F \mapsto p(x)\}\ ,$$

is invalid. However, σ is disallowed by the side condition that x should not occur free in F.

H (with the side condition) is a valid formula schema. Thus, the formula

$$G_2 : \ (\forall x.\ \exists y.\ p(z, y)) \ \leftrightarrow \ \exists y.\ p(z, y)$$

is valid: obtain it via the schema substitution

$$\sigma : \ \{F \mapsto \exists y.\ p(z, y)\}\ ,$$

which obeys H's side condition. ■

Reasoning about the validity of a formula schema with side conditions usually requires invoking its side conditions during a semantic argument.

Example 2.23. To prove the validity of

$$H : \ (\forall x.\ F) \ \leftrightarrow \ F \quad \text{provided } x \notin \mathsf{free}(F)\ ,$$

consider the two directions of ↔. First,

1.	$I \models \forall x.\ F$	assumption
2.	$I \not\models F$	assumption
3.	$I \models F$	1, $\forall$, since $x \notin \mathsf{free}(F)$
4.	$I \models \bot$	2, 3

Second,

1.	$I \not\models \forall x.\ F$	assumption
2.	$I \models F$	assumption
3.	$I \models \exists x.\ \neg F$	1 and $\neg$
4.	$I \models \neg F$	3, $\exists$, since $x \notin \mathsf{free}(F)$
5.	$I \models \bot$	2, 4

Thus, H is a valid formula schema. ■

Example 2.24. To prove the validity of

$$H:\ (\forall x.\ F_1 \wedge F_2)\ \leftrightarrow\ (\forall x.\ F_1)\ \wedge\ F_2 \quad \text{provided } x \notin \mathsf{free}(F_2)\ ,$$

consider two cases. First,

$$\begin{array}{lll}
1. & I \models \forall x.\ F_1\ \wedge\ F_2 & \text{assumption} \\
2. & I \not\models (\forall x.\ F_1)\ \wedge\ F_2 & \text{assumption} \\
3. & I \models (\forall x.\ F_1)\ \wedge\ (\forall x.\ F_2) & \text{1, valid schema from Example 2.21} \\
4. & I \models (\forall x.\ F_1)\ \wedge\ F_2 & \text{3, Example 2.23, since } x \notin \mathsf{free}(F_2) \\
5. & I \models \bot & \text{2, 4}
\end{array}$$

Observe the application of valid formula schemata from previous examples in lines 3 and 4. Second,

$$\begin{array}{lll}
1. & I \not\models \forall x.\ F_1\ \wedge\ F_2 & \text{assumption} \\
2. & I \models (\forall x.\ F_1)\ \wedge\ F_2 & \text{assumption} \\
3. & I \models (\forall x.\ F_1)\ \wedge\ (\forall x.\ F_2) & \text{2, Example 2.23, since } x \notin \mathsf{free}(F_2) \\
4. & I \models \forall x.\ F_1\ \wedge\ F_2 & \text{3, Example 2.21} \\
5. & I \models \bot & \text{1, 4}
\end{array}$$

Thus, H is a valid formula schema. ■

Proposition 2.25 (Formula Schema). *If H is a valid formula schema and σ is a substitution obeying H's side conditions, then $H\sigma$ is also valid.*

The valid PL formula

$$(P \rightarrow Q)\ \leftrightarrow\ (\neg P \vee Q)$$

can be treated as a valid formula schema:

$$(F_1 \rightarrow F_2)\ \leftrightarrow\ (\neg F_1 \vee F_2)\ .$$

In general, valid propositional templates are valid formulae schemata, so that Proposition 2.25 generalizes Proposition 1.17.

2.5 Normal Forms

The normal forms of PL extend to FOL. A FOL formula F can be transformed into negation normal form (NNF) by using the procedure of Section 1.6 augmented with these two (schema) equivalences:

$$\neg\forall x.\ F[x]\ \Leftrightarrow\ \exists x.\ \neg F[x] \qquad \neg\exists x.\ F[x]\ \Leftrightarrow\ \forall x.\ \neg F[x]\ .$$

Example 2.26. We apply the procedure to find a formula in NNF that is equivalent to

$$G:\ \forall x.\ (\exists y.\ p(x,y)\ \wedge\ p(x,z))\ \rightarrow\ \exists w.p(x,w)\ .$$

Each formula below is equivalent to G and is obtained from the previous one through an application of an equivalence.

1. $\forall x.\ (\exists y.\ p(x,y)\ \land\ p(x,z))\ \rightarrow\ \exists w.\ p(x,w)$
2. $\forall x.\ \neg(\exists y.\ p(x,y)\ \land\ p(x,z))\ \lor\ \exists w.\ p(x,w)$
3. $\forall x.\ (\forall y.\ \neg(p(x,y)\ \land\ p(x,z)))\ \lor\ \exists w.\ p(x,w)$
4. $\forall x.\ (\forall y.\ \neg p(x,y)\ \lor\ \neg p(x,z))\ \lor\ \exists w.\ p(x,w)$

Formula 2 follows from the equivalence

$$F_1 \rightarrow F_2 \ \Leftrightarrow\ \neg F_1 \lor F_2\ .$$

Formula 3 arises from an application of

$$\neg\exists x.\ F[x]\ \Leftrightarrow\ \forall x.\ \neg F[x]\ ,$$

and the final formula, which is in NNF, follows from De Morgan's Law. ■

A formula is in **prenex normal form** (**PNF**) if all of its quantifiers appear at the beginning of the formula:

$$\mathsf{Q}_1 x_1.\ \ldots \mathsf{Q}_n x_n.\ F[x_1,\ldots,x_n]\ ,$$

where $\mathsf{Q}_i \in \{\forall,\exists\}$ and F is quantifier-free. Every FOL formula F can be transformed into an equivalent formula F' in PNF. To compute an equivalent PNF F' of F,

1. Convert F into NNF formula F_1.
2. When multiple quantified variables have the same name, rename them to fresh variables, resulting in F_2.
3. Remove all quantifiers from F_2 to produce quantifier-free formula F_3.
4. Add the quantifiers before F_3,

   $$F_4:\ \mathsf{Q}_1 x_1.\ \ldots \mathsf{Q}_n x_n.\ F_3\ ,$$

   where the Q_i are the quantifiers such that if Q_j is in the scope of Q_i in F_1, then $i < j$.

F_4 is equivalent to F.

Example 2.27. We apply the procedure to find a PNF equivalent of

$$F:\ \forall x.\ \neg(\exists y.\ p(x,y)\ \land\ p(x,z))\ \lor\ \exists y.\ p(x,y)\ .$$

1. Write F in NNF:

   $$F_1:\ \forall x.\ (\forall y.\ \neg p(x,y)\ \lor\ \neg p(x,z))\ \lor\ \exists y.\ p(x,y)\ .$$

2. Rename quantified variables:

   $$F_2:\ \forall x.\ (\forall y.\ \neg p(x,y)\ \lor\ \neg p(x,z))\ \lor\ \exists w.\ p(x,w)\ .$$

3. Remove all quantifiers to produce quantifier-free formula

$$F_3 : \neg p(x,y) \ \lor \ \neg p(x,z) \ \lor \ p(x,w) \ .$$

4. Add the quantifiers before F_3:

$$F_4 : \forall x.\ \forall y.\ \exists w.\ \neg p(x,y) \ \lor \ \neg p(x,z) \ \lor \ p(x,w) \ .$$

Alternately, choose order

$$F_4' : \forall x.\ \exists w.\ \forall y.\ \neg p(x,y) \ \lor \ \neg p(x,z) \ \lor \ p(x,w) \ .$$

Both F_4 and F_4' are equivalent to F. However,

$$G : \forall y.\ \exists w.\ \forall x.\ \neg p(x,y) \ \lor \ \neg p(x,z) \ \lor \ p(x,w)$$

is not equivalent to F.

■

A FOL formula is in CNF (DNF) if it is in PNF and its main quantifier-free subformula is in CNF (DNF). CNF and DNF equivalents are obtained by transforming formula F into PNF formula F' and then applying the relevant procedure of Section 1.6 to the main quantifier-free subformula of F'.

2.6 Decidability and Complexity

We review the main concepts from decidability and complexity theory. **Bibliographic Remarks** refers the interested reader to other texts that focus on these topics.

2.6.1 Satisfiability as a Formal Language

Satisfiability of formulae is the primary decision problem in logic. We can formalize satisfiability as a formal language decision problem. Consider PL. Let L_{PL} be the set of all satisfiable formulae. That is, the word $w \in L_{\mathsf{PL}}$ iff

1. w is a syntactically well-formed formulae: it parses according to the definition of Section 1.1;
2. and when w is viewed as a PL formula F, F is satisfiable.

Then the formal decision problem is the following: given a word w, is $w \in L_{\mathsf{PL}}$?

Satisfiability of FOL formulae can be similarly formalized as a language question. Let L_{FOL} be the set of all FOL formulae (words that parse according to Section 2.1) that are satisfiable. Then the formal decision problem is the following: given a word w, is $w \in L_{\mathsf{FOL}}$? In other words, is it a well-formed FOL formula, and if so, is it satisfiable? Dually, we can define the validity problem for PL and FOL.

2.6.2 Decidability

A formal language L is **decidable** if there exists a procedure that, given a word w, (1) eventually halts and (2) answers *yes* if $w \in L$ and *no* if $w \notin L$. Other terms for "decidable" are **recursive** and **Turing-decidable**. A procedure for a decidable language is called an **algorithm**. Satisfiability of PL formulae is decidable: the truth-table method is a decision procedure.

A formal language is **undecidable** if it is not decidable.

A formal language L is **semi-decidable** if there exists a procedure that, given a word w, (1) halts and answers *yes* iff $w \in L$, (2) halts and answers *no* if $w \notin L$, or (3) does not halt if $w \notin L$. The possible outcomes (2) and (3) for the case $w \notin L$ mean that the procedure may or may not halt. Unlike a decidable language, the procedure is only guaranteed to halt if $w \in L$. Other terms for "semi-decidable" are **partially decidable**, **recursively enumerable**, and **Turing-recognizable**.

The terms "Turing-decidable" and "Turing-recognizable" arise from Alan Turing's classic formalization of procedures as **Turing machines**. A Turing machine consists of a finite automaton coupled with an infinite tape and tape head. Each cell of the tape can hold one character from a finite alphabet. The state of the automaton changes based on its control structure and the character currently under the tape head. During a state change, the automaton instructs the tape head to write a new character to the current cell and to move one position left or right.

Church and Turing showed that L_{FOL} is undecidable: there does not exist an algorithm for deciding if a FOL formula F is satisfiable (and similarly for validity). However, there is a procedure that halts and says *yes* if F is valid, so validity is semi-decidable. We describe such a procedure based on the semantic argument method in Section 2.7.

2.6.3 ★Complexity

If a language is decidable, then one considers the complexity of the decision problem. We define several of the main complexity classes here. A language L is **polynomial-time decidable**, or in PTIME (also, in P), if there exists a procedure that, given w, answers *yes* when $w \in L$, answers *no* when $w \notin L$, and halts with the answer in a number of steps that is at most proportionate to some polynomial of the size of w. For example, determining if the word w is a well-formed FOL formula is polynomial-time decidable and can be implemented using standard parsing methods.

A language L is **nondeterministic-polynomial-time decidable**, or in NPTIME (also, in NP), if there exists a nondeterministic procedure that, given w,

1. guesses a **witness** W to the fact that $w \in L$ that is at most proportionate in size to some polynomial of the size of w;

2. checks in time at most proportionate to some polynomial of the size of w that W really is a witness to $w \in L$;
3. and answers *yes* if the check succeeds and *no* otherwise.

For example, L_{PL} is in NP, as exhibited by the following nondeterministic procedure for deciding if a PL formula is satisfiable:

1. parse the input w as formula F (return *no* if w is not a well-formed PL formula);
2. guess an interpretation I, which is linear in the size of w;
3. check that $I \models F$.

I is the witness to the satisfiability of F.

A language L is in **co-**NP if its complement language $\overline{L}$ is in NP. For example, unsatisfiability of PL formulae is in co-NP because satisfiability is in NP. It is not known if unsatisfiability of PL formulae is in NP. While a satisfiable PL formula has a polynomial size witness of its satisfiability (a satisfying interpretation I), there is no known polynomial size witness of unsatisfiability.

A language L is NP**-hard** if every instance $v \in L'$ of every other NP decidable language L' can be reduced to deciding an instance $w^v_{L'} \in L$. Moreover, the size of $w^v_{L'}$ must be at most proportionate to some polynomial of the size of v. That is, L is NP-hard if every query $v \in L'$ of every NP language L' can be encoded into a query $w \in L$, where w is not much larger than v. L is NP**-complete** if it is in NP and is NP-hard.

L_{PL} is NP-complete. Indeed, L_{PL}, also called SAT, was the first language shown to be NP-complete. We proved that L_{PL} is in NP above by describing a nondeterministic polynomial time procedure. The Cook-Levin theorem shows that all NP-languages L can be reduced to L_{PL}, so that L_{PL} is NP-hard. They exhibited a polynomial time algorithm that, given L and input w, constructs an encoding into PL of a simulation of a run of a nondeterministic Turing machine for L on w. The encoding as PL formula F has length that is polynomial in the length of w. F is satisfiable iff the Turing machine decides that $w \in L$.

For discussing the complexity of decision problems and algorithms, we use the standard notation. Consider a function $f(n)$ over integers. For example, $f(n) = \log n$, $f(n) = n^2$, $f(n) = 2^n$, or $f(n) = 2^{2^n}$. A function $g(n)$ is of at most **order** $f(n)$ if there exist a scalar $c \geq 0$ and an integer $n_0 \geq 0$ such that

$$\forall n \geq n_0.\ g(n) \leq cf(n)\ .$$

$O(f(n))$ denotes the set of all functions of at most order $f(n)$. Similarly, $\Omega(f(n))$ denotes the set of all function of at least order $f(n)$: a function $g(n)$ is of at least order $f(n)$ if there exist a scalar $c \geq 0$ and an an integer $n_0 \geq 0$ such that

$$\forall n \geq n_0.\ g(n) \geq cf(n)\ .$$

Finally, $\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$ denotes the set of all functions of precisely order $f(n)$.

Example 2.28.

- $3n^2 + n \in O(n^2)$
- $3n^2 + n \in \Omega(n^2)$
- $3n^2 + n \in \Theta(n^2)$
- $\frac{1}{99}n^2 + n \in \Omega(n^2)$
- $3n^2 + n \in O(2^n)$
- $3n^2 + n \in \Omega(n)$
- $3n^2 + n \notin \Omega(2^n)$
- $3n^2 + n \notin \Theta(2^n)$
- $2^n \in \Omega(n^3)$
- $2^n \notin O(n^3)$

■

A decision problem has time complexity $O(f(n))$ if there exists a decision algorithm P for the problem and a function $g(n) \in O(f(n))$ such that P runs in time at most $g(n)$ on input of size n. A decision problem has time complexity $\Omega(f(n))$ if there exists a function $g(n) \in \Omega(f(n))$ such that all decision algorithms P for the problem run in time at least $g(n)$ on input of size n. Finally, a decision problem has time complexity $\Theta(f(n))$ if it has time complexities $\Omega(f(n))$ and $O(f(n))$.

Example 2.29. The algorithm SAT for deciding PL satisfiability runs in time $\Theta(2^n)$, where n is the number of variables in the input formula, because each level of recursion branches. Hence, the problem of PL satisfiability has time complexity $O(2^n)$. ■

2.7 ★Meta-Theorems of First-Order Logic

We prove that the semantic argument method for FOL is sound and, given a proper strategy of applying the proof rules, complete. A proof method is **sound** if every formula that has a proof according to the method is valid. A proof method is **complete** if every formula that is valid has a proof according to the method. That the semantic argument method is sound means that a closed semantic argument for $I \not\models F$ proves the validity of F; and that the semantic argument method is complete means that every valid formula F of FOL has a closed semantic argument proving its validity. Because there exists a complete proof method for FOL, FOL is a complete logic: every valid formula of FOL has a proof of its validity.

The second half of this section is devoted to proving two classic theorems that we apply in Chapter 10.

2.7.1 Simplifying the Language of FOL

In preparation for the proofs, we simplify the language of FOL without losing expressiveness. Exercises 1.3 and 4.6 show that we have many redundant logical connectives. We choose to use only the logical constant $\top$ and the connectives $\neg$ and $\wedge$, from which the others can be constructed. Additionally, we need only one quantifier since $\exists x.\ F$ is equivalent to $\neg\forall x.\ \neg F$. We choose $\forall$.

A second simplification is more involved. The goal is to remove constant and function symbols from the language by using predicate symbols instead. Given a formula F, let S be the set of function symbols appearing in it. Associate with each n-ary function symbol f of S a new $(n+1)$-ary predicate p_f. Then for each occurrence of a function f in a literal L of F

$$L[f(t_1, \ldots, t_n)] \ ,$$

replace L in F with the new formula

$$\exists x.\ p_f(t_1, \ldots, t_n, x) \ \wedge \ L[x] \ .$$

After all replacements, the resulting formula G does not contain any function symbols.

The next step ensures that the new predicate p_f describes a function f: it associates with each tuple of domain values $\mathsf{v}_1, \ldots, \mathsf{v}_n$ precisely one value v. For each introduced predicate p_f, construct the formula

$$I_f \ : \ \forall \overline{x}.\ \exists y.\ p_f(\overline{x}, y) \ \wedge \ (\forall z.\ p_f(\overline{x}, z) \ \rightarrow \ y = z) \ .$$

Then construct the formula

$$H : \ \left(\bigwedge_{f \in S} I_f \right) \ \rightarrow \ G \ .$$

The equality predicate $=$ is not yet defined. To make $=$ an equivalence relation, assert that it is reflexive, symmetric, and transitive:

$$E : \begin{array}{l} (\forall x.\ x = x) \\ \ \wedge \ (\forall x, y.\ x = y \ \rightarrow \ y = x) \\ \ \wedge \ (\forall x, y, z.\ x = y \ \wedge \ y = z \ \rightarrow \ x = z) \end{array}$$

Additionally, every predicate symbol p appearing in G should obey $=$:

$$E_p : \ \forall \overline{x}, \overline{y}.\ \overline{x} = \overline{y} \ \rightarrow \ (p(\overline{x}) \leftrightarrow p(\overline{y}))$$

Let T be the set of predicate symbols of G. Construct the final formula

$$F' : \ \left(E \ \wedge \ \bigwedge_{p \in T} E_p \right) \ \rightarrow \ H \ .$$

F' is valid iff F is valid. Moreover, F' does not contain any function symbols.

For the special case of constant symbols, it is simpler to replace $F[a]$ with $F' : \forall x.\ F[x]$.

For the remainder of this section, we consider a version of FOL with only the logical constant $\top$, the connectives $\neg$ and $\wedge$, the quantifier $\forall$, and predicate symbols. It is equivalent in expressive power to the richer language studied earlier in the chapter.

2.7.2 Semantic Argument Proof Rules

In this simplified form of FOL, we need only the following seven proof rules.

- For handling negation:

$$\frac{I \models \neg F}{I \not\models F} \qquad\qquad \frac{I \not\models \neg F}{I \models F}$$

- For handling conjunction:

$$\frac{I \models F \wedge G}{\begin{array}{l} I \models F \\ I \models G \end{array}} \qquad\qquad \frac{I \not\models F \wedge G}{I \not\models F \quad \mid \quad I \not\models G}$$

- For handling universal quantification:

$$\frac{I \models \forall x.\ F}{I \triangleleft \{x \mapsto \mathsf{v}\} \models F} \quad \text{for any } \mathsf{v} \in D_I$$

  and

$$\frac{I \not\models \forall x.\ F}{I \triangleleft \{x \mapsto \mathsf{v}\} \not\models F} \quad \text{for a } \textit{fresh}\ \mathsf{v} \in D_I$$

- For deriving a contradiction:

$$\frac{\begin{array}{l} J : I \triangleleft \cdots \models p(x_1, \ldots, x_n) \\ K : I \triangleleft \cdots \not\models p(y_1, \ldots, y_n) \end{array}}{I \models \bot} \quad \text{for } i \in \{1, \ldots, n\},\ \alpha_J[x_i] = \alpha_K[y_i]$$

Only the second rule for conjunction requires a case analysis.

2.7.3 Soundness and Completeness

That the semantic argument method is sound is fairly obvious for the first six rules: each follows almost directly from the semantics of FOL. The final rule for deriving a contradiction requires some explanation. The variants J and K are constructed only through the rules for handling quantification so that they simply assign values to the arguments of p. Hence, the truth value of p on the given tuple of domain values is already established by I. Furthermore, the disagreement between J and K on the truth value of p indicates that I is not in fact an interpretation. Therefore, we have the following theorem.

Theorem 2.30 (Sound). *If every branch of a semantic argument proof of $I \not\models F$ closes, then F is valid.*

Completeness is more complicated. We want to show that there exists a closed semantic argument proof of $I \not\models F$ when F is valid. Our strategy is as follows. We define a procedure for applying the proof rules. When applying the quantification rules, the procedure selects values from a predetermined countably infinite domain. We then show that when some falsifying interpretation I exists (such that $I \not\models F$) our procedure constructs, *at the limit*, a falsifying interpretation. Therefore, F must be valid if the procedures actually discovers an argument in which all branches are closed. We now proceed according to this proof plan.

Let D be a countably infinite domain of values $\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \ldots$ which we can enumerate in some fixed order. Start the semantic argument by placing $I \not\models F$ at the root and marking it as *unused*. Now assume that the procedure has constructed a partial semantic argument and that each line is marked as either *used* or *unused*. We describe the next iteration.

Select the earliest line $L : I \models G$ or $L : I \not\models G$ in the argument that is marked *unused*, and choose the appropriate proof rule to apply according to the root symbol of G's parse tree. To apply a rule, add the appropriate deductions at the end of every open branch that passes through line L; mark each new deduction as *unused*; and mark L as *used*. The application of the negation rules and the first conjunction rule is then straightforward. Applying the second (branching) conjunction rule introduces a fork at the end of every open branch, doubling the number of open branches. In applying the second quantification rule, choose the next domain element v_i that does not appear in the semantic argument so far. For the first quantification rule, assume that G has the form $\forall x.\ H$. Choose the first value v_i on which $\forall x.\ H$ has not been instantiated in any ancestor of L. Additionally, consider $I \models G$ as a second "deduction" of this rule (so that both $I \triangleleft \{x \mapsto \mathsf{v}_i\} \models H$ and $I \models G$ are added to every branch passing through L and marked as *unused*). This trick guarantees that x of $\forall x.\ H$ is instantiated on every domain element without preventing the rest of the proof from progressing. Finally, close any branch that has a contradiction resulting from a deduction in this iteration.

Recall that a semantic argument is finished if no further applications of rules are possible. In our proof procedure, this situation occurs when all lines are marked as *used*. Although we can never construct a finished semantic argument with infinitely many lines in practice, we can reason about such arguments. For example, such an argument has an infinitely long branch. For suppose not: then every branch has finite length, so there must be an infinite number of these finite branches. But such a situation requires a deduction step that results in an infinite number of branches, whereas each proof rule produces at most two branches. This result is known as **König's Lemma**. We next prove that each open branch of a finished semantic argument describes a falsifying interpretation.

Lemma 2.31. *Each open branch of a finished semantic argument for $I \not\models F$ defines a falsifying interpretation of F.*

Proof. We apply structural induction on the formulae appearing in the branch to conclude that each line $L : I \models G$ or $L : I \not\models G$ holds, including $I \not\models F$. In that case, I is a falsifying interpretation of F. The technique of **structural induction** is defined in Section 4.4.

For the base case, consider lines in which G is an atom. As the contradiction rule is never applied (otherwise, the branch would be closed) no contradiction exists. Therefore, each instance $I \triangleleft \cdots \models p(x_1, \ldots, x_n)$ or $I \triangleleft \cdots \not\models p(x_1, \ldots, x_n)$ defines the truth value of p on one tuple of domain elements without contradicting any other definition on the branch. (For tuples of domain elements on which p is not explicitly defined on the branch, p may take any value, say false.)

Consider when G is formed by applying a logical connective to one or two formulae. As the procedure applied the appropriate proof rule, the inductive hypothesis and the semantics of the logical connectives tell us that L holds. For example, consider the case $L : I \models F_1 \wedge F_2$. Then $I \models F_1$ and $I \models F_2$ appear on the branch, and both lines hold by the inductive hypothesis. The reader may verify the other logical connectives with similar reasoning.

Consider the case $L : I \not\models \forall x.\ F$. For L to hold, it must be the case that for some fresh domain value v, $M : I \triangleleft \{x \mapsto \mathsf{v}\} \not\models F$. But the procedure guarantees that M is a descendant of L. Moreover, F is a subformula of $\forall x.\ F$, so the inductive hypothesis asserts that M holds. Then so does L.

Consider the case $L : I \models \forall x.\ F$. For L to hold, it must be the case that for all domain values v, $M : I \triangleleft \{x \mapsto \mathsf{v}\} \models F$. But the procedure guarantees that such a line exists for every v. Moreover, F is a subformula of $\forall x.\ F$, so the inductive hypothesis asserts that each such lines holds. Hence, so does L, finishing the proof. ■

Remark 2.32. The formulae that appear on an open branch of a finished semantic proof comprise a **Hintikka set**. The proof strategy that we employed is essentially that used in proving **Hintikka's Lemma**, which asserts that a Hintikka set is satisfiable.

Remark 2.33. We defined the procedure with a fixed countably infinite domain in mind and then proved that an open branch of a finished semantic argument corresponds to at least one falsifying interpretation. Therefore, we have proved an additional fact: every satisfiable FOL formula is satisfied by an interpretation with a countable domain. This result is **Löwenheim's Theorem**.

Theorem 2.34 (Complete). *The semantic argument method is complete: each valid formula F has a semantic argument proof (in which every branch is closed).*

Proof. Suppose that F is valid, yet no semantic argument proof exists. Then a finished semantic argument constructed according to our procedure has an open branch. By Lemma 2.31, this branch describes a falsifying interpretation of F, a contradiction. Hence, all branches of a finished semantic argument must in fact be closed (and thus finite). By König's Lemma, the semantic argument itself has finite size. ■

2.7.4 Additional Theorems

Is a countable (but possibly infinite) set S of satisfiable formulae **simultaneously satisfiable**? That is, does there exist a single interpretation I that satisfies every member of S? The **Compactness Theorem** relates simultaneous satisfiability of S to satisfiability of the conjunction of each finite subset of S. Dually, we might consider whether the disjunction of a countable set of formulae is valid.

Theorem 2.35 (Compactness Theorem). *A countable set of first-order formulae S is simultaneously satisfiable iff the conjunction of every finite subset is satisfiable.*

Proof. The forward direction is clear: if I simultaneously satisfies the members of S, then it satisfies each finite conjunction.

For the other direction, extend the proof procedure of the previous section as follows. Arrange the members of S in some order $F_1, F_2, F_3, \ldots$, which is possible because S is countable. Start the procedure with $I \not\models \neg F_1$. At the end of each iteration of the procedure, choose the next formula F_i in the sequence and append $I \not\models \neg F_i$ to every open branch, marking it as *unused.* Since each finite subset of S is satisfiable, at least one branch remains open and the procedure does not terminate.

A finished semantic argument constructed in this manner enumerates an interpretation that falsifies every $\neg F_i$ and thus satisfies every F_i. Hence, S is simultaneously satisfiable. ■

Remark 2.36. This proof proves an additional fact that extends Löwenheim's Theorem: every simultaneously satisfiable countable set of FOL formulae is simultaneously satisfied by an interpretation with a countable domain. This result is the **Löwenheim-Skolem Theorem**.

We apply the next theorem, the **Craig Interpolation Lemma**, in Chapter 10. It asserts that if $F \to G$ is valid, then there is a formula H (called an **interpolant**) such that $F \to H$ and $H \to G$ are valid and whose predicates and free variables occur in both F and G. The proof is constructive: it describes a procedure for extracting the interpolant from a proof of the validity of $F \to G$. However, proofs constructed via the proof rules of Section 2.7.2 are not directly amenable to the interpolation procedure. We describe an alternate set of proof rules instead and show that any proof constructed from the rules of 2.7.2 can be translated into a proof using the new rules. Then we prove the Craig Interpolation Lemma using these new proof rules.

One trick that will prove convenient is the following. Associate a fresh variable x_i with each domain value v_i introduced during the proof. Whenever a variant $I \triangleleft \{x \mapsto \mathsf{v}_i\}$ is used, rename x to the variable x_i corresponding to the value v_i in both the variant interpretation and the formula. This renaming does not affect the soundness of the proof, but it makes contradictions more obvious.

The new rules are the following:

- For handling double negation:

$$\begin{array}{l} I \models \neg\neg F \\ \hline I \models F \end{array} \qquad \begin{array}{l} I \not\models \neg\neg F \\ \hline I \not\models F \end{array}$$

- For handling conjunction:

$$\begin{array}{l} I \models F \wedge G \\ \hline I \models F \\ I \models G \end{array} \qquad \begin{array}{c} I \not\models F \wedge G \\ \hline I \not\models F \quad \mid \quad I \not\models G \end{array}$$

and

$$\begin{array}{c} I \models \neg(F \wedge G) \\ \hline I \models \neg F \quad \mid \quad I \models \neg G \end{array} \qquad \begin{array}{l} I \not\models \neg(F \wedge G) \\ \hline I \not\models \neg F \\ I \not\models \neg G \end{array}$$

- For handling universal quantification:

$$\begin{array}{l} I \models \forall x.\ F \\ \hline I \triangleleft \{x \mapsto \mathsf{v}\} \models F \end{array} \qquad \begin{array}{l} I \not\models \neg\forall x.\ F \\ \hline I \triangleleft \{x \mapsto \mathsf{v}\} \not\models \neg F \end{array} \qquad \text{for any } \mathsf{v} \in D_I$$

and

$$\begin{array}{l} I \not\models \forall x.\ F \\ \hline I \triangleleft \{x \mapsto \mathsf{v}\} \not\models F \end{array} \qquad \begin{array}{l} I \models \neg\forall x.\ F \\ \hline I \triangleleft \{x \mapsto \mathsf{v}\} \models \neg F \end{array} \qquad \text{for a } \textit{fresh}\ \mathsf{v} \in D_I$$

- For deriving a contradiction (recall our trick of renaming variables to correspond uniquely to domain values):

$$\begin{array}{l} J : I \triangleleft \cdots \models p(x_1, \ldots, x_n) \\ K : I \triangleleft \cdots \not\models p(x_1, \ldots, x_n) \\ \hline I \models \bot \end{array}$$

and

$$\begin{array}{l} J : I \triangleleft \cdots \models p(x_1, \ldots, x_n) \\ K : I \triangleleft \cdots \models \neg p(x_1, \ldots, x_n) \\ \hline I \models \bot \end{array} \qquad \begin{array}{l} J : I \triangleleft \cdots \not\models p(x_1, \ldots, x_n) \\ K : I \triangleleft \cdots \not\models \neg p(x_1, \ldots, x_n) \\ \hline I \models \bot \end{array}$$

The important characteristic (for proving the interpolation lemma) of this set of proof rules is that premises and deductions agree on the use of $\models$ or $\not\models$, except in the contradiction rules. In contrast, the negation rules of Section 2.7.2 do not have this property. We obtained this property by folding each negation rule into every other rule.

Before proving the interpolation lemma, let us prove that the new semantic argument proof system based on these rules is sound and complete. Soundness is fairly obvious; for completeness, we briefly describe how to map a proof from the system of Section 2.7.2 to a proof using these rules.

Lemma 2.37. *Every proof in the proof system of Section 2.7.2 has a corresponding proof in the new proof system.*

Proof. In constructing the new proof, ignore any use of the negation rules of Section 2.7.2, instead choosing from the (doubled) set of conjunction and quantification rules depending on whether a $\neg$ is at the root of the parse tree of a formula. Use the new negation rules to remove double negations when necessary. For deriving a contradiction, one of the three cases represented by the contradiction rules must occur when a contradiction occurs in the original proof. ■

We can now prove the theorem.

Theorem 2.38 (Craig Interpolation Lemma). *If $F \rightarrow G$ is valid, then there exists a formula H such that $F \rightarrow H$ and $H \rightarrow G$ are valid and whose predicates and free variables occur in both F and G.*

Proof. We prove the result by describing a procedure that extracts from a (closed) semantic argument proof of the validity of $F \rightarrow G$ the interpolant H. For convenience, let the proof itself begin with the lines

1.	$I \models F$	assumption
2.	$I \not\models G$	assumption

Notice that with the new set of proof rules, only $\models$ rules will be applied to deductions stemming from line 1, while only $\not\models$ rules will be applied to those stemming from line 2. The three contradiction rules correspond to three possible situations: a contradiction between $I \models F$ and $I \not\models G$ ($F \rightarrow G$ is valid), within $I \models F$ itself (F is unsatisfiable), and within $I \not\models G$ itself (G is valid).

The procedure runs backwards through a proof. It associates with each line L of the proof a set of *positive* formulae U and a set of *negative* formulae V. U consists of formulae on lines from which L descends (including itself) that are satisfied by their interpretation (lines of the form $K \models F_1$). V consists of formulae on lines from which L descends (including itself) that are falsified by their interpretation (lines of the form $K \not\models F_2$). Define L's **characteristic formula** as

$$\bigwedge U \rightarrow \bigvee V ,$$

written as $\{U\} \rightarrow \{V\}$ for concision. That the branch on which L lies ends in a contradiction implies that $\{U\} \rightarrow \{V\}$ is valid. The procedure constructs for each line an interpolant X of $\{U\} \rightarrow \{V\}$; that is, X is such that

$$\bigwedge U \Rightarrow X \quad \text{and} \quad X \Rightarrow \bigvee V$$

and the predicates and free variables of X appear in both U and V. The interpolant of line 2 of the proof is the interpolant H that we seek.

Let us begin with the end of a branch, $L : I \models \bot$. It must have been deduced via a contradiction. If the first contradiction rule produced L, then its characteristic formula is of the form

$$\{U, p(x_1, \ldots, x_n), \bot\} \rightarrow \{V, p(x_1, \ldots, x_n)\} ,$$

where the variable renaming trick ensures that the arguments to p are syntactically the same. Its parent has characteristic formula

$$\{U, p(x_1, \ldots, x_n)\} \rightarrow \{V, p(x_1, \ldots, x_n)\} ,$$

and both have interpolant $p(x_1, \ldots, x_n)$. If the second contradiction rule produced L, then its characteristic formula is of the form

$$\{U, p(x_1, \ldots, x_n), \neg p(x_1, \ldots, x_n), \bot\} \rightarrow \{V\} ,$$

and its parent's is of the form

$$\{U, p(x_1, \ldots, x_n), \neg p(x_1, \ldots, x_n)\} \rightarrow \{V\} .$$

Both have interpolant $\bot$ ($\neg\top$ in the restricted language). Similarly, if the third contradiction rule produced L, then the interpolant is $\top$.

Consider lines derived via the conjunction rules. Suppose $L : I \models F$ is deduced from $I \models F \wedge G$. Then the characteristic formulae of L and its parent are

$$\{U, F \wedge G, F\} \rightarrow \{V\} \quad \text{and} \quad \{U, F \wedge G\} \rightarrow \{V\} ,$$

respectively. If L has interpolant X, then so does its parent. The case is similar for a line $L : I \not\models \neg F$ deduced from $I \not\models \neg(F \wedge G)$.

For the next conjunction rule, suppose that $L : I \not\models F$ is deduced on one branch from $I \not\models F \wedge G$. Then L is at a fork in the proof and has sibling line $L' : I \not\models G$. The characteristic formulae of L, L', and their parent are

$$\{U\} \rightarrow \{V, F \wedge G, F\} , \ \{U\} \rightarrow \{V, F \wedge G, G\} , \text{ and } \{U\} \rightarrow \{V, F \wedge G\} ,$$

respectively. If L and L' have interpolants X and Y, respectively, then their parent has interpolant $X \wedge Y$.

Similarly, suppose $L : I \models \neg F$ is deduced on one branch from $I \models \neg(F \wedge G)$ (so that its sibling is $L' : I \models \neg G$). If L and L' have interpolants X and Y, respectively, then their parent has interpolant $X \vee Y$ ($\neg(\neg X \wedge \neg Y)$ in the restricted language).

The interpolant X of a line L derived via a double-negation rule passes directly to its parent, for the characteristic formula of L simply has a repetition $\neg\neg F$ of a formula F that the parent's characteristic formula does not have.

We turn to the quantification rules. Consider a line $L : I \triangleleft \{z \mapsto \mathsf{v}\} \not\models F$ derived from $I \not\models \forall x.\ F$. L and its parent M have characteristic formulae

$$\{U\} \rightarrow \{V, \forall x.\ F, F\} \quad \text{and} \quad \{U\} \rightarrow \{V, \forall x.\ F\}\ ,$$

respectively. Moreover, v is fresh, and thus z does not appear in either U or V according to our trick. Hence, z cannot occur free in L's interpolant X. It thus follows that

$$\forall *.\ \forall z.\ X \ \rightarrow\ V \ \vee\ \forall x.F \ \vee\ F$$

is equivalent to

$$\forall *.\ X \ \rightarrow\ V \ \vee\ \forall x.F \ \vee\ \forall z.\ F$$

and thus to

$$\forall *.\ X \ \rightarrow\ V \ \vee\ \forall x.F\ .$$

Therefore, X is an interpolant of M. Similarly, X is an interpolant of the parent of $L : I \triangleleft \{z \mapsto \mathsf{v}\} \models \neg F$ deduced from $I \models \neg\forall x.\ F$.

Consider $L : I \triangleleft \{z \mapsto \mathsf{v}\} \models F$ with interpolant X derived from $I \models \forall x.\ F$, where z is not necessarily fresh. The characteristic formulae of L and its parent M are

$$\{U, \forall x.\ F, F\} \rightarrow \{V\} \quad \text{and} \quad \{U, \forall x.\ F\} \rightarrow \{V\}\ ,$$

respectively. Clearly,

$$U \ \wedge\ \forall x.\ F \ \wedge\ F \ \Rightarrow\ X$$

implies that

$$U \ \wedge\ \forall x.\ F \ \Rightarrow\ X\ .$$

Hence, X is an interpolant of M when z is not free in U or V (so that z is not free in X) and when it is free in both. However, if z is free in V but not in U, then X is not an interpolant of M. But $\forall z.\ X$ is an interpolant. In particular, we have

$$U \ \wedge\ \forall x.\ F \ \Rightarrow\ \forall z.\ X$$

and

$$\forall z.\, X \;\Rightarrow\; V \quad \text{because} \quad \forall z.\, X \;\Rightarrow\; X \quad \text{and} \quad X \;\Rightarrow\; V \;.$$

For the final case, suppose that $L : I \triangleleft \{z \mapsto \mathsf{v}\} \not\models \neg F$ is deduced from $I \not\models \neg\forall x.\ F$. The characteristic formula of L is

$$\{U\} \to \{V, \neg\forall x.\ F, F\} \;.$$

Then X is the interpolant of the parent M unless z is free in U but not free in V. In the latter case, the interpolant is $\exists z.\ X$ ($\neg\forall z.\ \neg X$ in the restricted language). The reasoning is similar to the previous case, completing the proof. ■

2.8 Summary

Building on the presentation of PL in Chapter 1, this chapter introduces first-order logic (FOL). It covers:

- Its *syntax*. How one constructs a FOL formula. Variables, terms, function symbols, predicate symbols, atoms, literals, logical connectives, quantifiers.
- Its *semantics*. What a FOL formula means. Truth values true and false. Interpretations: domain and assignments. Difference between a function (predicate) symbol and a function (predicate) over a domain.
- *Satisfiability* and *validity*. Whether a FOL formula evaluates to true under any or all interpretations. Semantic argument method.
- *Substitution*, which is a tool for manipulating formulae and making general claims. Safe and schema substitutions. Substitution of equivalent formulae. Valid schemata.
- *Normal forms*. A normal form is a set of syntactically restricted formulae such that every FOL formula is equivalent to some member of the set.
- A review of *decidability and complexity theory*, which provides the concepts necessary for discussing decidability and complexity questions in logic.
- *Meta-theorems*. Semantic argument method is sound and complete. Compactness Theorem. Craig Interpolation Lemma.

The results of Section 2.7 are the groundwork for our theoretical treatment of the Nelson-Oppen combination method in Chapter 10.

FOL is the most general logic that is discussed in this book. Its applications include software and hardware design and analysis, knowledge representation, and complexity and decidability theory.

FOL is a complete logic: every valid FOL formula has a proof in the semantic argument method. However, validity is undecidable. Many applications benefit from complete automation, which is impossible when considering all of FOL. Therefore, Chapter 3 introduces first-order theories, which formalize interesting structures, such as integers, rationals, lists, stacks, and arrays. Part II of this book explores algorithms for reasoning within these theories.

Bibliographic Remarks

For a complete and concise presentation of propositional and first-order logic, see Smullyan's text *First-Order Logic* [87]. The semantic argument method is similar to Smullyan's tableau method. Also, the proofs of completeness of the semantic argument method, the Compactness Theorem, and the Craig Interpolation Lemma are inspired by Smullyan's presentation.

The history of the development of mathematical logic is rich. For an overview, see [98] and related articles in *The Stanford Encyclopedia of Philosophy.* We mention in particular *Hilbert's program* of the 1920s — see, for example, [38] — to find a consistent and complete axiomatization of arithmetic. Gödels two *incompleteness theorems* proved that such a goal is impossible. The first incompleteness theorem, which Gödel presented in a lecture in September, 1930, and then in [36], states that any axiomatization of arithmetic contains theorems that are not provable within the theory. The second, which Gödel had proved by October, 1930, states that a theory such as Peano arithmetic cannot prove its own consistency unless it is itself inconsistent. Earlier, Gödel proved that first-order logic is complete [35]: every theorem has a proof. However, Church — and, independently, Turing — proved that satisfiability in first-order logic is undecidable [13]. Thus, while every theorem of first-order logic has a finite proof, invalid formulae need not have a finite proof of their invalidity.

For an introduction to formal languages, decidability, and complexity theory, see [85, 72, 41].

Exercises

2.1 (English and FOL). Encode the following English sentences into FOL.

(*a*) Some days are longer than others.
(*b*) In all the world, there is but one place that I call home.
(*c*) My mother's mother is my grandmother.
(*d*) The intersection of two convex sets is convex.

2.2 (FOL validity & satisfiability). For each of the following FOL formulae, identify whether it is valid or not. If it is valid, prove it with a semantic argument; otherwise, identify a falsifying interpretation.

(*a*) $(\forall x, y.\ p(x, y) \rightarrow p(y, x)) \rightarrow \forall z.\ p(z, z)$
(*b*) $\forall x, y.\ p(x, y) \rightarrow p(y, x) \rightarrow \forall z.\ p(z, z)$
(*c*) $(\exists x.\ p(x)) \rightarrow \forall y.\ p(y)$
(*d*) $(\forall x.\ p(x)) \rightarrow \exists y.\ p(y)$
(*e*) $\exists x, y.\ (p(x, y) \rightarrow (p(y, x) \rightarrow \forall z.\ p(z, z)))$

2.3 (Semantic argument). Use the semantic argument method to prove the following formula schemata.

(a) $\neg(\forall x.\ F) \ \Leftrightarrow\ \exists x.\ \neg F$
(b) $\neg(\exists x.\ F) \ \Leftrightarrow\ \forall x.\ \neg F$
(c) $\forall x, y.\ F \ \Leftrightarrow\ \forall y, x.\ F$
(d) $\exists y.\ \forall x.\ F \ \Rightarrow\ \forall x.\ \exists y.\ F$
(e) $\exists x.\ F \lor G \ \Leftrightarrow\ (\exists x.\ F) \ \lor\ (\exists y.\ G)$
(f) $\exists x.\ F \rightarrow G \ \Leftrightarrow\ (\forall x.\ F) \ \rightarrow\ (\exists x.\ G)$
(g) $\exists x.\ F \lor G \ \Leftrightarrow\ (\exists x.\ F) \ \lor\ G$, provided $x \notin \mathsf{free}(G)$
(h) $\forall x.\ F \lor G \ \Leftrightarrow\ (\forall x.\ F) \ \lor\ G$, provided $x \notin \mathsf{free}(G)$
(i) $\exists x.\ F \land G \ \Leftrightarrow\ (\exists x.\ F) \ \land\ G$, provided $x \notin \mathsf{free}(G)$
(j) $\forall x.\ F \rightarrow G \ \Leftrightarrow\ (\exists x.\ F) \ \rightarrow\ G$, provided $x \notin \mathsf{free}(G)$

2.4 (Normal forms). Put the following formulae into prenex normal form.

(a) $(\forall x.\ \exists y.\ p(x, y)) \ \rightarrow\ \forall x.\ p(x, x)$
(b) $\exists z.\ (\forall x.\ \exists y.\ p(x, y)) \ \rightarrow\ \forall x.\ p(x, z)$
(c) $\forall w.\ \neg(\exists x, y.\ \forall z.\ p(x, z) \ \rightarrow\ q(y, z)) \ \land\ \exists z.\ p(w, z)$

2.5 (★Characteristic formula). Why is the characteristic formula of a line on a closed branch of a semantic argument valid?

3

First-Order Theories

> *Formalization works as "an early-warning system" when things are getting contorted.*
>
> — Edsger W. Dijkstra
> *EWD764: Repaying Our Debts*, 1980

When reasoning in particular application domains such as software or hardware, one often has particular structures in mind. For example, programs manipulate numbers, lists, and arrays. **First-order theories** formalize these structures to enable reasoning about them. This chapter introduces first-order theories in general and then focuses on theories useful in verification and related tasks. These theories include a theory of equality, of integers, of rationals and reals, of recursive data structures, and of arrays.

There is another reason to study first-order theories. While validity in FOL is undecidable, validity in particular theories or fragments of theories is sometimes decidable. Many of the theories studied in this chapter have important fragments for which validity is efficiently decidable. For each theory, we identify the decidable and efficiently decidable fragments, which we summarize in Section 3.7. Part II studies decision procedures for the decidable fragments.

3.1 First-Order Theories

A first-order **theory** T is defined by the following components.

1. Its **signature** Σ is a set of constant, function, and predicate symbols.
2. Its set of **axioms** $\mathcal{A}$ is a set of closed FOL formulae in which only constant, function, and predicate symbols of Σ appear.

A **Σ-formula** is constructed from constant, function, and predicate symbols of Σ, as well as variables, logical connectives, and quantifiers. As usual, the symbols of Σ are just symbols without prior meaning. The axioms $\mathcal{A}$ provide their meaning.

A Σ-formula F is **valid in the theory** T, or T**-valid**, if every interpretation I that satisfies the axioms of T,

$$I \models A \quad \text{for every } A \in \mathcal{A} , \tag{3.1}$$

also satisfies F: $I \models F$. For this reason, we write

$$T \models F$$

to mean that F is T-valid. Formally, the theory T consists of all (closed) formulae that are T-valid. We call an interpretation satisfying (3.1) a T**-interpretation**.

A Σ-formula F is **satisfiable in** T, or T**-satisfiable**, if there is a T-interpretation I that satisfies F.

A theory T is **complete** if for every closed Σ-formula F, $T \models F$ or $T \models \neg F$. A theory is **consistent** if there is at least one T-interpretation. In particular, in a consistent theory T, there does not exist a Σ-formula F such that both $T \models F$ and $T \models \neg F$. (Otherwise, by the semantics of conjunction, $T \models F \wedge \neg F$ and thus $T \models \bot$; but $\bot$ is not satisfied by any interpretation.)

Concepts from general FOL validity carry over to first-order theories in the natural way. For example, two formulae F_1 and F_2 are **equivalent in** T, or T**-equivalent**, if $T \models F_1 \leftrightarrow F_2$: for every T-interpretation I, $I \models F_1$ iff $I \models F_2$.

A **fragment** of a theory is a syntactically-restricted subset of formulae of the theory. For example, the **quantifier-free fragment** of a theory T is the set of formulae without quantifiers that are valid in T. Recall our convention that non-closed formula F is valid iff its universal closure is valid. Technically, the "quantifier-free fragment" of T actually consists of valid formulae in which all variables are universally quantified. However, the term "quantifier-free fragment" is the common and accepted name for this fragment. Subsequent chapters show that the quantifier-free fragments of theories are of great practical and theoretical importance.

A theory T is **decidable** if $T \models F$ is decidable for every Σ-formula F. That is, there is an algorithm that always terminates with "yes" if F is T-valid or with "no" if F is T-invalid. A fragment of T is decidable if $T \models F$ is decidable for every Σ-formula F that obeys the fragment's syntactic restrictions.

The union $T_1 \cup T_2$ of two theories T_1 and T_2 has signature $\Sigma_1 \cup \Sigma_2$ and axioms $\mathcal{A}_1 \cup \mathcal{A}_2$. Clearly, a $(T_1 \cup T_2)$-interpretation is both a T_1-interpretation and a T_2-interpretation since it satisfies the axioms of both T_1 and T_2. Hence, a formula that is T_1-valid or T_2-valid is $(T_1 \cup T_2)$-valid, while a formula that is $(T_1 \cup T_2)$-satisfiable is both T_1-satisfiable and T_2-satisfiable.

Because FOL (the "empty" theory, or the theory without axioms) is undecidable in general, we must turn to theories and fragments of theories for the possibility of fully automated reasoning. While many interesting theories are undecidable, there are several important theories and fragments of theories that are decidable. These theories and fragments are the main subject of Part

II of this book. We introduce them in the following sections. In Section 3.7, we summarize the decidability and complexity results for these theories and fragments.

3.2 Equality

The theory of equality T_{E} is the simplest first-order theory. Its signature

$$\Sigma_{\mathsf{E}} : \{=, a, b, c, \ldots, f, g, h, \ldots, p, q, r, \ldots\}$$

consists of

- = (equality), a binary predicate;
- and all constant, function, and predicate symbols.

Equality = is an **interpreted** predicate symbol: its meaning is defined via the axioms of T_{E}. The other constant, function, and predicate symbols are uninterpreted except as they relate to equality. The axioms of T_{E} are the following:

1. $\forall x.\ x = x$ (reflexivity)
2. $\forall x, y.\ x = y \ \rightarrow\ y = x$ (symmetry)
3. $\forall x, y, z.\ x = y \ \wedge\ y = z \ \rightarrow\ x = z$ (transitivity)
4. for each positive integer n and n-ary function symbol f,

$$\forall \overline{x}, \overline{y}.\ \left(\bigwedge_{i=1}^{n} x_i = y_i\right) \ \rightarrow\ f(\overline{x}) = f(\overline{y}) \qquad \text{(function congruence)}$$

5. for each positive integer n and n-ary predicate symbol p,

$$\forall \overline{x}, \overline{y}.\ \left(\bigwedge_{i=1}^{n} x_i = y_i\right) \ \rightarrow\ (p(\overline{x}) \leftrightarrow p(\overline{y})) \qquad \text{(predicate congruence)}$$

The notation $\overline{x}$ stands for the list of variables $x_1, \ldots, x_n$. Axioms (function congruence) and (predicate congruence) are actually axiom schemata. An **axiom schema** stands for a set of axioms, each an instantiation of the parameters (f and p in (function congruence) and (predicate congruence), respectively). For example, for binary function symbol f_2, (function congruence) instantiates to the following axiom:

$$\forall x_1, x_2, y_1, y_2.\ x_1 = y_1 \ \wedge\ x_2 = y_2 \ \rightarrow\ f_2(x_1, x_2) = f_2(y_1, y_2)\ .$$

The first three axioms state that = is an **equivalence relation**: it is a binary predicate that obeys reflexivity, symmetry, and transitivity. The final two axiom schemata formalize our intuition for the behavior of functions and predicates under equality. A function (predicate) always evaluates to the same value (truth value) for a given set of argument values. They assert that $=$ is a **congruence relation**.

T_{E} is just as undecidable as full FOL because it allows all constant, function, and predicate symbols. In particular, any FOL formula F can be encoded as a Σ_{E}-formula F' simply by replacing occurrences of the symbol $=$ with a fresh symbol. Since $=$ does not occur in this transformed formula F', the axioms of T_{E} are irrelevant; hence, F' is T_{E}-satisfiable iff F' is first-order satisfiable.

However, the quantifier-free fragment of T_{E} is both interesting and efficiently decidable, as we show in Chapter 9.

Example 3.1. Without quantifiers, free variables and constants play the same role. In the formula

$$F:\ a = b\ \wedge\ b = c\ \rightarrow\ g(f(a), b) = g(f(c), a)\ ,$$

a, b, and c are constants, while in

$$F':\ x = y\ \wedge\ y = z\ \rightarrow\ g(f(x), y) = g(f(z), x)\ ,$$

x, y, and z are free variables. F is T_{E}-valid iff F' is T_{E}-valid; F is T_{E}-satisfiable iff F' is T_{E}-satisfiable. ■

It is often useful to reason about the T-satisfiability or T-validity of a Σ-formula F in a structured but informal way. We show how to use the semantic argument method with T_{E}.

Example 3.2. To prove that

$$F:\ a = b\ \wedge\ b = c\ \rightarrow\ g(f(a), b) = g(f(c), a)$$

is T_{E}-valid, assume otherwise: there exists a T_{E}-interpretation I such that $I \not\models F$:

1.	$I \not\models F$	assumption
2.	$I \models a = b\ \wedge\ b = c$	1, →
3.	$I \not\models g(f(a), b) = g(f(c), a)$	1, →
4.	$I \models a = b$	2, ∧
5.	$I \models b = c$	2, ∧
6.	$I \models a = c$	4, 5, (transitivity)
7.	$I \models f(a) = f(c)$	6, (function congruence)
8.	$I \models b = a$	4, (symmetry)
9.	$I \models g(f(a), b) = g(f(c), a)$	7, 8 (function congruence)
10.	$I \models \bot$	3, 9

Our assumption is apparently false: F is T_{E}-valid. ■

3.3 Natural Numbers and Integers

Arithmetic involving the addition and multiplication of the natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ is perhaps the oldest of mathematical theories. In this section we describe three theories of arithmetic. **Peano arithmetic** allows addition and multiplication over natural numbers, while **Presburger arithmetic** is restricted to addition over natural numbers. The final theory, the **theory of integers**, is convenient for automated reasoning but is no more expressive than Presburger arithmetic.

3.3.1 Peano Arithmetic

The **theory of Peano arithmetic** T_{PA}, or **first-order arithmetic**, has signature

$$\Sigma_{\mathsf{PA}} : \ \{0,\ 1,\ +,\ \cdot,\ =\}\ ,$$

where

- 0 and 1 are constants;
- $+$ (addition) and $\cdot$ (multiplication) are binary functions;
- and $=$ (equality) is a binary predicate.

Its axioms are the following:

1. $\forall x.\ \neg(x + 1 = 0)$ (zero)
2. $\forall x, y.\ x + 1 = y + 1 \ \rightarrow\ x = y$ (successor)
3. $F[0]\ \wedge\ (\forall x.\ F[x] \rightarrow F[x + 1])\ \rightarrow\ \forall x.\ F[x]$ (induction)
4. $\forall x.\ x + 0 = x$ (plus zero)
5. $\forall x, y.\ x + (y + 1) = (x + y) + 1$ (plus successor)
6. $\forall x.\ x \cdot 0 = 0$ (times zero)
7. $\forall x, y.\ x \cdot (y + 1) = x \cdot y + x$ (times successor)

These axioms concisely define addition, multiplication, and equality over natural numbers. Informally, axioms (zero), (plus zero), and (times zero) define 0 as we understand it: it is the minimal element of the natural numbers; it is the identity for addition ($x + 0 = x$); and under multiplication, it maps any number to 0 ($x \cdot 0 = 0$). Axioms (zero), (successor), (plus zero), and (plus successor) define addition. Axioms (times zero) and (times successor) define multiplication: in particular, (times successor) defines multiplication in terms of addition.

(induction) is an axiom schema: it stands for the set of axioms obtained by substituting for F each Σ_{PA}-formula that has precisely one free variable. It asserts that every T_{PA}-interpretation I obeys induction: if I satisfies $F[0]$ and $\forall x.\ F[x] \rightarrow F[x + 1]$, then I also satisfies $\forall x.\ F[x]$.

For convenience, we usually do not write the "$\cdot$" for multiplication. For example, we write xy rather than $x \cdot y$.

The **intended interpretations** of T_{PA} have domain $\mathbb{N}$ and assignments α_I defining 0, 1, +, $\cdot$, and = as we understand them in everyday arithmetic. In particular,

- $\alpha_I[0]$ is $0_{\mathbb{N}}$: α_I maps the symbols "0" to $0_{\mathbb{N}} \in \mathbb{N}$;
- $\alpha_I[1]$ is $1_{\mathbb{N}}$: α_I maps the symbols "1" to $1_{\mathbb{N}} \in \mathbb{N}$;
- $\alpha_I[+]$ is $+_{\mathbb{N}}$, addition over $\mathbb{N}$;
- $\alpha_I[\cdot]$ is $\cdot_{\mathbb{N}}$, multiplication over $\mathbb{N}$;
- $\alpha_I[=]$ is $=_{\mathbb{N}}$, equality over $\mathbb{N}$.

Example 3.3. The formula $3x + 5 = 2y$ can be written using the signature Σ_{PA} as

$$x + x + x + 1 + 1 + 1 + 1 + 1 = y + y$$

or as

$$(1 + 1 + 1) \cdot x + 1 + 1 + 1 + 1 + 1 = (1 + 1) \cdot y \ .$$

In practice, we use the abbreviated notation $3x + 5 = 2y$. ■

Example 3.4. Rather than augmenting T_{PA} with axioms defining inequality $>$, we can transform formulae with inequality into formulae over the restricted signature Σ_{PA}. Write

$$3x + 5 > 2y \quad \text{as} \quad \exists z.\ z \neq 0\ \wedge\ 3x + 5 = 2y + z \ ,$$

where $z \neq 0$ abbreviates $\neg(z = 0)$. The latter formula is a Σ_{PA}-formula. Weak inequality can be similarly transformed. Write

$$3x + 5 \geq 2y \quad \text{as} \quad \exists z.\ 3x + 5 = 2y + z \ .$$

■

Example 3.5. The Σ_{PA}-formula

$$\exists x, y, z.\ x \neq 0\ \wedge\ y \neq 0\ \wedge\ z \neq 0\ \wedge\ xx + yy = zz$$

is T_{PA}-valid. It asserts that there exists a triple of positive integers fulfilling the Pythagorean Theorem. The formula

$$\exists x, y, z.\ x \neq 0\ \wedge\ y \neq 0\ \wedge\ z \neq 0\ \wedge\ xxx + yyy = zzz$$

is the cubic analogue. For constant n, let x^n represent n multiplications of x; then every formula of the set

$$\{\forall x, y, z.\ x \neq 0\ \wedge\ y \neq 0\ \wedge\ z \neq 0\ \rightarrow\ x^n + y^n \neq z^n\ :\ n > 2\ \wedge\ n \in \mathbb{Z}\}$$

is T_{PA}-valid, as claimed by Fermat's Last Theorem and proved by Andrew Wiles in 1994. ■

Remark 3.6. Gödel's first incompleteness theorem (see **Bibliographic Remarks** of Chapter 2) implies that Peano arithmetic T_{PA} does not capture true arithmetic: there exist closed Σ_{PA}-formulae representing valid propositions of number theory that are T_{PA}-invalid. Gödel's proof constructs such a formula: it encodes the assertion that the formula itself cannot be proved. Now, either this formula can be proved from the axioms of T_{PA} (contradicting itself so that T_{PA} is inconsistent) or it cannot be proved (so that T_{PA} is incomplete).

Satisfiability and validity in T_{PA} is undecidable. Therefore, we turn to a more restricted theory of arithmetic that does not allow multiplication.

3.3.2 Presburger Arithmetic

The **theory of Presburger arithmetic** $T_{\mathbb{N}}$ has signature

$$\Sigma_{\mathbb{N}} : \ \{0,\ 1,\ +,\ =\}\ ,$$

where

- 0 and 1 are constants;
- $+$ (addition) is a binary function;
- and $=$ (equality) is a binary predicate.

Its axioms are a subset of the axioms of T_{PA}:

1. $\forall x.\ \neg(x + 1 = 0)$ (zero)
2. $\forall x, y.\ x + 1 = y + 1\ \rightarrow\ x = y$ (successor)
3. $F[0]\ \wedge\ (\forall x.\ F[x] \rightarrow F[x+1])\ \rightarrow\ \forall x.\ F[x]$ (induction)
4. $\forall x.\ x + 0 = x$ (plus zero)
5. $\forall x, y.\ x + (y + 1) = (x + y) + 1$ (plus successor)

Again, (induction) is an axiom schema standing for the set of axioms obtained by replacing F with each $\Sigma_{\mathbb{N}}$-formula that has precisely one free variable.

The intended interpretations of $T_{\mathbb{N}}$ have domain $\mathbb{N}$ and are such that

- $\alpha_I[0]$ is $0_{\mathbb{N}} \in \mathbb{N}$;
- $\alpha_I[1]$ is $1_{\mathbb{N}} \in \mathbb{N}$;
- $\alpha_I[+]$ is $+_{\mathbb{N}}$, addition over $\mathbb{N}$;
- $\alpha_I[=]$ is $=_{\mathbb{N}}$, equality over $\mathbb{N}$.

How does one reason about all integers, $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$? Such formulae can be encoded as $\Sigma_{\mathbb{N}}$-formulae.

Example 3.7. Consider the formula

$$F_0 :\ \forall w, x.\ \exists y, z.\ x + 2y - z - 13 > -3w + 5\ ,$$

where $-$ is meant to be interpreted as standard subtraction, and w, x, y, and z are intended to range over $\mathbb{Z}$. The formula

$$F_1 : \begin{array}{l} \forall w_p, w_n, x_p, x_n.\ \exists y_p, y_n, z_p, z_n. \\ \quad (x_p - x_n) + 2(y_p - y_n) - (z_p - z_n) - 13 > -3(w_p - w_n) + 5 \end{array}$$

introduces two variables, v_p and v_n, for each variable v of F_0. While each of v_p and v_n can only range over $\mathbb{N}$, $v_p - v_n$ should range over the integers. But how is $-$ interpreted? Moving negated terms to the other side of the inequality eliminates $-$:

$$F_2 : \begin{array}{l} \forall w_p, w_n, x_p, x_n.\ \exists y_p, y_n, z_p, z_n. \\ \quad x_p + 2y_p + z_n + 3w_p > x_n + 2y_n + z_p + 13 + 3w_n + 5\ . \end{array}$$

The final transformation eliminates constant coefficients and strict inequality:

$$F_3 : \begin{array}{l} \forall w_p, w_n, x_p, x_n.\ \exists y_p, y_n, z_p, z_n.\ \exists u. \\ \quad \neg(u = 0) \land \\ \quad x_p + y_p + y_p + z_n + w_p + w_p + w_p \\ \qquad = x_n + y_n + y_n + z_p + w_n + w_n + w_n + u \\ \qquad\quad + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ \qquad\quad + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1\ . \end{array}$$

■

Presburger showed in 1929 that $T_{\mathbb{N}}$ is decidable. Therefore, the "theory of (negative and positive) integers" that we loosely constructed above is also decidable via the syntactic rewriting of formulae into $\Sigma_{\mathbb{N}}$-formulae. Rather than using this cumbersome rewriting, however, we next study a theory of integers.

3.3.3 Theory of Integers

The **theory of integers** $T_{\mathbb{Z}}$ has signature

$$\Sigma_{\mathbb{Z}} : \{\ldots, -2, -1, 0,\ 1,\ 2,\ \ldots, -3\cdot, -2\cdot,\ 2\cdot,\ 3\cdot,\ \ldots,\ +,\ -,\ =,\ >\}\ ,$$

where

- $\ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots$ are constants, intended to be assigned the obvious corresponding values in the intended domain of integers $\mathbb{Z}$;
- $\ldots,\ -3\cdot,\ -2\cdot,\ 2\cdot,\ 3\cdot,\ \ldots$ are unary functions, intended to represent constant coefficients (*e.g.*, $2 \cdot x$, abbreviated $2x$);
- $+$ and $-$ are binary functions, intended to represent the obvious corresponding functions over $\mathbb{Z}$;
- $=$ and $>$ are binary predicates, intended to represent the obvious corresponding predicates over $\mathbb{Z}$.

Since Example 3.7 shows that $\Sigma_{\mathbb{Z}}$-formulae can be reduced to $\Sigma_{\mathbb{N}}$-formulae, we do not axiomatize $T_{\mathbb{Z}}$. $T_{\mathbb{Z}}$ is merely a convenient representation for reasoning about addition over all integers.

The intended interpretations of $T_{\mathbb{Z}}$ have domain $\mathbb{Z}$ and are such that α_I assigns the obvious values, functions, and predicates to the constant, function, and predicate symbols of $\Sigma_{\mathbb{Z}}$.

In Chapter 7, we discuss Cooper's decision procedure for deciding $T_{\mathbb{Z}}$-validity, while in Chapter 8, we discuss decision procedures for the quantifier-free fragment of $T_{\mathbb{Z}}$. These procedures decide $T_{\mathbb{N}}$-validity as well: the following example illustrates that $\Sigma_{\mathbb{N}}$-formulae can be reduced to $\Sigma_{\mathbb{Z}}$-formulae.

Example 3.8. To decide the $T_{\mathbb{N}}$-validity of

$$\forall x.\ \exists y.\ x = y + 1\ ,$$

decide the $T_{\mathbb{Z}}$-validity of

$$\forall x.\ x \geq 0\ \rightarrow\ \exists y.\ y \geq 0\ \wedge\ x = y + 1\ ,$$

where $t_1 \geq t_2$ expands to $t_1 = t_2\ \vee\ t_1 > t_2$. ■

We prove the validity of several $\Sigma_{\mathbb{Z}}$-formulae using the semantic argument method. Our application of the semantic argument method in this context is informal; it is intended to allow us to argue intuitively about validity until Chapter 7.

Example 3.9. To prove that the $\Sigma_{\mathbb{Z}}$-formula

$$F:\ \forall x, y, z.\ x > z\ \wedge\ y \geq 0\ \rightarrow\ x + y > z\ .$$

is $T_{\mathbb{Z}}$-valid, assume otherwise: there is a $T_{\mathbb{Z}}$-interpretation I such that $I \not\models F$:

$$\begin{array}{lll}
1. & I\ \not\models\ F & \text{assumption} \\
2. & I_1 :\ I \triangleleft \{x \mapsto \mathsf{v}_1\} \triangleleft \{y \mapsto \mathsf{v}_2\} \triangleleft \{z \mapsto \mathsf{v}_3\} & \\
 & \quad\ \not\models\ x > z\ \wedge\ y \geq 0\ \rightarrow\ x + y > z & 1, \forall \\
3. & I_1\ \models\ x > z\ \wedge\ y \geq 0 & 2, \rightarrow \\
4. & I_1\ \not\models\ x + y > z & 2, \rightarrow \\
5. & I_1\ \models\ \neg(x + y > z) & 4, \neg
\end{array}$$

We derive a contradiction by collecting formulae from lines 3 and 5, applying the variant interpretation I_1, and querying the theory $T_{\mathbb{Z}}$: are there integers $\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3$ such that

$$\mathsf{v}_1 > \mathsf{v}_3\ \wedge\ \mathsf{v}_2 \geq 0\ \wedge\ \neg(\mathsf{v}_1 + \mathsf{v}_2 > \mathsf{v}_3)\ ?$$

No, for $\mathsf{v}_1 > \mathsf{v}_3\ \wedge\ \mathsf{v}_2 \geq 0$ implies $\mathsf{v}_1 + \mathsf{v}_2 > \mathsf{v}_3$. We summarize this reasoning in $T_{\mathbb{Z}}$ with the line

$$\begin{array}{lll}
6. & I_1\ \models\ \bot & 3,\ 5,\ T_{\mathbb{Z}}
\end{array}$$

Therefore, F is $T_{\mathbb{Z}}$-valid. ■

Arguing the validity of arithmetic formulae at the level of axioms is tedious. Therefore, unlike in Example 3.2 in which the theory-specific reasoning is incorporated into the semantic argument method by applying and stating specific axioms of T_{E}, the semantic argument method for $T_{\mathbb{Z}}$ handles the "logical" aspects of the structured reasoning, while a separate informal argument reasons about the theory-specific elements.

Example 3.10. To prove that the $\Sigma_{\mathbb{Z}}$-formula

$$F : \forall x, y.\ x > 0 \ \land\ (x = 2y \ \lor\ x = 2y + 1) \ \rightarrow\ x - y > 0$$

is $T_{\mathbb{Z}}$-valid, assume otherwise: there is a $T_{\mathbb{Z}}$-interpretation I such that $I \not\models F$:

$$\begin{array}{lll}
1. & I \ \not\models\ F & \text{assumption} \\
2. & I_1 : \ I \lhd \{x \mapsto \mathsf{v}_1\} \lhd \{y \mapsto \mathsf{v}_2\} & \\
 & \quad \not\models\ x > 0 \ \land\ (x = 2y \ \lor\ x = 2y + 1) \ \rightarrow\ x - y > 0 & \\
 & & 1, \forall \\
3. & I_1 \ \models\ x > 0 \ \land\ (x = 2y \ \lor\ x = 2y + 1) & 2, \rightarrow \\
4. & I_1 \ \models\ x > 0 & 3, \land \\
5. & I_1 \ \models\ x = 2y \ \lor\ x = 2y + 1 & 3, \land \\
6. & I_1 \ \not\models\ x - y > 0 & 2, \rightarrow \\
7. & I_1 \ \models\ \neg(x - y > 0) & 6, \neg
\end{array}$$

There are two cases to consider. In the first case,

$$8a. \quad I_1 \ \models\ x = 2y \qquad 5, \lor$$

We collect the formulae of lines 4, 7, and $8a$, apply the variant interpretation I_1, and query the theory $T_{\mathbb{Z}}$: are there integers $\mathsf{v}_1, \mathsf{v}_2$ such that

$$\mathsf{v}_1 > 0 \ \land\ \mathsf{v}_1 = 2\mathsf{v}_2 \ \land\ \neg(\mathsf{v}_1 - \mathsf{v}_2 > 0) \ ?$$

No, for substituting $\mathsf{v}_1 = 2\mathsf{v}_2$ throughout produces

$$2\mathsf{v}_2 > 0 \ \land\ \neg(2\mathsf{v}_2 - \mathsf{v}_2 > 0) \ ,$$

which simplifies to

$$\mathsf{v}_2 > 0 \ \land\ \neg(\mathsf{v}_2 > 0) \ ,$$

a contradiction. This reasoning is summarized by

$$9a. \quad I_1 \ \models\ \bot \qquad 4,\ 7,\ 8a,\ T_{\mathbb{Z}}$$

In the second case,

$$8b. \quad I_1 \ \models\ x = 2y + 1 \qquad 5, \lor$$

Considering lines 4, 7, and $8b$, are there integers $\mathsf{v}_1, \mathsf{v}_2$ such that

$$\mathsf{v}_1 > 0 \;\wedge\; \mathsf{v}_1 = 2\mathsf{v}_2 + 1 \;\wedge\; \neg(\mathsf{v}_1 - \mathsf{v}_2 > 0) \;?$$

No, for substituting $\mathsf{v}_1 = 2\mathsf{v}_2 + 1$ throughout produces

$$2\mathsf{v}_2 + 1 > 0 \;\wedge\; \neg(2\mathsf{v}_2 + 1 - \mathsf{v}_2 > 0) \;,$$

which simplifies to

$$2\mathsf{v}_2 + 1 > 0 \;\wedge\; \neg(\mathsf{v}_2 + 1 > 0) \;.$$

The first literal holds only when $\mathsf{v}_2 > -1$, while the second holds only when $\mathsf{v}_2 \leq -1$, a contradiction. This reasoning is summarized by

$$9b. \quad I_1 \;\models\; \bot \qquad 4,\,7,\,8b,\,T_{\mathbb{Z}}$$

Thus, F is $T_{\mathbb{Z}}$-valid. ■

3.4 Rationals and Reals

Almost as old as arithmetic on integers is arithmetic on the rational numbers $\mathbb{Q}$ and (not quite as old) on the real numbers $\mathbb{R}$. In this section, we describe two theories of real arithmetic. The latter theory can also be seen as a theory of rational arithmetic.

The first theory is the **theory of reals**, involving addition and multiplication over $\mathbb{R}$; it is also known as **elementary algebra**. The term "elementary" refers to the restriction that variables range only over domain elements (as in all first-order theories), not over sets or functions of domain elements. Most junior high students are familiar with elementary algebra. As a first-order theory, elementary algebra is of course more complex since formulae are constructed with logical connectives and quantifiers.

The second theory is the theory of addition over $\mathbb{R}$ or $\mathbb{Q}$. Interpretations with domains of $\mathbb{R}$ are indistinguishable from interpretations with domains of $\mathbb{Q}$, as we discuss below. For this reason, we call this second theory the **theory of rationals**.

Example 3.11. Let us distinguish informally between the theories of integers (with only addition), reals (with addition and multiplication), and rationals (with only addition).

In the theory of integers,

$$F:\; \exists x.\; 2x = 7$$

is $T_{\mathbb{Z}}$-invalid. However, assigning to x the rational number $\frac{7}{2}$ satisfies $2x = 7$, so F should be satisfiable in the theory of rationals. Moreover, $\frac{7}{2}$ is also a real number, so F should also be satisfiable in the theory of reals.

The theory of reals includes multiplication, allowing a formula like

$$G : \; \exists x.\; x^2 = 2$$

to be expressed, where x^2 abbreviates $x \cdot x$. G should be valid in the theory of reals because assigning to x the real number $\sqrt{2}$ satisfies $x^2 = 2$. $\sqrt{2}$ is irrational. ■

3.4.1 Theory of Reals

The **theory of reals** $T_{\mathbb{R}}$, or **elementary algebra**, has signature

$$\Sigma_{\mathbb{R}} : \; \{0,\; 1,\; +,\; -,\; \cdot,\; =,\; \geq\} \;,$$

where

- 0 and 1 are constants;
- $+$ (addition) and $\cdot$ (multiplication) are binary functions;
- $-$ (negation) is a unary function;
- and $=$ (equality) and $\geq$ (weak inequality) are binary predicates.

$T_{\mathbb{R}}$ has the most complex axiomatization of the theories that we study. We group axioms by their mathematical content.

First are the axioms of an **abelian group**. An abelian group is a structure with additive identity 0, associative and commutative addition $+$, additive inverse $-$, and equality $=$. The qualifier "abelian" simply means that addition is commutative. The axioms are the following:

1. $\forall x, y, z.\; (x + y) + z = x + (y + z)$ (+ associativity)
2. $\forall x.\; x + 0 = x$ (+ identity)
3. $\forall x.\; x + (-x) = 0$ (+ inverse)
4. $\forall x, y.\; x + y = y + x$ (+ commutativity)

The first three axioms are the axioms of a **group**.

Second are the additional axioms of a **ring**. A ring is an abelian group with a multiplicative identity 1 and associative multiplication $\cdot$ that distributes over addition. For convenience, we usually shorten $x \cdot y$ to xy.

1. $\forall x, y, z.\; (xy)z = x(yz)$ (· associativity)
2. $\forall x.\; x1 = x$ (· left identity)
3. $\forall x.\; 1x = x$ (· right identity)
4. $\forall x, y, z.\; x(y + z) = xy + xz$ (left distributivity)
5. $\forall x, y, z.\; (x + y)z = xz + yz$ (right distributivity)

Both left and right identity and distributivity axioms are required since $\cdot$ is not commutative (yet). It is made so in the next set of axioms.

Third are the additional axioms of a **field**. In a field, $\cdot$ is commutative; the additive and multiplicative identities are different; and the multiplicative inverse of a non-0 value exists (*e.g.*, $\frac{1}{2}$ is the multiplicative inverse of 2).

1. $\forall x, y.\; xy = yx$ (· commutativity)

2. $0 \neq 1$ (separate identities)
3. $\forall x.\ x \neq 0 \ \rightarrow\ \exists y.\ xy = 1$ (· inverse)

The axiom (· commutativity) makes the (· right identity) and (right distributivity) axioms redundant.

Fourth are the additional axioms characterizing $\geq$ as a **total order**.

1. $\forall x, y.\ x \geq y \ \land\ y \geq x \ \rightarrow\ x = y$ (antisymmetry)
2. $\forall x, y, z.\ x \geq y \ \land\ y \geq z \ \rightarrow\ x \geq z$ (transitivity)
3. $\forall x, y.\ x \geq y \ \lor\ y \geq x$ (totality)

Finally are the additional axioms of a **real closed field**.

1. $\forall x, y, z.\ x \geq y \ \rightarrow\ x + z \geq y + z$ (+ ordered)
2. $\forall x, y.\ x \geq 0 \ \land\ y \geq 0 \ \rightarrow\ xy \geq 0$ (· ordered)
3. $\forall x.\ \exists y.\ x = y^2 \ \lor\ x = -y^2$ (square-root)
4. for each odd integer n,

$$\forall \overline{x}.\ \exists y.\ y^n + x_1 y^{n-1} + \cdots + x_{n-1} y + x_n = 0 \qquad \text{(at least one root)}$$

We again abbreviate $x_1, \ldots, x_n$ by $\overline{x}$. By y^n, we mean y multiplied by itself n times: $y \cdots y$. The axioms (+ ordered) and (· ordered) assert that every $T_{\mathbb{R}}$-interpretation is an **ordered field**. The axiom (square-root) asserts the existence of the square-root of every value. The final axiom schema states that polynomials of odd degree have at least one root.

Putting all axioms together and pruning redundant axioms, we have:

1. $\forall x, y.\ x \geq y \ \land\ y \geq x \ \rightarrow\ x = y$ (antisymmetry)
2. $\forall x, y, z.\ x \geq y \ \land\ y \geq z \ \rightarrow\ x \geq z$ (transitivity)
3. $\forall x, y.\ x \geq y \ \lor\ y \geq x$ (totality)

4. $\forall x, y, z.\ (x + y) + z = x + (y + z)$ (+ associativity)
5. $\forall x.\ x + 0 = x$ (+ identity)
6. $\forall x.\ x + (-x) = 0$ (+ inverse)
7. $\forall x, y.\ x + y = y + x$ (+ commutativity)
8. $\forall x, y, z.\ x \geq y \ \rightarrow\ x + z \geq y + z$ (+ ordered)

9. $\forall x, y, z.\ (xy)z = x(yz)$ (· associativity)
10. $\forall x.\ 1x = x$ (· identity)
11. $\forall x.\ x \neq 0 \ \rightarrow\ \exists y.\ xy = 1$ (· inverse)
12. $\forall x, y.\ xy = yx$ (· commutativity)
13. $\forall x, y.\ x \geq 0 \ \land\ y \geq 0 \ \rightarrow\ xy \geq 0$ (· ordered)

14. $\forall x, y, z.\ x(y + z) = xy + xz$ (distributivity)
15. $0 \neq 1$ (separate identities)

16. $\forall x.\ \exists y.\ x = y^2 \ \lor\ -x = y^2$ (square-root)

17. for each odd integer n,

$$\forall \overline{x}.\ \exists y.\ y^n + x_1 y^{n-1} + \cdots + x_{n-1} y + x_n = 0 \qquad \text{(at least one root)}$$

Example 3.12. The method of **quantifier elimination**, which we study in Chapter 7, eliminates quantifiers from a formula to produce an equivalent quantifier-free formula. If a formula F contains free variables, then a quantifier elimination procedure produces an equivalent quantifier-free formula F' such that $\mathsf{free}(F') \subseteq \mathsf{free}(F)$. For example, when is the formula

$$F :\ \exists x.\ ax^2 + bx + c = 0$$

satisfiable? That is, what are the conditions on a, b, and c such that a quadratic polynomial has a real root? Recall that the discriminant must be nonnegative:

$$F' :\ b^2 - 4ac \geq 0\ .$$

F' is the quantifier-free formula that is $T_{\mathbb{R}}$-equivalent to F. ■

Tarski proved that $T_{\mathbb{R}}$ was decidable in the 1930s, although the Second World War prevented his publishing the result until 1956. Collins proposed the more efficient technique of **cylindrical algebraic decomposition** (CAD) in 1975. Unfortunately, even the most efficient decision procedures for $T_{\mathbb{R}}$ have prohibitively high time complexity: CAD runs in time proportionate to $2^{2^{k|F|}}$, for some constant k and for $|F|$ the length of $\Sigma_{\mathbb{R}}$-formula F.

3.4.2 Theory of Rationals

Given the high complexity of deciding $T_{\mathbb{R}}$-validity (and the high intellectual complexity of Tarski's and subsequent decision procedures for $T_{\mathbb{R}}$), we turn to a simpler theory without multiplication, the **theory of rationals** $T_{\mathbb{Q}}$. It has signature

$$\Sigma_{\mathbb{Q}} :\ \{0,\ 1,\ +,\ -,\ =,\ \geq\}\ ,$$

where

- 0 and 1 are constants;
- $+$ (addition) is a binary function;
- $-$ (negation) is a unary function;
- and $=$ (equality) and $\geq$ (weak inequality) are binary predicates.

Its axioms are the following:

1. $\forall x, y.\ x \geq y\ \wedge\ y \geq x\ \rightarrow\ x = y$ (antisymmetry)
2. $\forall x, y, z.\ x \geq y\ \wedge\ y \geq z\ \rightarrow\ x \geq z$ (transitivity)
3. $\forall x, y.\ x \geq y\ \vee\ y \geq x$ (totality)

4. $\forall x, y, z.\ (x + y) + z = x + (y + z)$ (+ associativity)
5. $\forall x.\ x + 0 = x$ (+ identity)
6. $\forall x.\ x + (-x) = 0$ (+ inverse)
7. $\forall x, y.\ x + y = y + x$ (+ commutativity)
8. $\forall x, y, z.\ x \geq y \ \rightarrow\ x + z \geq y + z$ (+ ordered)

9. for each positive integer n,

$$\forall x.\ nx = 0 \ \rightarrow\ x = 0 \qquad \text{(torsion-free)}$$

10. for each positive integer n,

$$\forall x.\ \exists y.\ x = ny \qquad \text{(divisible)}$$

By nx we mean x added to itself n times: $x + \cdots + x$. The first eight axioms are a subset of the axioms of $T_{\mathbb{R}}$. They state that every $T_{\mathbb{Q}}$-interpretation is an ordered abelian group. $\geq$ is a total order by the first three axioms. Identity 0, addition $+$, additive inverse $-$, and equality $=$ comprise an abelian group by the next four axioms. The eighth axiom asserts that the abelian group is ordered.

The axiom schema (torsion-free) states that only 0 can be added to itself to produce 0. The name "torsion-free" comes from the following mathematical context. In a group, the **order** of an element v is the integer n such that $n\mathsf{v}$ is the identity element 0: $n\mathsf{v} = 0$. If no such n exists, then the element v has infinite order. A group is **torsion-free** if the only element with finite order is the identity 0.

Finally, the axiom schema (divisible) asserts that all elements of the domain D_I of a $T_{\mathbb{Q}}$-interpretation I are divisible. That is, for every positive integer n, every element $\mathsf{v} \in D_I$ is the sum of n of some other element $\mathsf{w} \in D_I$.

Thus, every $T_{\mathbb{Q}}$-interpretation is a divisible torsion-free abelian group. In particular, the rationals and reals with $+$, $-$, $=$, and $\geq$ are divisible torsion-free abelian groups. As $T_{\mathbb{Q}}$-interpretations, the rationals and reals are **elementarily equivalent**: there does not exist a $\Sigma_{\mathbb{Q}}$-formula that distinguishes between a real $T_{\mathbb{Q}}$-interpretation (an interpretation with domain $\mathbb{R}$) and a rational $T_{\mathbb{Q}}$-interpretation (an interpretation with domain $\mathbb{Q}$).

This characteristic makes sense, intuitively: no linear expression with only integer coefficients can capture, say, $\sqrt{2}$ without also being satisfied by some rational values. When junior high students solve linear algebra problems, they apply addition, subtraction, multiplication, and division; but they do not take roots.

In contrast, $T_{\mathbb{R}}$ is a theory of reals: the $\Sigma_{\mathbb{R}}$-formula $x \cdot x = 2$ is only satisfied by $T_{\mathbb{R}}$-interpretations I in which $\alpha_I[x] = -\sqrt{2}$ or $\alpha_I[x] = \sqrt{2}$.

Example 3.13. Strict inequality is simple to express in $T_{\mathbb{Q}}$. Write

$$\forall x, y.\ \exists z.\ x + y > z$$

as the $\Sigma_{\mathbb{Q}}$-formula

$$\forall x, y.\ \exists z.\ \neg(x + y = z)\ \wedge\ x + y \geq z\ .$$

The situation is similar for $T_{\mathbb{R}}$. ■

Example 3.14. Rational coefficients are simple to express in $T_{\mathbb{Q}}$. Write

$$\frac{1}{2}x + \frac{2}{3}y \geq 4$$

as the $\Sigma_{\mathbb{Q}}$-formula $3x + 4y \geq 24$. ■

For convenience, we sometimes write $x \leq y$ for $y \geq x$.

In Chapter 7, we study a procedure for eliminating quantifiers in the theory $T_{\mathbb{Q}}$. On closed formulae, this procedure decides validity. In Chapter 8, we study a decision procedure for the quantifier-free fragment of $T_{\mathbb{Q}}$, which is efficiently decidable.

3.5 Recursive Data Structures

The **theory of recursive data structures** (**RDS**) describes a set of data structures that are ubiquitous in programming. The most basic RDS is a non-recursive structure, like C's `struct`, in which a single variable has multiple fields. Truly recursive RDSs include lists, stacks, and binary trees.

The theory of recursive data structures T_{RDS} formalizes the reasoning about such structures. It builds on the theory of equality T_{E}.

Theory of Lists

We first focus on the theory of LISP-like lists, T_{cons}, which has signature

$$\Sigma_{\mathsf{cons}} :\ \{\mathsf{cons},\ \mathsf{car},\ \mathsf{cdr},\ \mathsf{atom},\ =\}\ ,$$

where

- cons is a binary function, called the constructor: $\mathsf{cons}(a, b)$ represents the list constructed by concatenating a to b;
- car is a unary function, called the left projector: $\mathsf{car}(\mathsf{cons}(a, b)) = a$;
- cdr is a unary function, called the right projector: $\mathsf{cdr}(\mathsf{cons}(a, b)) = b$;
- atom is a unary predicate: $\mathsf{atom}(x)$ is true iff x is a single-element list;
- and $=$ (equality) is a binary predicate.

car and cdr are historical names abbreviating "contents of address register" and "contents of decrement register", respectively. In the intended interpretations, atoms are individual elements, while lists are multiple elements assembled together via cons. For example, $\mathsf{cons}(a, \mathsf{cons}(b, c))$ is a list of three elements, while a for which $\mathsf{atom}(a)$ holds is an atom. car and cdr are functions for accessing parts of lists. For example, $\mathsf{car}(\mathsf{cons}(a, \mathsf{cons}(b, c)))$ returns the head a of the list; $\mathsf{cdr}(\mathsf{cons}(a, \mathsf{cons}(b, c)))$ returns the tail $\mathsf{cons}(b, c)$ of the list; and $\mathsf{cdr}(\mathsf{cdr}(\mathsf{cons}(a, \mathsf{cons}(b, c))))$ returns c.

The axioms of T_{cons} are the following:

1. the axioms of (reflexivity), (symmetry), and (transitivity) of T_{E}
2. instantiations of the (function congruence) axiom schema for cons, car, and cdr:

$$\forall x_1, x_2, y_1, y_2.\ x_1 = x_2\ \land\ y_1 = y_2\ \rightarrow\ \mathsf{cons}(x_1, y_1) = \mathsf{cons}(x_2, y_2)$$

$$\forall x, y.\ x = y\ \rightarrow\ \mathsf{car}(x) = \mathsf{car}(y)$$

$$\forall x, y.\ x = y\ \rightarrow\ \mathsf{cdr}(x) = \mathsf{cdr}(y)$$

3. an instantiation of the (predicate congruence) axiom schema for atom:

$$\forall x, y.\ x = y\ \rightarrow\ (\mathsf{atom}(x) \leftrightarrow \mathsf{atom}(y))$$

4. $\forall x, y.\ \mathsf{car}(\mathsf{cons}(x, y)) = x$ (left projection)
5. $\forall x, y.\ \mathsf{cdr}(\mathsf{cons}(x, y)) = y$ (right projection)
6. $\forall x.\ \neg\mathsf{atom}(x)\ \rightarrow\ \mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(x)) = x$ (construction)
7. $\forall x, y.\ \neg\mathsf{atom}(\mathsf{cons}(x, y))$ (atom)

The first three sets of axioms define = to be a congruence relation for cons, car, cdr, and atom. The axioms (left projection) and (right projection) define the behavior of car and cdr on non-atom lists: car returns the first element of a cons structure, and cdr returns the second element. However, they do not specify the behavior of car and cdr on atoms. The (construction) axiom states that the cons of $\mathsf{car}(x)$ and $\mathsf{cdr}(x)$ is x itself, unless x is an atom. In other words, cons constructs structures, and car and cdr deconstructs them. Finally, the axiom (atom) asserts that a term with root function symbol cons is not an atom; it is a non-atomic list.

The congruence axioms for cons, car, and cdr assert an important property about lists: two lists are equal iff their components are equal. The forward direction — if two lists are equal, then their components are equal — is a consequence of the (function congruence) axioms for car and cdr. The backward direction is a consequence of the (function congruence) axiom for cons. This relationship between two structures and their components is sometimes called **extensionality**. We see it in arrays as well.

General Theory of RDS

T_{cons} is an instance of the general theory of recursive data structures T_{RDS}. Each RDS contributes the following to the signature:

- an n-ary constructor C;

- n projection functions $\pi_1^{\mathsf{C}}, \ldots, \pi_n^{\mathsf{C}}$;
- and one atom predicate $\mathsf{atom}_{\mathsf{C}}$.

Associated with each RDS is an instantiation of the following axiom schema:

1. the axioms of (reflexivity), (symmetry), and (transitivity) of T_{E};
2. instantiations of the (function congruence) axiom schema for constructor C and set of projectors $\pi_1^{\mathsf{C}}, \ldots, \pi_n^{\mathsf{C}}$;
3. an instantiation of the (predicate congruence) axiom schema for $\mathsf{atom}_{\mathsf{C}}$;
4. for each $i \in \{1, \ldots, n\}$,

$$\forall x_1, \ldots, x_n.\ \pi_i^{\mathsf{C}}(\mathsf{C}(x_1, \ldots, x_n)) = x_i \qquad \text{(projection)}$$

5. $\forall x.\ \neg\mathsf{atom}_{\mathsf{C}}(x) \ \rightarrow\ \mathsf{C}(\pi_1^{\mathsf{C}}(x), \ldots, \pi_n^{\mathsf{C}}(x)) = x$ (construction)
6. $\forall x_1, \ldots, x_n.\ \neg\mathsf{atom}_{\mathsf{C}}(\mathsf{C}(x_1, \ldots, x_n))$ (atom)

The axioms of T_{cons} are an instantiation of this schema. We subsequently focus on T_{cons} for concreteness.

Theory of Acyclic Lists

A variation on this theory in which data structures are acyclic has been studied. Acyclicity makes sense for stacks, but not necessarily for lists and other data structures. Consider the theory of acyclic LISP-like lists, T^+_{cons}. Its axioms include those of T_{cons} and the following axiom schema:

$$\begin{array}{l}
\forall x.\ \mathsf{car}(x) \neq x \\
\forall x.\ \mathsf{cdr}(x) \neq x \\
\forall x.\ \mathsf{car}(\mathsf{car}(x)) \neq x \\
\forall x.\ \mathsf{car}(\mathsf{cdr}(x)) \neq x \\
\forall x.\ \mathsf{cdr}(\mathsf{car}(x)) \neq x \\
\ldots
\end{array}$$

T^+_{cons} is decidable, but T_{cons} is not. However, the quantifier-free fragments of these theories are efficiently decidable.

Theory of Lists with Specified Atoms

The axioms of T_{cons} leave the behavior of car and cdr on atoms unspecified. Adding the axiom

$$\forall x.\ \mathsf{atom}(x) \ \rightarrow\ \mathsf{atom}(\mathsf{car}(x)) \ \wedge\ \mathsf{atom}(\mathsf{cdr}(x))$$

to those of T_{cons} makes decidability of the resulting theory $T^{\mathsf{atom}}_{\mathsf{cons}}$ NP-complete.

Theory of Lists with Equality

In Chapter 9, we describe a decision procedure for satisfiability in the quantifier-free fragment of T_{cons}. The decision procedure is actually applicable to the quantifier-free fragment of a more expressive theory, $T^{=}_{\mathsf{cons}}$, which is the combination of T_{E} and T_{cons} and thus includes uninterpreted constants, functions, and predicates. Thus, its signature is $\Sigma_{\mathsf{E}} \cup \Sigma_{\mathsf{cons}}$, and its axioms are the union of the axioms of T_{E} and T_{cons}. In Section 3.8 and Chapter 10, we discuss more general combinations of theories.

Example 3.15. To prove that the $\Sigma^{=}_{\mathsf{cons}}$-formula

$$F: \begin{array}{l} \mathsf{car}(a) = \mathsf{car}(b)\ \wedge\ \mathsf{cdr}(a) = \mathsf{cdr}(b)\ \wedge\ \neg\mathsf{atom}(a)\ \wedge\ \neg\mathsf{atom}(b) \\ \quad \rightarrow\ f(a) = f(b) \end{array}$$

is $T^{=}_{\mathsf{cons}}$-valid, assume otherwise: there exists a $T^{=}_{\mathsf{cons}}$-interpretation I such that $I \not\models F$:

1.	$I \not\models F$	assumption
2.	$I \models \mathsf{car}(a) = \mathsf{car}(b)$	1, →, ∧
3.	$I \models \mathsf{cdr}(a) = \mathsf{cdr}(b)$	1, →, ∧
4.	$I \models \neg\mathsf{atom}(a)$	1, →, ∧
5.	$I \models \neg\mathsf{atom}(b)$	1, →, ∧
6.	$I \not\models f(a) = f(b)$	1, →
7.	$I \models \mathsf{cons}(\mathsf{car}(a), \mathsf{cdr}(a)) = \mathsf{cons}(\mathsf{car}(b), \mathsf{cdr}(b))$	
		2, 3, (function congruence)
8.	$I \models \mathsf{cons}(\mathsf{car}(a), \mathsf{cdr}(a)) = a$	4, (construction)
9.	$I \models \mathsf{cons}(\mathsf{car}(b), \mathsf{cdr}(b)) = b$	5, (construction)
10.	$I \models a = b$	7, 8, 9, (transitivity)
11.	$I \models f(a) = f(b)$	10, (function congruence)
12.	$I \models \bot$	6, 11

Therefore, F is $T^{=}_{\mathsf{cons}}$-valid. ■

3.6 Arrays

Arrays are another common data structure in programming. They are similar to the uninterpreted functions of T_{E} except that they can be modified. The **theory of arrays** T_{A} describes the basic characteristic of an array: if value v is written to position i of array a, then subsequently reading from position i of a should return v. Because logic is static, modified arrays are represented functionally, as in functional programming.

The theory of arrays T_{A} has signature

$$\Sigma_{\mathsf{A}} :\ \{\cdot[\cdot],\ \cdot\langle\cdot \triangleleft \cdot\rangle,\ =\}\ ,$$

where

- $a[i]$ (read) is a binary function: $a[i]$ represents the value of array a at position i;
- $a\langle i \triangleleft v\rangle$ (write) is a ternary function: $a\langle i \triangleleft v\rangle$ represents the modified array a in which position i has value v;
- and $=$ (equality) is a binary predicate.

$\cdot[\cdot]$ and $\cdot\langle\cdot\triangleleft\cdot\rangle$ really are binary and ternary functions, respectively, even though we write them using a convenient notation. Writing $a[i]$ as $\mathsf{read}(a, i)$ and $a\langle i\triangleleft e\rangle$ as $\mathsf{write}(a, i, e)$ emphasizes that they are functions.

Arrays are represented functionally. The term $a\langle i \triangleleft v\rangle$ is an array that is like a except that it has value v at position i. The term $a\langle i \triangleleft v\rangle[j]$ (which abbreviates $(a\langle i\triangleleft v\rangle)[j]$) is equal to the value of array $a\langle i\triangleleft v\rangle$ at position j: it is v if $j = i$ and $a[j]$ otherwise. $a\langle i\triangleleft v\rangle\langle j\triangleleft w\rangle$ (which abbreviates $(a\langle i\triangleleft v\rangle)\langle j\triangleleft w\rangle$) is an array that is like a except that it differs at the positions i, where it has value v, and j, where is has value w. Finally, the term $a\langle i\triangleleft v\rangle\langle j\triangleleft w\rangle[k]$ (which abbreviates $((a\langle i \triangleleft v\rangle)\langle j \triangleleft w\rangle)[k]$) has value w if $k = j$ (even if $k = i$ also), value v if $k = i$ and $k \neq j$, and value $a[k]$ otherwise.

The axioms of T_{A} are the following:

1. the axioms of (reflexivity), (symmetry), and (transitivity) of T_{E};
2. $\forall a, i, j.\ i = j \ \rightarrow\ a[i] = a[j]$ (array congruence)
3. $\forall a, v, i, j.\ i = j \ \rightarrow\ a\langle i \triangleleft v\rangle[j] = v$ (read-over-write 1)
4. $\forall a, v, i, j.\ i \neq j \ \rightarrow\ a\langle i \triangleleft v\rangle[j] = a[j]$ (read-over-write 2)

The first set of axioms defines $=$ as an equivalence relation. The next axiom asserts that accessing an array with two equal expressions produces the same element. The final two axioms capture the basic characteristic of arrays: reading at an index that has been written produces the most recently written value.

The equality predicate $=$ is only defined for array "elements". For example,

$$F:\ a[i] = e \ \rightarrow\ a\langle i \triangleleft e\rangle = a$$

is not T_{A}-valid, although our intuition suggests that it should be. The problem is that the interaction between $=$ and the read and write functions is not captured in the axioms of T_{A}. In other words, equality between arrays, not just between elements, is undefined.

Instead of F, we write

$$F':\ a[i] = e \ \rightarrow\ \forall j.\ a\langle i \triangleleft e\rangle[j] = a[j]\ ,$$

which is T_{A}-valid.

Example 3.16. To prove that

$$F':\ a[i] = e \ \rightarrow\ \forall j.\ a\langle i \triangleleft e\rangle[j] = a[j]\ ,$$

is T_{A}-valid, assume otherwise: there is a T_{A}-interpretation I such that $I \not\models F'$:

1.	$I \not\models F'$		assumption
2.	$I \models a[i] = e$		1, $\rightarrow$
3.	$I \not\models \forall j.\ a\langle i \triangleleft e\rangle[j] = a[j]$		1, $\rightarrow$
4.	$I_1 :\ I \triangleleft \{j \mapsto \mathsf{j}\} \not\models a\langle i \triangleleft e\rangle[j] = a[j]$		3, $\forall$, for some $\mathsf{j} \in D_I$
5.	$I_1 \models a\langle i \triangleleft e\rangle[j] \neq a[j]$		4, $\neg$
6.	$I_1 \models i = j$		5, (read-over-write 2)
7.	$I_1 \models a[i] = a[j]$		6, (array congruence)
8.	$I_1 \models a\langle i \triangleleft e\rangle[j] = e$		6, (read-over-write 1)
9.	$I_1 \models a\langle i \triangleleft e\rangle[j] = a[j]$		2, 7, 8, (transitivity)
10.	$I_1 \models \bot$		4, 9

We derive line 6 from line 5 by using the **contrapositive** of (read-over-write 2). The contrapositive of $F_1 \rightarrow F_2$ is $\neg F_2 \rightarrow \neg F_1$, and

$$F_1 \rightarrow F_2 \ \Leftrightarrow\ \neg F_2 \rightarrow \neg F_1\ .$$

Lines 4 and 9 are contradictory, so that actually $I \models F'$. Thus, F' is T_{A}-valid. ■

Unfortunately, T_{A}-validity is undecidable. It is straightforward to encode arbitrary formulae of FOL in T_{A} by viewing functions as multi-dimensional arrays (arrays whose elements are arrays). Therefore, a theory $T_{\mathsf{A}}^{=}$ in which the behavior of $=$ on arrays is axiomatized has been studied. Its quantifier-free fragment is decidable. The signature of $T_{\mathsf{A}}^{=}$ is the same as that of T_{A}. Its axioms consists of those of T_{A} and the following axiom:

$$\forall a, b.\ (\forall i.\ a[i] = b[i]) \ \leftrightarrow\ a = b \qquad \text{(extensionality)}$$

Example 3.17. To prove that

$$F:\ a[i] = e \ \rightarrow\ a\langle i \triangleleft e\rangle = a$$

is $T_{\mathsf{A}}^{=}$-valid, assume otherwise: there is a $T_{\mathsf{A}}^{=}$-interpretation I such that $I \not\models F$:

1.	$I \not\models F$	assumption
2.	$I \models a[i] = e$	1, $\rightarrow$
3.	$I \not\models a\langle i \triangleleft e\rangle = a$	1, $\rightarrow$
4.	$I \models a\langle i \triangleleft e\rangle \neq a$	3, $\neg$
5.	$I \models \neg(\forall j.\ a\langle i \triangleleft e\rangle[j] = a[j])$	4, (extensionality)
6.	$I \not\models \forall j.\ a\langle i \triangleleft e\rangle[j] = a[j]$	5, $\neg$

The rest of the proof then proceeds as in Example 3.16. ■

We present a decision procedure for the quantifier-free fragment of T_{A} in Chapter 9. In Chapter 11, we present a decision procedure for satisfiability in a fragment of T_{A} that is more expressive than even the quantifier-free fragment of $T_{\mathsf{A}}^{=}$.

Table 3.1. Decidability of theories and quantifier-free fragments

Theory	Description	Full	QFF
T_{E}	equality	no	yes
T_{PA}	Peano arithmetic	no	no
$T_{\mathbb{N}}$	Presburger arithmetic	yes	yes
$T_{\mathbb{Z}}$	linear integers	yes	yes
$T_{\mathbb{R}}$	reals (with $\cdot$)	yes	yes
$T_{\mathbb{Q}}$	rationals (without $\cdot$)	yes	yes
T_{RDS}	recursive data structures	no	yes
T_{RDS}^{+}	acyclic recursive data structures	yes	yes
T_{A}	arrays	no	yes
$T_{\mathsf{A}}^{=}$	arrays with extensionality	no	yes

Table 3.2. Complexities for decidable theories

Theory	Complexity
PL	NP-complete
$T_{\mathbb{N}}$, $T_{\mathbb{Z}}$	$\Omega\left(2^{2^{n}}\right)$, $O\left(2^{2^{2^{kn}}}\right)$
$T_{\mathbb{R}}$	$O\left(2^{2^{kn}}\right)$
$T_{\mathbb{Q}}$	$\Omega\left(2^{n}\right)$, $O\left(2^{2^{kn}}\right)$
T_{RDS}^{+}	not elementary recursive

3.7 ★Survey of Decidability and Complexity

We survey the known decidability and complexity results of the theories of this chapter.

Table 3.1 summarizes the decidability results for the first-order theories. The quantifier-free fragment of each theory that we study in Part II of this book is decidable.

Table 3.2 summarizes the complexity results for satisfiability in PL and the decidable first-order theories. For all complexities, n is the size of the input formula, and k is some positive integer. A decision problem is not **elementary recursive** if its running time cannot be bounded by a fixed-height stack of exponentials. Only decision procedures for satisfiability in PL scale well to large problems.

Table 3.3 summarizes the complexity results for the quantifier-free fragments. As satisfiability in PL is already NP-complete, we consider only **conjunctive** formulae, which are just conjunctions of literals. For example, satisfiability of propositional conjunctive formulae is decidable in linear time: if both P and $\neg P$ appear in F, for some propositional variable P, then F is unsatisfiable; otherwise, F is satisfiable. For quantifier-free (but not conjunctive) formulae, all complexities except that for $T_{\mathbb{R}}$ are NP-complete. Satisfiability in the quantifier-free fragments of T_{E}, $T_{\mathbb{Q}}$, T_{RDS}, and T_{RDS}^{+} is efficiently decidable.

Table 3.3. Complexities for quantifier-free, conjunctive fragments of theories

Theory	Complexity	Theory	Complexity
PL	$\Theta(n)$	T_{E}	$O(n \log n)$
$T_{\mathbb{N}}$, $T_{\mathbb{Z}}$	NP-complete	$T_{\mathbb{R}}$	$O\left(2^{2^{kn}}\right)$
$T_{\mathbb{Q}}$	PTIME	T^{+}_{RDS}	$\Theta(n)$
T_{RDS}	$O(n \log n)$	T_{A}	NP-complete

3.8 Combination Theories

In practice, the formulae that we want to check for satisfiability or validity span multiple theories. For example, in program verification, one might want to prove a property about an array of integers or a list of reals. We will see many such examples in Chapter 5. Thus, decision procedures for fragments of first-order theories are essentially useless unless they can be combined.

What does every signature of every theory presented so far have in common? They all have equality, =. Nelson and Oppen made equality the focal predicate in their general method for combining quantifier-free fragments of first-order theories (with some restrictions). Given two theories T_1 and T_2 such that $\Sigma_1 \cap \Sigma_2 = \{=\}$ — only = is shared — the combined theory $T_1 \cup T_2$ has signature $\Sigma_1 \cup \Sigma_2$ and axioms $A_1 \cup A_2$. Nelson and Oppen showed that if

- satisfiability in the quantifier-free fragment of T_1 is decidable;
- satisfiability in the quantifier-free fragment of T_2 is decidable;
- and certain technical requirements are met,

then satisfiability in the quantifier-free fragment of $T_1 \cup T_2$ is decidable. Furthermore, if the decision procedures for T_1 and T_2 are in P (in NP), then the combined decision procedure for $T_1 \cup T_2$ is in P (in NP).

Chapter 10 studies the Nelson-Oppen combination of decision procedures.

Example 3.18. To prove that the $(\Sigma_{\mathsf{A}}^{=} \cup \Sigma_{\mathbb{Z}})$-formula

$$F : \; a = b \;\rightarrow\; a[i] \geq b[i]$$

is $(T_{\mathsf{A}}^{=} \cup T_{\mathbb{Z}})$-valid we assume otherwise: there is a $(T_{\mathsf{A}}^{=} \cup T_{\mathbb{Z}})$-interpretation I such that $I \not\models F$:

1.	$I \not\models F$	assumption
2.	$I \models a = b$	1, $\rightarrow$
3.	$I \not\models a[i] \geq b[i]$	1, $\rightarrow$
4.	$I \models \neg(a[i] \geq b[i])$	3, $\neg$
5.	$I \models a[i] = b[i]$	2, $T_{\mathsf{A}}^{=}$ (extensionality)
6.	$I \models \bot$	4, 5, $T_{\mathsf{A}}^{=} \cup T_{\mathbb{Z}}$

Line 6 summarizes the argument that it is impossible for a $T_{\mathbb{Z}}$-interpretation to satisfy both $a[i] = b[i]$ and $\neg(a[i] \geq b[i])$. ■

Example 3.19. The $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$1 \leq x \ \land \ x \leq 2 \ \land \ f(x) \neq f(1) \ \land \ f(x) \neq f(2)$$

is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-unsatisfiable, for x cannot be either 1 or 2 without violating (function congruence). Seen as a $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Q}})$-formula, it is $(T_{\mathsf{E}} \cup T_{\mathbb{Q}})$-satisfiable: choose $x = \frac{3}{2}$.

The $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Q}})$-formula

$$f(f(x) - f(y)) \neq f(z) \ \land \ x \leq y \ \land \ y + z \leq x \ \land \ 0 \leq z$$

is $(T_{\mathsf{E}} \cup T_{\mathbb{Q}})$-unsatisfiable. In particular, the final three literals imply that $z = 0$ and $x = y$, so that $f(x) = f(y)$. But then from the first literal, $f(0) \neq f(0)$ since both $f(x) - f(y)$ and z equal 0.

Finally, the $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$1 \leq x \ \land \ x \leq 3 \ \land \ f(x) \neq f(1) \ \land \ f(x) \neq f(3) \ \land \ f(1) \neq f(2)$$

is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-satisfiable since x can be 2 without violating (function congruence). ■

3.9 Summary

Important data types in software and hardware models include integers; rationals; recursive data structures like records, lists, stacks, and trees; and arrays. This chapter introduces first-order theories that formalize these data types. It covers:

- *First-order theories.* Formalizations of structures and operations into first-order logic: signatures, axioms. Fragments of theories, in particular quantifier-free fragments. Interpretations, satisfiability, validity.
- Specific theories:
  - *Equality* defines the binary predicate $=$ as a congruence relation. Satisfiability in the quantifier-free fragment is efficiently decidable, and the decision procedure is the basis for decision procedures for data structures (see Chapter 9).
  - *Integer arithmetic.* Satisfiability in integer arithmetic without multiplication is decidable.
  - *Rational* and *real arithmetic.* Satisfiability in real arithmetic with multiplication is decidable with high complexity. Satisfiability in rational arithmetic without multiplication is efficiently decidable. Rational arithmetic without multiplication is indistinguishable from real arithmetic without multiplication.
  - *Recursive data structures* include records, lists, stacks, and queues. Satisfiability in the quantifier-free fragment is efficiently decidable.

  - *Arrays* can be read and written. Satisfiability in the quantifier-free fragment is decidable. Chapter 11 studies a larger fragment in which satisfiability is still decidable.
- *Combination theories.* How can decision procedures for multiple theories be combined to decide satisfiability in combination theories?

Studying first-order theories is important for two reasons. First, theories formalize into FOL interesting structures and operations on the structures. Second, satisfiability in some theories or fragments of theories is decidable and thus can be reasoned about algorithmically, whereas satisfiability in general FOL is undecidable. Part II of this book focuses on such theories and fragments that are useful for program analysis. Chapters 5 and 6 provide many examples of formulae from combinations of these theories in the context of program verification.

Bibliographic Remarks

The undecidability of validity in FOL [13] motivated the subsequent study of first-order theories and fragments. In 1929, Presburger proved that satisfiability in arithmetic without multiplication is decidable [73]. Tarski showed in the 1930s that real arithmetic is decidable even with multiplication, although the Second World War delayed the publication of this result [90]. The axiomatization of recursive data structures that we study is from work by Nelson and Oppen [66]. Oppen studied a variation in which structures are acyclic [69]. The axiomatization of arrays, in particular the read-over-write axioms, is due to McCarthy [59]. The Nelson-Oppen combination method is based on work by Nelson and Oppen in the late 1970s and early 1980s [65].

Exercises

3.1 (Semantic argument in T_{E}). Use the semantic method to argue the validity of the following Σ_{E}-formulae, or identify a counterexample (a falsifying T_{E}-interpretation).

(*a*) $f(x,y) = f(y,x) \;\rightarrow\; f(a,y) = f(y,a)$
(*b*) $f(g(x)) = g(f(x)) \;\wedge\; f(g(f(y))) = x \;\wedge\; f(y) = x \;\rightarrow\; g(f(x)) = x$
(*c*) $f(f(f(a))) = f(f(a)) \;\wedge\; f(f(f(f(a)))) = a \;\rightarrow\; f(a) = a$
(*d*) $f(f(f(a))) = f(a) \;\wedge\; f(f(a)) = a \;\rightarrow\; f(a) = a$
(*e*) $p(x) \;\wedge\; f(f(x)) = x \;\wedge\; f(f(f(x))) = x \;\rightarrow\; p(f(x))$

3.2 (Semantic argument in $T_{\mathbb{Z}}$). Use the semantic method to argue the validity of the following $\Sigma_{\mathbb{Z}}$-formulae, or identify a counterexample (a falsifying $T_{\mathbb{Z}}$-interpretation).

(*a*) $x \leq y \;\wedge\; z = x + 1 \;\rightarrow\; z \leq y$

(*b*) $x \leq y \;\wedge\; z = x - 1 \;\rightarrow\; z \leq y$
(*c*) $3x = 2 \;\rightarrow\; x \leq 0$
(*d*) $1 \leq x \;\wedge\; x \leq 2 \;\rightarrow\; x = 1 \;\vee\; x = 2$
(*e*) $1 \leq x \;\wedge\; x + y \leq 3 \;\wedge\; 1 \leq y \;\rightarrow\; x = 1 \;\vee\; x = 2$
(*f*) $0 \leq x \;\wedge\; 0 \leq x + y \;\wedge\; x + y \leq 1 \;\wedge\; (y \leq -2 \;\vee\; 2 \leq y) \;\rightarrow\; 0 \leq -1$

3.3 (Semantic argument in $T_{\mathbb{Q}}$). Use the semantic method to argue the validity of the following $\Sigma_{\mathbb{Q}}$-formulae, or identify a counterexample (a falsifying $T_{\mathbb{Q}}$-interpretation).

(*a*) $3x = 2 \;\rightarrow\; x \leq 0$
(*b*) $0 \leq x + 2y \;\wedge\; 2x + y \leq 1$
(*c*) $1 \leq x \;\wedge\; x \leq 2 \;\rightarrow\; x = 1 \;\vee\; x = 2$

3.4 (Semantic argument in T_{cons}). Use the semantic method to argue the validity of the following Σ_{cons}-formulae, or identify a counterexample (a falsifying T_{cons}-interpretation).

(*a*) $\mathsf{car}(x) = y \;\wedge\; \mathsf{cdr}(x) = z \;\rightarrow\; x = \mathsf{cons}(y, z)$
(*b*) $\neg\mathsf{atom}(x) \;\wedge\; \mathsf{car}(x) = y \;\wedge\; \mathsf{cdr}(x) = z \;\rightarrow\; x = \mathsf{cons}(y, z)$

3.5 (Semantic argument in T_{A}). Use the semantic method to argue the validity of the following Σ_{A}-formulae, or identify a counterexample (a falsifying T_{A}-interpretation).

(*a*) $a\langle i \triangleleft e\rangle[j] = e \;\rightarrow\; i = j$
(*b*) $a\langle i \triangleleft e\rangle[j] = e \;\rightarrow\; a[j] = e$
(*c*) $a\langle i \triangleleft e\rangle[j] = e \;\rightarrow\; i = j \;\vee\; a[j] = e$
(*d*) $a\langle i \triangleleft e\rangle\langle j \triangleleft f\rangle[k] = g \;\wedge\; j \neq k \;\wedge\; i = j \;\rightarrow\; a[k] = g$

3.6 (Semantic argument in combinations). For each of the following formulae, identify the combination of theories in which it lies. To avoid ambiguity, prefer $T_{\mathbb{Z}}$ to $T_{\mathbb{Q}}$. Then argue its validity in that combination of theories using the semantic method, or identify a counterexample.

(*a*) $1 \leq x \;\wedge\; x \leq 2 \;\wedge\; \mathsf{cons}(1, y) \neq \mathsf{cons}(x, y) \;\rightarrow\; \mathsf{cons}(2, y) = \mathsf{cons}(x, y)$
(*b*) $a[i] \geq 1 \;\wedge\; a[i] + x \leq 2 \;\wedge\; x > 0 \;\wedge\; x = i \;\rightarrow\; a\langle x \triangleleft 2\rangle[i] = 1$
(*c*) $1 \leq x \;\wedge\; x \leq 2 \;\wedge\; \mathsf{cons}(1, y) \neq \mathsf{cons}(x, y) \;\rightarrow\; \mathsf{cons}(2, y) = \mathsf{cons}(x, y)$
(*d*) $x + y = z \;\wedge\; f(z) = z \;\rightarrow\; f(x + y) = z$
(*e*) $g(x + y, z) = f(g(x, y)) \;\wedge\; x + z = y \;\wedge\; z \geq 0 \;\wedge\; x \geq y \;\wedge\; g(x, x) = z$
$\quad\rightarrow\; f(z) = g(2x, 0)$

3.7 (Semantic argument in combinations). Redo Exercise 3.6, preferring $T_{\mathbb{Q}}$ to $T_{\mathbb{Z}}$.

4

Induction

Even though this proposition may have an infinite number of cases, I shall give a very short proof of it assuming two lemmas. The first, which is self evident, is that the proposition is valid for the second row. The second is that if the proposition is valid for any row then it must necessarily be valid for the following row.

— Blaise Pascal
Traité du Triangle Arithmetique, c. 1654

This chapter discusses **induction**, a classic proof technique for proving first-order theorems with universal quantifiers. Section 4.1 begins with **stepwise induction**, which may be familiar to the reader from earlier education. Section 4.2 then introduces **complete induction** in the context of arithmetic. Complete induction is theoretically equivalent in power to stepwise induction but sometimes produces more concise proofs. Section 4.3 generalizes complete induction to **well-founded induction** in the context of arithmetic and recursive data structures. Finally, Section 4.4 covers a form of well-founded induction over logical formulae called **structural induction**. It is useful for reasoning about correctness of decision procedures and properties of logical theories and their interpretations.

We apply induction in various ways throughout the book. Structural induction is applied in proofs. Additionally, induction is the basis for the program verification methods of Chapter 5.

4.1 Stepwise Induction

We review stepwise induction for arithmetic and then show that it extends naturally to other theories, such as the theory of lists T_{cons}.

Arithmetic

Recall from Chapter 3 that the theory of Peano arithmetic T_{PA} formalizes arithmetic over the natural numbers. Its axioms include an instance of the (induction) axiom schema

$$F[0] \ \wedge \ (\forall n.\ F[n] \rightarrow F[n+1]) \ \rightarrow \ \forall x.\ F[x]$$

for each Σ_{PA}-formula $F[x]$ with only one free variable x. This axiom schema says that to prove $\forall x.\ F[x]$ — that is, $F[x]$ is T_{PA}-valid for all natural numbers x — it is sufficient to do the following:

- For the **base case**, prove that $F[0]$ is T_{PA}-valid.
- For the **inductive step**, assume as the **inductive hypothesis** that for some arbitrary natural number n, $F[n]$ is T_{PA}-valid. Then prove that $F[n+1]$ is T_{PA}-valid under this assumption.

These two steps comprise the **stepwise induction principle** for Peano (and Presburger) arithmetic.

Example 4.1. Consider the theory T_{PA}^{+} obtained from augmenting T_{PA} with the following axioms:

- $\forall x.\ x^0 = 1$ (exp. zero)
- $\forall x, y.\ x^{y+1} = x^y \cdot x$ (exp. successor)
- $\forall x, z.\ exp_3(x, 0, z) = z$ (exp_3 zero)
- $\forall x, y, z.\ exp_3(x, y+1, z) = exp_3(x, y, x \cdot z)$ (exp_3 successor)

The first two axioms define exponentiation x^y, while the latter two axioms define a ternary function $exp_3(x, y, z)$.

Let us prove that the following formula is T_{PA}^{+}-valid:

$$\forall x, y.\ exp_3(x, y, 1) = x^y \ . \tag{4.1}$$

We need to choose either x or y as the induction variable. Considering the exp_3 axioms, it appears that y is the smarter choice: (exp_3 successor) defines exp_3 recursively by considering the predecessor of $y+1$.

Therefore, we prove by stepwise induction on y that

$$F[y] :\ \forall x.\ exp_3(x, y, 1) = x^y \ .$$

For the **base case**, we prove

$$F[0] :\ \forall x.\ exp_3(x, 0, 1) = x^0 \ .$$

But $x^0 = 1$ by (exp. zero), and $exp_3(x, 0, 1) = 1$ by (exp_3 zero).

Assume as the inductive hypothesis that for arbitrary natural number n,

$$F[n] :\ \forall x.\ exp_3(x, n, 1) = x^n \ . \tag{4.2}$$

We want to prove that

$$F[n+1]:\ \forall x.\ exp_3(x, n+1, 1) = x^{n+1}\ . \tag{4.3}$$

By $(exp_3$ successor), we have

$$exp_3(x, n+1, 1) = exp_3(x, n, x \cdot 1)\ .$$

Unfortunately, the inductive hypothesis (4.2) does not apply to the left side of the equation since $n \neq n+1$, and it does not apply to the right side of the equation because the third argument is $x \cdot 1$ rather than 1. Continuing to apply axioms is unlikely to bring us closer to the proof. Thus, we have failed to prove the property.

What went wrong in the proof? Did we choose the wrong induction variable? Would x have worked better? In fact, it is often the case that the property must be **strengthened** to allow the induction to go through. A stronger theorem provides a stronger inductive hypothesis.

Let us strengthen the property to be proved to

$$\forall x, y, z.\ exp_3(x, y, z) = x^y \cdot z\ . \tag{4.4}$$

It clearly implies the desired property (4.1): just choose $z = 1$.

Again, we must choose the induction variable. Based on $(exp_3$ successor), we use y again. Thus, we prove by stepwise induction on y that

$$F[y]:\ \forall x, z.\ exp_3(x, y, z) = x^y \cdot z\ .$$

For the base case, we prove

$$F[0]:\ \forall x, z.\ exp_3(x, 0, z) = x^0 \cdot z\ .$$

From $(exp_3$ zero), we have $exp_3(x, 0, z) = z$, while from (exp. zero), we have $x^0 \cdot z = 1 \cdot z = z$.

Assume as the inductive hypothesis that

$$F[n]:\ \forall x, z.\ exp_3(x, n, z) = x^n \cdot z \tag{4.5}$$

for arbitrary natural number n. We want to prove that

$$F[n+1]:\ \forall x, z'.\ exp_3(x, n+1, z') = x^{n+1} \cdot z'\ , \tag{4.6}$$

where we have renamed z to z' for convenience. We have

$$\begin{aligned} exp_3(x, n+1, z') &= exp_3(x, n, x \cdot z') && (exp_3 \text{ successor}) \\ &= x^n \cdot (x \cdot z') && \text{IH (4.5), } z \mapsto x \cdot z' \\ &= x^{n+1} \cdot z' && (\text{exp. successor}) \end{aligned}$$

finishing the proof. The annotation $z \mapsto x \cdot z'$ indicates that $x \cdot z'$ is substituted for z when applying the inductive hypothesis (4.5). This substitution is justified because z is universally quantified. Renaming z to z' avoids confusion during the application of the inductive hypothesis in the second line. ■

Lists

We can define stepwise induction over recursive data structures such as lists (see Chapters 3 and 9). Consider the theory of lists T_{cons}. **Stepwise induction** in T_{cons} is defined according to the following schema

$$(\forall\ \mathsf{atom}\ u.\ F[u])\ \wedge\ (\forall u, v.\ F[v] \rightarrow F[\mathsf{cons}(u, v)])\ \rightarrow\ \forall x.\ F[x]$$

for Σ_{cons}-formulae $F[x]$ with only one free variable x. The notation $\forall\ \mathsf{atom}\ u.\ F[u]$ abbreviates $\forall u.\ \mathsf{atom}(u) \rightarrow F[u]$. In other words, to prove $\forall x.\ F[x]$ — that is, $F[x]$ is T_{cons}-valid for all lists x — it is sufficient to do the following:

- For the **base case**, prove that $F[u]$ is T_{cons}-valid for an arbitrary $\mathsf{atom}\ u$.
- For the **inductive step**, assume as the **inductive hypothesis** that for some arbitrary list v, $F[v]$ is valid. Then prove that for arbitrary list u, $F[\mathsf{cons}(u, v)]$ is T_{cons}-valid under this assumption.

These steps comprise the **stepwise induction principle** for lists.

Example 4.2. Consider the theory T^{+}_{cons} obtained from augmenting T_{cons} with the following axioms:

- $\forall\ \mathsf{atom}\ u.\ \forall v.\ concat(u, v) = \mathsf{cons}(u, v)$ (concat. atom)
- $\forall u, v, x.\ concat(\mathsf{cons}(u, v), x) = \mathsf{cons}(u, concat(v, x))$ (concat. list)
- $\forall\ \mathsf{atom}\ u.\ rvs(u) = u$ (reverse atom)
- $\forall x, y.\ rvs(concat(x, y)) = concat(rvs(y), rvs(x))$ (reverse list)
- $\forall\ \mathsf{atom}\ u.\ \mathit{flat}(u)$ (flat atom)
- $\forall u, v.\ \mathit{flat}(\mathsf{cons}(u, v))\ \leftrightarrow\ \mathsf{atom}(u)\ \wedge\ \mathit{flat}(v)$ (flat list)

The first two axioms define the *concat* function, which concatenates two lists together. For example,

$$\begin{aligned}&concat(\mathsf{cons}(a, b), \mathsf{cons}(b, \mathsf{cons}(c, d)))\\ &\qquad = \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{cons}(b, \mathsf{cons}(c, \mathsf{cons}(d)))))\ .\end{aligned}$$

The next two axioms define the *rvs* function, which reverses a list. For example,

$$rvs(\mathsf{cons}(a, \mathsf{cons}(b, c))) = \mathsf{cons}(c, \mathsf{cons}(b, a))\ .$$

Note, however, that *rvs* is undefined on lists like $\mathsf{cons}(\mathsf{cons}(a, b), c)$, for $\mathsf{cons}(\mathsf{cons}(a, b), c)$ cannot result from concatenating two lists together. Therefore, the final two axioms define the *flat* predicate, which evaluates to $\top$ on a list iff every element is an atom. For example, $\mathsf{cons}(a, \mathsf{cons}(b, c))$ is *flat*, but $\mathsf{cons}(\mathsf{cons}(a, b), c)$ is not because the first element of the list is itself a list.

Let us prove that the following formula is T^{+}_{cons}-valid:

$$\forall x.\ \mathit{flat}(x)\ \rightarrow\ rvs(rvs(x)) = x\ . \tag{4.7}$$

For example,

$$\begin{aligned} rvs(rvs(\mathsf{cons}(a, \mathsf{cons}(b, c)))) &= rvs(\mathsf{cons}(c, \mathsf{cons}(b, a))) \\ &= \mathsf{cons}(a, \mathsf{cons}(b, c)) \end{aligned}$$

We prove by stepwise induction on x that

$$F[x] : \ \mathit{flat}(x) \ \rightarrow \ rvs(rvs(x)) = x \ .$$

For the base case, we consider arbitrary atom u and prove

$$F[u] : \ \mathit{flat}(u) \ \rightarrow \ rvs(rvs(u)) = u \ .$$

But $rvs(rvs(u)) = u$ follows from two applications of (reverse atom).

Assume as the inductive hypothesis that for arbitrary list v,

$$F[v] : \ \mathit{flat}(v) \ \rightarrow \ rvs(rvs(v)) = v \ . \tag{4.8}$$

We want to prove that for arbitrary list u,

$$F[\mathsf{cons}(u, v)] : \ \mathit{flat}(\mathsf{cons}(u, v)) \ \rightarrow \ rvs(rvs(\mathsf{cons}(u, v))) = \mathsf{cons}(u, v) \ . \tag{4.9}$$

Consider two cases: either $\mathsf{atom}(u)$ or $\neg\mathsf{atom}(u)$.

If $\neg\mathsf{atom}(u)$, then

$$\mathit{flat}(\mathsf{cons}(u, v)) \ \Leftrightarrow \ \mathsf{atom}(u) \ \wedge \ \mathit{flat}(v) \ \Leftrightarrow \ \bot \ ,$$

by (flat list) and assumption. Therefore, (4.9) holds since its antecedent is $\bot$.

If $\mathsf{atom}(u)$, then we have that

$$\mathit{flat}(\mathsf{cons}(u, v)) \ \Leftrightarrow \ \mathsf{atom}(u) \ \wedge \ \mathit{flat}(v) \ \Leftrightarrow \ \mathit{flat}(v)$$

by (flat list). Furthermore,

$$\begin{aligned} & rvs(rvs(\mathsf{cons}(u, v))) \\ & \quad = rvs(rvs(concat(u, v))) && \text{(concat. atom)} \\ & \quad = rvs(concat(rvs(v), rvs(u))) && \text{(reverse list)} \\ & \quad = concat(rvs(rvs(u)), rvs(rvs(v))) && \text{(reverse list)} \\ & \quad = concat(u, rvs(rvs(v))) && \text{(reverse atom)} \\ & \quad = concat(u, v) && \text{IH (4.8), since } \mathit{flat}(v) \\ & \quad = \mathsf{cons}(u, v) && \text{(concat. atom)} \end{aligned}$$

which finishes the proof. ■

4.2 Complete Induction

Complete induction is a form of induction that sometimes yields more concise proofs. For the theory of arithmetic T_{PA} it is defined according to the following schema

$$(\forall n.\ (\forall n'.\ n' < n \rightarrow F[n']) \rightarrow F[n]) \ \rightarrow\ \forall x.\ F[x]$$

for Σ_{PA}-formulae $F[x]$ with only one free variable x. In other words, to prove $\forall x.\ F[x]$ — that is, $F[x]$ is T_{PA}-valid for all natural numbers x — it is sufficient to follow the **complete induction principle**:

- Assume as the **inductive hypothesis** that for arbitrary natural number n and for every natural number n' such that $n' < n$, $F[n']$ is T_{PA}-valid. Then prove that $F[n]$ is T_{PA}-valid.

It appears that we are missing a base case. In practice, a case analysis usually requires at least one base case. In other words, the base case is implicit in the structure of complete induction. For example, for $n = 0$, the inductive hypothesis does not provide any information — there does not exist a natural number $n' < 0$. Hence, $F[0]$ must be shown separately without assistance from the inductive hypothesis.

Example 4.3. Consider another augmented version of Peano arithmetic, T^*_{PA}, that defines integer division. It has the usual axioms of T_{PA} plus the following:

- $\forall x, y.\ x < y \ \rightarrow\ \mathit{quot}(x, y) = 0$ (quotient less)
- $\forall x, y.\ y > 0 \ \rightarrow\ \mathit{quot}(x + y, y) = \mathit{quot}(x, y) + 1$ (quotient successor)
- $\forall x, y.\ x < y \ \rightarrow\ \mathit{rem}(x, y) = x$ (remainder less)
- $\forall x, y.\ y > 0 \ \rightarrow\ \mathit{rem}(x + y, y) = \mathit{rem}(x, y)$ (remainder successor)

These axioms define functions for computing integer quotients $\mathit{quot}(x, y)$ and remainders $\mathit{rem}(x, y)$. For example, $\mathit{quot}(5, 3) = 1$ and $\mathit{rem}(5, 3) = 2$. We prove two properties, which the reader may recall from grade school, about these functions. First, we prove that the remainder is always less than the divisor:

$$\forall x, y.\ y > 0 \ \rightarrow\ \mathit{rem}(x, y) < y\ . \tag{4.10}$$

Then we prove that

$$\forall x, y.\ y > 0 \ \rightarrow\ x = y \cdot \mathit{quot}(x, y) + \mathit{rem}(x, y)\ . \tag{4.11}$$

For property (4.10), (remainder successor) suggests that we apply complete induction on x to prove

$$F[x] :\ \forall y.\ y > 0 \ \rightarrow\ \mathit{rem}(x, y) < y\ . \tag{4.12}$$

Thus, for the inductive hypothesis, assume that for arbitrary natural number x,

$$\forall x'.\ x' < x \ \rightarrow\ \underbrace{\forall y.\ y > 0 \ \rightarrow\ \mathit{rem}(x', y) < y}_{F[x']}\ . \tag{4.13}$$

Let y be an arbitrary positive natural number. Consider two cases: either $x < y$ or $\neg(x < y)$.

If $x < y$, then

$$\begin{aligned} rem(x,y) &= x && \textsf{(remainder less)} \\ &< y && \text{by assumption } x < y \end{aligned}$$

as desired.

If $\neg(x < y)$, then there is a natural number n, $n < x$, such that $x = n + y$. Compute

$$\begin{aligned} rem(x,y) &= rem(n+y,y) && x = n + y \\ &= rem(n,y) && \textsf{(remainder successor)} \\ &< y && \text{IH (4.13), } x' \mapsto n, \text{ since } n < x \end{aligned}$$

finishing the proof of this property.

For property (4.11), (remainder successor) again suggests that we apply complete induction on x to prove

$$G[x] : \ \forall y.\ y > 0 \ \rightarrow \ x = y \cdot quot(x,y) + rem(x,y)\ . \tag{4.14}$$

Thus, for the inductive hypothesis, assume that for arbitrary natural number x,

$$\forall x'.\ x' < x \ \rightarrow \ \underbrace{\forall y.\ y > 0 \ \rightarrow \ x' = y \cdot quot(x',y) + rem(x',y)}_{G[x']}\ . \tag{4.15}$$

Let y be an arbitrary positive natural number. Consider two cases: either $x < y$ or $\neg(x < y)$.

If $x < y$, then

$$\begin{aligned} & y \cdot quot(x,y) + rem(x,y) \\ & \quad = y \cdot 0 + rem(x,y) && \textsf{(quotient less)} \\ & \quad = x && \textsf{(remainder less)} \end{aligned}$$

as desired.

If $\neg(x < y)$, then there is a natural number $n < x$ such that $x = n + y$. Compute

$$\begin{aligned} & y \cdot quot(x,y) + rem(x,y) \\ & \quad = y \cdot quot(n+y,y) + rem(n+y,y) && x = n + y \\ & \quad = y \cdot (quot(n,y) + 1) + rem(n+y,y) && \textsf{(quotient successor)} \\ & \quad = y \cdot (quot(n,y) + 1) + rem(n,y) && \textsf{(remainder successor)} \\ & \quad = (y \cdot quot(n,y) + rem(n,y)) + y \\ & \quad = n + y && \text{IH (4.15), } x' \mapsto n, \text{ since } n < x \\ & \quad = x && x = n + y \end{aligned}$$

finishing the proof of this property. ■

In the next section, we generalize complete induction so that we can apply it in other theories.

4.3 Well-Founded Induction

A binary predicate $\prec$ over a set S is a **well-founded relation** iff there does not exist an infinite sequence $s_1, s_2, s_3, \ldots$ of elements of S such that each successive element is less than its predecessor:

$$s_1 \succ s_2 \succ s_3 \succ \cdots \ ,$$

where $s \prec t$ iff $t \succ s$. In other words, each sequence of elements of S that decreases according to $\prec$ is finite.

Example 4.4. The relation $<$ is well-founded over the natural numbers. Any sequence of natural numbers decreasing according to $<$ is finite:

$$1023 > 39 > 30 > 29 > 8 > 3 > 0 \ .$$

However, the relation $<$ is not well-founded over the rationals. Consider the infinite decreasing sequence

$$1 > \frac{1}{2} > \frac{1}{3} > \frac{1}{4} > \cdots \ ,$$

that is, the sequence $s_i = \frac{1}{i}$ for $i \geq 0$. ■

Example 4.5. Consider the theory $T_{\mathsf{cons}}^{\mathsf{PA}}$, which includes the axioms of T_{cons} and T_{PA} and the following axioms:

- $\forall\,\mathsf{atom}\ u, v.\ u \preceq_{\mathsf{c}} v \ \leftrightarrow\ u = v$ $\quad(\preceq_{\mathsf{c}}\ (1))$
- $\forall\,\mathsf{atom}\ u.\ \forall v.\ \neg\mathsf{atom}(v) \ \rightarrow\ \neg(v \preceq_{\mathsf{c}} u)$ $\quad(\preceq_{\mathsf{c}}\ (2))$
- $\forall\,\mathsf{atom}\ u.\ \forall v, w.\ u \preceq_{\mathsf{c}} \mathsf{cons}(v, w) \ \leftrightarrow\ u = v \ \lor\ u \preceq_{\mathsf{c}} w$ $\quad(\preceq_{\mathsf{c}}\ (3))$
- $\forall u_1, v_1, u_2, v_2.\ \mathsf{cons}(u_1, v_1) \preceq_{\mathsf{c}} \mathsf{cons}(u_2, v_2)$
  $\leftrightarrow\ (u_1 = u_2 \ \land\ v_1 \preceq_{\mathsf{c}} v_2) \ \lor\ \mathsf{cons}(u_1, v_1) \preceq_{\mathsf{c}} v_2$ $\quad(\preceq_{\mathsf{c}}\ (4))$
- $\forall x, y.\ x \prec_{\mathsf{c}} y \ \leftrightarrow\ x \preceq_{\mathsf{c}} y \ \land\ x \neq y$ $\quad(\prec_{\mathsf{c}})$
- $\forall\,\mathsf{atom}\ u.\ |u| = 1$ $\quad$(length atom)
- $\forall u, v.\ |\mathsf{cons}(u, v)| = 1 + |v|$ $\quad$(length list)

The first four axioms define the sublist relation $\preceq_{\mathsf{c}}$. $x \preceq_{\mathsf{c}} y$ holds iff x is a (not necessarily strict) sublist of y. The next axiom defines the strict sublist relation: $x \prec_{\mathsf{c}} y$ iff x is a strict sublist of y. The final two axioms define the length function, which returns the number of elements in a list.

The strict sublist relation $\prec_{\mathsf{c}}$ is well-founded on the set of all lists. One can prove that the number of sublists of a list is finite; and that its set of strict sublists is a superset of the set of strict sublists of any of its sublists. Hence, there cannot be an infinite sequence of lists descending according to $\prec_{\mathsf{c}}$. ■

Well-founded induction generalizes complete induction to arbitrary theory T by allowing the use of any binary predicate $\prec$ that is well-founded in the domain of every T-interpretation. It is defined in the theory T with well-founded relation $\prec$ by the following schema

$$(\forall n.\ (\forall n'.\ n' \prec n \rightarrow F[n']) \rightarrow F[n]) \ \rightarrow\ \forall x.\ F[x]$$

for Σ-formulae $F[x]$ with only one free variable x. In other words, to prove the T-validity of $\forall x.\ F[x]$, it is sufficient to follow the **well-founded induction principle**:

- Assume as the **inductive hypothesis** that for arbitrary element n and for every element n' such that $n' \prec n$, $F[n']$ is T-valid. Then prove that $F[n]$ is T-valid.

Complete induction in T_{PA} of Section 4.2 is a specific instance of well-founded induction that uses the well-founded relation $<$.

A theory of lists augmented with the first five axioms of Example 4.5 has well-founded induction in which the well-founded relation is $\prec_{\mathsf{c}}$.

Example 4.6. Consider proving the trivial property

$$\forall x.\ |x| \geq 1 \tag{4.16}$$

in $T_{\mathsf{cons}}^{\mathsf{PA}}$, which was defined in Example 4.5. We apply well-founded induction on x using the well-founded relation $\prec_{\mathsf{c}}$ to prove

$$F[x] :\ |x| \geq 1\ . \tag{4.17}$$

For the inductive hypothesis, assume that

$$\forall x'.\ x' \prec_{\mathsf{c}} x \ \rightarrow\ \underbrace{|x'| \geq 1}_{F[x']}\ . \tag{4.18}$$

Consider two cases: either $\mathsf{atom}(x)$ or $\neg\mathsf{atom}(x)$.

In the first case $|x| = 1 \geq 1$ by (length atom).

In the second case x is not an atom, so $x = \mathsf{cons}(u, v)$ for some u, v by the (construction) axiom. Then

$$\begin{aligned}
|x| &= |\mathsf{cons}(u, v)| \\
&= 1 + |v| && \text{(length list)} \\
&\geq 1 + 1 && \text{IH (4.18), } x' \mapsto v \text{, since } v \prec_{\mathsf{c}} \mathsf{cons}(u, v) \\
&\geq 1
\end{aligned}$$

as desired. Exercise 4.2 asks the reader to prove formally that $\forall u, v.\ v \prec_{\mathsf{c}} \mathsf{cons}(u, v)$.

This property is also easily proved using stepwise induction. ■

In applying well-founded induction, we need not restrict ourselves to the intended domain D of a theory T. A useful class of well-founded relations are **lexicographic relations**. From a finite set of pairs of sets and well-founded relations $(S_1, \prec_1), \ldots, (S_m, \prec_m)$, construct the set

$$S = S_1 \times \cdots \times S_m \ ,$$

and define the relation $\prec$:

$$(s_1, \ldots, s_m) \prec (t_1, \ldots, t_m) \ \Leftrightarrow \ \bigvee_{i=1}^{m} \left(s_i \prec_i t_i \ \wedge \ \bigwedge_{j=1}^{i-1} s_j = t_j \right)$$

for $s_i, t_i \in S_i$. That is, for elements $s : (s_1, \ldots, s_m)$, $t : (t_1, \ldots, t_m)$ of S, $s \prec t$ iff at some position i, $s_i \prec_i t_i$, and for all preceding positions j, $s_j = t_j$. For convenience, we abbreviate $(s_1, \ldots, s_m)$ by $\overline{s}$ and thus write, for example, $\overline{s} \prec \overline{t}$.

Lexicographic well-founded induction has the form

$$(\forall \overline{n}.\ (\forall \overline{n}'.\ \overline{n}' \prec \overline{n} \rightarrow F[\overline{n}']) \rightarrow F[\overline{n}]) \ \rightarrow \ \forall \overline{x}.\ F[\overline{x}]$$

for Σ-formula $F[\overline{x}]$ with only free variables $\overline{x} = \{x_1, \ldots, x_m\}$. Notice that the form of this induction principle is the same as well-founded induction. The only difference is that we are considering tuples $\overline{n} = (n_1, \ldots, n_m)$ rather than single elements n.

Example 4.7. Consider the following puzzle. You have a bag of red, yellow, and blue chips. If only one chip remains in the bag, you take it out. Otherwise, you remove two chips at random:

1. If one of the two removed chips is red, you do not put any chips in the bag.
2. If both of the removed chips are yellow, you put one yellow chip and five blue chips in the bag.
3. If one of the chips is blue and the other is not red, you put ten red chips in the bag.

These cases cover all possibilities for the two chips. Does this process always halt?

We prove the following property: *for all bags of chips, you can execute the choose-and-replace process only a finite number of times before the bag is empty.* Let the triple

$$(\#\text{yellow},\ \#\text{blue},\ \#\text{red})$$

represent the current state of the bag. Such a tuple is in the set of triples of natural numbers $S : \mathbb{N}^3$. Let $<_3$ be the natural lexicographic extension of $<$ to such triples. For example,

$$(11, 13, 3) \not<_3 (11, 9, 104) \quad \text{but} \quad (11, 9, 104) <_3 (11, 13, 3)\ .$$

We prove that for arbitrary bag state (y, b, r) represented by the triple of natural numbers y, b, and r, only a finite number of steps remain.

For the base cases, consider when the bag has no chips (state $(0, 0, 0)$) or only one chip (one of states $(1, 0, 0)$, $(0, 1, 0)$, or $(0, 0, 1)$). In the first case, you are done; in the second set of cases, only one step remains.

Assume for the inductive hypothesis that for any bag state (y', b', r') such that

$$(y', b', r') <_3 (y, b, r) \ ,$$

only a finite number of steps remain. Now remove two chips from the current bag, represented by state (y, b, r). Consider the three possible cases:

1. *If one of the two removed chips is red, you do not put any chips in the bag.* Then the new bag state is $(y - 1, b, r - 1)$, $(y, b - 1, r - 1)$, or $(y, b, r - 2)$. Each is less than (y, b, r) by $<_3$.
2. *If both of the removed chips are yellow, you put one yellow chip and five blue chips in the bag.* Then the new bag state is $(y - 1, b + 5, r)$, which is less than (y, b, r) by $<_3$.
3. *If one of the chips is blue and the other is not red, you put ten red chips in the bag.* Then the new bag state is $(y - 1, b - 1, r + 10)$ or $(y, b - 2, r + 10)$. Each is less than (y, b, r) by $<_3$.

In all cases, we can apply the inductive hypothesis to deduce that only a finite number of steps remain from the next state. Since only one step of the process is required to get to the next state, there are only a finite number of steps remaining from the current state (y, b, r). Hence, the process always halts. ■

Example 4.8. Consider proving the property

$$\forall x, y.\ x \preceq_{\mathsf{c}} y \ \rightarrow\ |x| \leq |y| \tag{4.19}$$

in $T_{\mathsf{cons}}^{\mathsf{PA}}$. Let $\prec_{\mathsf{c}}^2$ be the natural lexicographic extension of $\prec_{\mathsf{c}}$ to pairs of lists. That is, $(x_1, y_1) \prec_{\mathsf{c}}^2 (x_2, y_2)$ iff $x_1 \prec_{\mathsf{c}} x_2 \lor (x_1 = x_2 \ \land\ y_1 \prec_{\mathsf{c}} y_2)$.

We apply lexicographic well-founded induction to pairs (x, y) to prove

$$F[x, y] :\ x \preceq_{\mathsf{c}} y \ \rightarrow\ |x| \leq |y| \ . \tag{4.20}$$

For the inductive hypothesis, assume that

$$\forall x', y'.\ (x', y') \prec_{\mathsf{c}}^2 (x, y) \ \rightarrow\ \underbrace{x' \preceq_{\mathsf{c}} y' \ \rightarrow\ |x'| \leq |y'|}_{F[x', y']} \ . \tag{4.21}$$

Now consider arbitrary lists x and y. Consider two cases: either $\mathsf{atom}(x)$ or $\neg\mathsf{atom}(x)$.

If $\mathsf{atom}(x)$, then

$\lvert x\rvert = 1$	(length atom)
$\quad \leq \lvert y\rvert$	Example 4.6

Hence, regardless of whether $x \preceq_{\mathsf{c}} y$, we have that $|x| \leq |y|$ so that (4.20) holds.

If $\neg\mathsf{atom}(x)$, then consider two cases: either $\mathsf{atom}(y)$ or $\neg\mathsf{atom}(y)$. If $\mathsf{atom}(y)$, then

$$x \preceq_{\mathsf{c}} y \;\Leftrightarrow\; \perp$$

by ($\preceq_{\mathsf{c}}$ (2)); therefore, (4.20) holds trivially.

For the final case, we have that $\neg\mathsf{atom}(x)$ and $\neg\mathsf{atom}(y)$. Then $x = \mathsf{cons}(u_1, v_1)$ and $y = \mathsf{cons}(u_2, v_2)$ for some lists u_1, v_1, u_2, v_2. We have

$$\begin{array}{rll} x \preceq_{\mathsf{c}} y & \Leftrightarrow\; \mathsf{cons}(u_1, v_1) \preceq_{\mathsf{c}} \mathsf{cons}(u_2, v_2) & \text{assumption} \\ & \Leftrightarrow\; (u_1 = u_2 \;\wedge\; v_1 \preceq_{\mathsf{c}} v_2) \;\vee\; \mathsf{cons}(u_1, v_1) \preceq_{\mathsf{c}} v_2 & (\preceq_{\mathsf{c}}\ (4)) \end{array}$$

The disjunction suggests two possibilities. Consider the first disjunct. Because $v_1 \prec_{\mathsf{c}} \mathsf{cons}(u_1, v_1) = x$, we have that

$$(v_1, v_2) \prec_{\mathsf{c}}^2 (x, y) \ ,$$

allowing us to appeal to the inductive hypothesis (4.21): from $v_1 \preceq_{\mathsf{c}} v_2$, deduce that $|v_1| \leq |v_2|$. Then with two applications of (length list), we have

$$|x| \leq |y| \;\Leftrightarrow\; 1 + |v_1| \leq 1 + |v_2| \;\Leftrightarrow\; |v_1| \leq |v_2| \ .$$

Therefore, $|x| \leq |y|$ and (4.20) holds for this case.

Suppose the second disjunct ($\mathsf{cons}(u_1, v_1) \preceq_{\mathsf{c}} v_2$) holds. We again look to the inductive hypothesis (4.21). We have

$$(\mathsf{cons}(u_1, v_1), v_2) \prec_{\mathsf{c}}^2 (x, y)$$

because $\mathsf{cons}(u_1, v_1) = x$ and $v_2 \prec_{\mathsf{c}} \mathsf{cons}(u_2, v_2) = y$. Therefore, the inductive hypothesis tells us that $|x| \leq |v_2|$, while (length list) implies that $|v_2| < |y|$. In short,

$$|x| \leq |v_2| < |y| \ ,$$

which implies $|x| \leq |y|$ as desired, completing the proof. ■

Example 4.9. Augment the theory of Presburger arithmetic $T_{\mathbb{N}}$ (see Chapters 3 and 7) with the following axioms to define the Ackermann function:

- $\forall y.\ ack(0, y) = y + 1$ (ack left zero)
- $\forall x.\ ack(x + 1, 0) = ack(x, 1)$ (ack right zero)
- $\forall x, y.\ ack(x + 1, y + 1) = ack(x, ack(x + 1, y))$ (ack successor)

The Ackermann function grows quickly for increasing arguments:

- $ack(0, 0) = 1$
- $ack(1, 1) = 3$

- $ack(2,2) = 7$
- $ack(3,3) = 61$
- $ack(4,4) = 2^{2^{2^{2^{16}}}} - 3$

One might expect that proving properties about the Ackermann function would be difficult.

However, lexicographic well-founded induction allows us to reason about certain properties of the function. Define $<_2$ as the natural lexicographic extension of $<$ to pairs of natural numbers. Now consider input arguments to *ack* and the resulting arguments in recursive calls:

- (ack left zero) does not involve a recursive call.
- In (ack right zero), $(x+1, 0) >_2 (x, 1)$.
- In (ack successor),
  - $(x+1, y+1) >_2 (x+1, y)$, and
  - $(x+1, y+1) >_2 (x, ack(x+1, y))$.

As the arguments decrease according to $<_2$ with each level of recursion, we conclude that the computation of $ack(x, y)$ halts for every x and y. In Chapter 5, we show that finding well-founded relations is a general technique for showing that functions always halt.

Additionally, we can induct over the execution of *ack* to prove properties of the *ack* function itself. Let us prove that

$$\forall x, y.\ ack(x, y) > y \tag{4.22}$$

is $T_{\mathbb{N}}^{ack}$-valid. We apply lexicographic well-founded induction to the arguments of *ack* to prove

$$F[x, y] :\ ack(x, y) > y \tag{4.23}$$

for arbitrary natural numbers x and y. For the inductive hypothesis, assume that

$$\forall x', y'.\ (x', y') <_2 (x, y) \rightarrow \underbrace{ack(x', y') > y'}_{F[x', y']} . \tag{4.24}$$

Consider three cases: $x = 0$, $x > 0 \ \wedge \ y = 0$, and $x > 0 \ \wedge \ y > 0$.

If $x = 0$, then $ack(0, y) = y + 1 > y$ by (ack left zero), as desired.

If $x > 0 \ \wedge \ y = 0$, then

$$ack(x, 0) = ack(x - 1, 1)$$

by (ack right zero). Since

$$(x' : x - 1,\ y' : 1) <_2 (x, y) ,$$

the inductive hypothesis (4.24) tells us that

$$ack(x-1,1) > 1 \ .$$

Therefore, we have

$$ack(x,0) = ack(x-1,1) > 1 \ ,$$

so $ack(x,0) > 0$ as desired.

For the final case, $x > 0 \ \land \ y > 0$, we have

$$ack(x,y) = ack(x-1, ack(x, y-1))$$

by (ack successor). Since

$$(x' : x-1, \ y' : ack(x, y-1)) <_2 (x,y) \ ,$$

the inductive hypothesis (4.24) implies that

$$ack(x-1, ack(x,y-1)) > ack(x, y-1) \ .$$

Furthermore, since

$$(x' : x, \ y' : y-1) <_2 (x,y) \ ,$$

the inductive hypothesis (4.24) implies that

$$ack(x, y-1) > y-1 \ .$$

All together, then, we have

$$ack(x,y) = ack(x-1, ack(x,y-1)) > ack(x,y-1) > y-1 \ ;$$

hence, $ack(x,y) > (y-1)+1 = y$, completing the proof. ■

4.4 Structural Induction

Induction has many other applications outside of reasoning about the validity of first-order formulae. In this section, we introduce the proof technique of **structural induction** for proving properties about formulae themselves. Structural induction is applied in Section 2.7, in analyzing the quantifier elimination procedures of Chapter 7, and in other applications throughout the book.

Define the **strict subformula relation** over FOL formulae as follows: two formulae F_1 and F_2 are related by the strict subformula relation iff F_1 is a strict subformula of F_2. The strict subformula relation is well founded over the set of FOL formulae since every formula, having only a finite number of symbols, has only a finite number of strict subformulae; and each of its strict subformulae has fewer strict subformulae that it does. To prove a desired property of FOL formulae, instantiate the well-founded induction principle with the strict subformula relation:

- Assume as the **inductive hypothesis** that for arbitrary FOL formula F and for every strict subformula G of F, G has the desired property. Then prove that F has the property.

Since atoms do not have strict subformulae, they are treated as base cases. This induction principle is the **structural induction principle**.

Example 4.10. Exercise 1.3 asks the reader to prove that certain logical connectives are redundant in the presence of others. Formally, the exercise is asking the reader to prove the following claim: Every propositional formula F is equivalent to a propositional formula F' constructed with only the logical connectives $\top$, $\wedge$, and $\neg$.

There are three base cases to consider:

- The formula $\top$ can be represented directly as $\top$.
- The formula $\bot$ is equivalent to $\neg\top$.
- Any propositional variable P can be represented directly as P.

For the inductive step, consider formulae G, G_1, and G_2, and assume as the inductive hypothesis that each is equivalent to formulae G', G'_1, and G'_2, respectively, which are constructed only from the connectives $\top$, $\vee$, and $\neg$ (and propositional variables, of course). We show that each possible formulae that can be constructed from G, G_1, and G_2 with only one logical connective is equivalent to another constructed with only $\top$, $\vee$, and $\neg$:

- $\neg G$ is equivalent to $\neg G'$ from the inductive hypothesis.
- By considering the truth table in which the four possible valuations of G_1 and G_2 are considered, one can establish that $G_1 \vee G_2$ is equivalent to $\neg(\neg G'_1 \wedge \neg G'_2)$. By the inductive hypothesis, the latter formula is constructed only from propositional variables, $\top$, $\wedge$, and $\neg$.
- By similar reasoning, $G_1 \rightarrow G_2$ is equivalent to $\neg(G'_1 \wedge \neg G'_2)$, which satisfies the claim.
- Similar reasoning handles $G_1 \leftrightarrow G_2$ as well.

Hence, the claim is proved.

Note that the main argument is essentially similar to the answer that the reader might have provided in answering Exercise 1.3. Structural induction merely provides the basis for lifting the truth-table argument to a general statement about propositional formulae. ■

Structural induction is also useful for reasoning about interpretations of formulae, as the following example shows.

Example 4.11. This example relies on several basic concepts of set theory; however, even the reader unfamiliar with set theory can understand the application of structural induction without understanding the actual claim.

Consider $\Sigma_{\mathbb{Q}}$-formulae $F[x_1, \ldots, x_n]$ in which the only predicate is $\leq$, the only logical connectives are $\vee$ and $\wedge$, and the only quantifier is $\forall$. We prove that the set of satisfying $T_{\mathbb{Q}}$-interpretations of F (intuitively, those $T_{\mathbb{Q}}$-interpretations that assign to $x_1, \ldots, x_n$ values from $\mathbb{Q}^n$ that satisfy F) describes a closed subset of $\mathbb{Q}^n$.

For the base case, consider any inequality $\alpha \leq \beta$ with free variables $x_1, \ldots, x_n$. From basic set theory, the set of satisfying points is closed.

For the inductive step, consider formulae G, G_1, and G_2 constructed as specified. Assume as the inductive hypothesis that the satisfying $T_{\mathbb{Q}}$-interpretations for each comprise closed sets. Consider applying the allowed logical connectives and quantifier:

- $G_1 \wedge G_2$: The set described by this formula is the set-theoretic intersection of the sets described by G_1 and G_2, and is thus closed by the inductive hypothesis and set theory.
- $G_1 \vee G_2$: Similarly, the set described by this formula is the set-theoretic union of the sets described by G_1 and G_2, and is thus closed by the inductive hypothesis and set theory.
- $\forall x.\ G$: Consider subformula G with free variable x (if x is not free in G, then the formula is equivalent to just G, which describes a closed set by the inductive hypothesis). For each value $\frac{a}{b} \in \mathbb{Q}$, consider the formula

  $$G_{\frac{a}{b}} : G \wedge bx \leq a \wedge a \leq bx .$$

  The set described by each $G_{\frac{a}{b}}$ is closed according to the inductive hypothesis and reasoning similar to the previous cases. From set theory, the conjunction of all such sets is still closed, so the set of satisfying $T_{\mathbb{Q}}$-interpretations of $\forall x.\ G$ describes a closed set.

The induction is complete, so the claim is proved.

Results from Chapter 7 prove that $\exists$ also preserves closed sets in $T_{\mathbb{Q}}$. ■

Remark 4.12. Example 4.11 considers a subset of FOL formulae. However, this subset is by definition closed under conjunction, disjunction, and universal quantification: if F, F_1, and F_2 are in the subset, then so are $F_1 \wedge F_2$, $F_1 \vee F_2$, and $\forall x.\ F$; and conversely. In other words, all strict subformulae of a formula in the subset are also in the subset, so that structural induction is applicable.

The proof of Lemma 2.31 provides another example of the application of structural induction.

4.5 Summary

This chapter covers several induction principles in several first-order theories:

- *Stepwise induction* is presented in the context of integer arithmetic and lists. The induction principle requires defining a step such as adding one or constructing a list with one more element.

- *Complete induction* is presented in the context of integer arithmetic. The induction principle relies on the well-foundedness of the $<$ predicate. Rather than assuming that the desired property holds for one element n and proving the property for the case $n+1$ as in stepwise reduction, one assumes that the property holds for all elements $n' < n$ and proves that it holds for n. This stronger assumption sometime yields easier or more concise proofs.
- *Well-founded induction* generalizes complete induction to other theories; it is presented in the context of lists and lexicographic tuples. The induction principle requires a well-founded relation over the domain.
- *Structural induction* is an instance of well-founded induction in which the domain is formulae and the well-founded relation is the strict subformula relation.

Besides being an important tool for proving first-order validities, induction is the basis for both verification methodologies studied in Chapter 5. Structural induction also serves as the basis for the quantifier elimination procedures studied in Chapter 7.

Bibliographic Remarks

The induction proofs in Examples 4.1, 4.3, and 4.9 are taken from the text of Manna and Waldinger [55].

Blaise Pascal (1623–1662) and Jacob Bernoulli (1654–1705) are recognized as having formalized stepwise and complete induction, respectively. Less formal versions of induction appear in texts by Francesco Maurolico (1494–1575); Rabbi Levi Ben Gershon (1288–1344), who recognized induction as a distinct form of mathematical proof; Abu Bekr ibn Muhammad ibn al-Husayn Al-Karaji (953–1029); and Abu Kamil Shuja Ibn Aslam Ibn Mohammad Ibn Shaji (850–930) [97]. Some historians claim that Euclid may have applied induction informally.

Exercises

4.1 (T_{cons}^{+})**.** Prove the following in T_{cons}^{+}:

(a) $\forall u, v.\ \mathit{flat}(u) \ \land\ \mathit{flat}(v) \ \rightarrow\ \mathit{flat}(\mathit{concat}(u, v))$
(b) $\forall u.\ \mathit{flat}(u) \ \rightarrow\ \mathit{flat}(\mathit{rvs}(u))$

4.2 ($T_{\mathsf{cons}}^{\mathsf{PA}}$)**.** Prove or disprove the following in $T_{\mathsf{cons}}^{\mathsf{PA}}$:

(a) $\forall u.\ u \preceq_{\mathsf{c}} u$
(b) $\forall u, v, w.\ \mathsf{cons}(u, v) \preceq_{\mathsf{c}} w \ \rightarrow\ v \preceq_{\mathsf{c}} w$
(c) $\forall u, v.\ v \prec_{\mathsf{c}} \mathsf{cons}(u, v)$

4.3 ($T^{+}_{\mathsf{cons}} \cup T^{\mathsf{PA}}_{\mathsf{cons}}$). Prove the following in $T^{+}_{\mathsf{cons}} \cup T^{\mathsf{PA}}_{\mathsf{cons}}$:

(a) $\forall u, v.\ |concat(u, v)| = |u| + |v|$
(b) $\forall u.\ flat(u) \ \rightarrow\ |rvs(u)| = |u|$

4.4 (Chips). Does the process of Example 4.7 still halt if

(a) in Step 1, you return one red chip to the bag?
(b) in Step 1, you add one blue chip?
(c) in Step 1, you add one blue chip; and in Step 3, you return the blue chip to the bag but do not add any other chips?

4.5 (Strict sublist). Modify Example 4.8 to prove

$$\forall x, y.\ x \prec_{\mathsf{c}} y \ \rightarrow\ |x| < |y| \ .$$

4.6 (Structural induction). Prove that every first-order formula F is equivalent to a first-order formula F' constructed with only the logical connectives $\top$, $\wedge$, and $\neg$ and the quantifier $\forall$.

4.7 (Finite number of sublists). Prove that the number of sublists of a list (defined in $T^{\mathsf{PA}}_{\mathsf{cons}}$) is finite.

4.8 (★$\prec_{\mathsf{c}}$ is well-founded). Prove that $\prec_{\mathsf{c}}$, defined in $T^{\mathsf{PA}}_{\mathsf{cons}}$, is well-founded over lists. To avoid circularity, do not apply well-founded induction in this proof. *Hint*: Prove that $\prec_{\mathsf{c}}$ is transitive and **irreflexive** ($\forall u.\ \neg(u \prec_{\mathsf{c}} u)$); then apply Exercise 4.7.

5

Program Correctness: Mechanics

When examining the detail of the algorithm, it seems probable that the proof will be helpful in explaining not only what *is happening but* why.
— Tony Hoare
An Axiomatic Basis for Computer Programming, 1969

We are finally ready to apply FOL and induction to a real problem: specifying and proving properties of programs. In this chapter, we develop the three foundational methods that underly all verification and program analysis techniques. In the next chapter, we discuss strategies for applying them.

First, **specification** is the precise statement of properties that a program should exhibit. The language of FOL offers precision. The remaining task is to develop a scheme for embedding FOL statements into program text as **program annotations**. We focus on two forms of properties. **Partial correctness** properties, or **safety** properties, assert that certain states — typically, error states — cannot ever occur during the execution of a program. An important subset of this form of property is the partial correctness of programs: *if* a program halts, then its output satisfies some relation with its input. **Total correctness** properties, or **progress** properties, assert that certain states are eventually reached during program execution. Section 5.1 presents specification in the context of a simple programming language, pi.

The next foundational method is the **inductive assertion method** for proving partial correctness properties. The inductive assertion method is based on the mathematical induction of Chapter 4. To prove that every state during the execution of a program satisfies FOL formula F, prove as the base case that F holds at the beginning of execution; assume as the inductive hypothesis that F currently holds (at some point during the execution); and prove as the inductive step that F holds after one more step of the program. Section 5.2 discusses the mechanics for reducing a program with a partial correctness specification to this inductive argument. The challenge in applying this method is to discover additional annotations to make the induction go through. Chapter 6 discusses strategies for finding the extra information.

The third foundational method is the **ranking function method** for proving total correctness properties. Proving total correctness breaks down into two arguments. First, one proves that some partial correctness property holds using the inductive assertion method; second, one argues that some set of loops and recursive functions always halt. The ranking function method applies to the latter argument: one associates with each loop and recursive function a **ranking function** that maps the program variables to a well-founded domain (see Chapter 4); then one proves that whenever program control moves from one ranking function to the next, the value decreases according to the well-founded relation. Since the relation is well-founded, the looping and recursion must eventually halt. A typical total correctness property asserts that the program halts *and* its output satisfies some relation with its input. The ranking function method applies to the first conjunct (the program halts); the inductive assertion method applies to the second (its output satisfies some relation with its input). Section 5.3 presents the ranking function method.

We explain all concepts in this and the next chapter with a set of example programs that manipulate arrays. We chose these programs for several reasons. First, they should be familiar to most readers; the reader can thus focus on the verification methodology rather than on understanding new complex programs. Second, our correctness proofs rely heavily on the decision procedures that are discussed in Part II of this book. Finally, they are small but dense, allowing us to exhibit common techniques for proving interesting facts about programs. The reader should keep in mind, however, that the methods of this chapter underlie software and hardware analyses that are applied in practice.

The reader may find the contents of this chapter to be rather technical. Indeed, the transformation of an annotated program into a set of verification conditions is a purely mechanical task. Fortunately, a **verifying compiler** does this work in practice: it parses an annotated program, checking the syntax and semantics as usual, and generates a set of verification conditions. But just as learning how compilers work is important for understanding programming languages, learning how verifying compilers work is important for understanding verification and programming languages with annotations. Moreover, applying the steps of generating verification conditions provides practice in manipulating and understanding FOL formulae, a useful skill. Chapter 6 discusses strategies for writing annotations, a task that cannot be fully automated.

5.1 pi: A Simple Imperative Language

This section introduces the programming language pi, an imperative language with facilities for annotations. To allow us to focus on the fundamentals of program verification, pi lacks complicating features of typical imperative lan-

```
@pre ⊤
@post ⊤
bool LinearSearch(int[] a, int ℓ, int u, int e) {
  for @ ⊤
    (int i := ℓ; i ≤ u; i := i + 1) {
    if (a[i] = e) return true;
  }
  return false;
}
```

Fig. 5.1. LinearSearch

guages: its data types do not include pointer or reference types; and it does not allow global variables, although it does have global constants (see Exercise 6.5). After reading this chapter and Chapter 12, the interested reader should consult the wide literature on program analysis to learn how the techniques of these chapters extend to reasoning about standard programming languages.

5.1.1 The Language

Because pi is superficially a C-like language with restrictions, we present the essential features of pi through examples.

Example 5.1. Figure 5.1 lists the function LinearSearch, which searches the range $[\ell, u]$ of an array a of integers for a value e. It returns `true` iff the given array contains the value between the lower bound ℓ and upper bound u. It behaves correctly only if $0 \leq \ell$ and $u < |a|$; otherwise, the array a is accessed outside of its domain $[0, |a| - 1]$. $|a|$ denotes the length of array a.

Observe that most of the syntax is similar to C. For example, the `for` loop sets i to be ℓ initially and then executes the body of the loop and increments i by 1 as long as $i \leq u$. Also, an integer array has type `int`[], which is constructed from base type `int`. One syntactic difference occurs in assignment, which is written := to distinguish it from the equality predicate =. We use = as the equality predicate, rather than ==, to correspond to the standard equality predicate of FOL. Finally, unlike C, pi has type `bool` and constants `true` and `false`.

Notice the lines beginning with @. They are program annotations, which we discuss in detail in the next section.

In LinearSearch, a, ℓ, u, and e are the **formal parameters** (also, **parameters**) of the function. If LinearSearch is called as LinearSearch$(b, 0, |b| - 1, v)$, then b, 0, $|b| - 1$, and v are the **arguments**. ■

Example 5.2. Figure 5.2 lists the recursive function BinarySearch, which searches a range $[\ell, u]$ of a sorted (weakly increasing: $a[i] \leq a[j]$ if $i \leq j$) array a of integers for a value e. Like LinearSearch, it returns `true` iff the

```
@pre ⊤
@post ⊤
bool BinarySearch(int[] a, int ℓ, int u, int e) {
  if (ℓ > u) return false;
  else {
    int m := (ℓ + u) div 2;
    if (a[m] = e) return true;
    else if (a[m] < e) return BinarySearch(a, m + 1, u, e);
    else return BinarySearch(a, ℓ, m − 1, e);
  }
}
```

Fig. 5.2. BinarySearch

given array contains the value in the range $[\ell, u]$. It behaves correctly only if $0 \leq \ell$ and $u < |a|$.

One level of recursion operates as follows. If the lower bound ℓ of the range is greater than the upper bound u, then the (empty) subarray cannot contain e, so it returns `false`. Otherwise, it examines the middle element $a[m]$ of the subarray: if it is e, then the subarray clearly contains e; otherwise, it recurses on the left half if $a[m] < e$ and on the right half if $a[m] > e$.

pi syntactically distinguishes between integer division and real division: for `int` variables a and b, write a `div` b instead of a/b. Integer division is defined as follows:

$$a \ \texttt{div} \ b \ \stackrel{\text{def}}{=} \ \left\lfloor \frac{a}{b} \right\rfloor .$$

That is, a `div` b is equal to the greatest integer less than or equal to $\frac{a}{b}$ (the *floor* of $\frac{a}{b}$). ■

Example 5.3. Figure 5.3 lists the function BubbleSort, which sorts an integer array. It works by "bubbling" the largest element of the left unsorted region of the array toward the sorted region on the right; this element then becomes the left element of the sorted region, enlarging the region by one cell. In Figure 5.4, for example, the first line shows an array in which the rightmost boxed cells comprise the sorted region and the other cells comprise the unsorted region. In the final line, the sorted region has been expanded by one cell.

Figure 5.4 lists a portion of a sample execution trace. The right two cells (5, 6) of the array have already been sorted. In the trace, the inner loop moves the largest element 4 of the unsorted region to the right to join the sorted region, which is indicated by the dotted rectangle. In the first two steps, $a[j] \leq a[j+1]$ ($2 \leq 3$ and $3 \leq 4$), so the values of cell j and $j+1$ are not swapped in either case. In the subsequent two steps, $a[j] > a[j+1]$ ($4 > 1$ and $4 > 2$), causing a swap at each step. In the fifth step, the inner loop's guard $i < j$ no longer holds, so the inner loop exits and the outer

```
@pre $\top$
@post $\top$
int[] BubbleSort(int[] $a_0$) {
  int[] $a := a_0$;
  for @ $\top$
    (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    for @ $\top$
      (int $j := 0$; $j < i$; $j := j + 1$) {
      if ($a[j] > a[j+1]$) {
        int $t := a[j]$;
        $a[j] := a[j+1]$;
        $a[j+1] := t$;
      }
    }
  }
  return $a$;
}
```

Fig. 5.3. BubbleSort

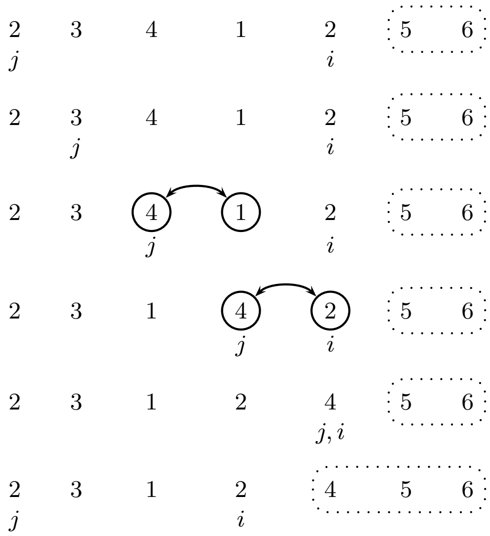

Fig. 5.4. Sample execution of BubbleSort

loop decrements i by 1. The sorted region has been expanded by one cell, as indicated by the final dotted rectangle. The last step shows the beginning of the next round of the inner loop.

Because pi does not have pointer or reference types, all data are passed by value, including arrays and structures. If BubbleSort were missing the `return`

```
typedef struct qs {
  int pivot;
  int[] array;
} qs;
```

Fig. 5.5. Structure qs

statement, then calling it would not have any discernible effect on the calling context. Additionally, pi does not allow updates to parameters, so BubbleSort assigns a_0 to a fresh variable a in the first line. This artificial requirement makes reasoning about functions easier: in annotations (see Section 5.1.2) throughout the function, one can always reference the input. ■

In this book, our example programs manipulate arrays rather than recursive data structures. The reason is that we can express more interesting properties about arrays in the fragment of the theory of arrays studied in Chapter 11 than we can about lists in the fragment of the theory of recursive data structures studied in Chapter 9. This bias is a reflection of the structure and content of this book, not of what is theoretically possible.

However, we sometimes use records, a basic recursive data type, to allow a function to return multiple values. The following example illustrates such a record type, which is used in the program QuickSort (see Section 6.2).

Example 5.4. The structure qs of Figure 5.5 is a record with two fields: the *pivot* field of type `int` and the *array* field of type `array`. If x is a variable of type qs, then $x.pivot$ returns the value in its *pivot* field; also, $x.array[i] := v$ assigns v to position i of x's *array* field. ■

5.1.2 Program Annotations

The most important feature of pi is the capacity for complex function annotations. An **annotation** is a FOL formula F whose free variables include only the program variables of the function in which the annotation occurs. An annotation F at location L asserts that F is true whenever program control reaches L. We discuss several forms of annotations in this section.

Function Specifications

The **function specification** of a function is a pair of annotations. The **function precondition** is a formula F whose free variables include only the formal parameters. It specifies what should be true upon entering the function — or, in other words, under what inputs the function is expected to work. The **function postcondition** is a formula G whose free variables include only the formal parameters and the special variable rv representing the return value of the function. The postcondition relates the function's output (the return value rv) to its input (the parameters).

```
@pre $0 \le \ell \;\land\; u < |a|$
@post $rv \;\leftrightarrow\; \exists i.\; \ell \le i \le u \;\land\; a[i] = e$
bool LinearSearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  for @ $\top$
    (int $i := \ell$; $i \le u$; $i := i + 1$) {
    if ($a[i] = e$) return true;
  }
  return false;
}
```

Fig. 5.6. LinearSearch with function specification

```
@pre $\top$
@post $rv \;\leftrightarrow\; \exists i.\; 0 \le \ell \le i \le u < |a| \;\land\; a[i] = e$
bool LinearSearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  if ($\ell < 0 \;\lor\; u \ge |a|$) return false;
  for @ $\top$
    (int $i := \ell$; $i \le u$; $i := i + 1$) {
    if ($a[i] = e$) return true;
  }
  return false;
}
```

Fig. 5.7. Robust LinearSearch with function specification

Example 5.5. In Example 5.1, we informally specified the behavior of LinearSearch as follows: LinearSearch returns `true` iff the array a contains the value e in the range $[\ell, u]$. It behaves correctly only when $\ell \ge 0$ and $u < |a|$.

Function specifications formalize such statements. Figure 5.6 presents LinearSearch with its specification. The precondition asserts that the lower bound ℓ should be at least 0 and that the upper bound u should be less than the length $|a|$ of the array a. The postcondition asserts that the return value rv is `true` iff $a[i] = e$ for some index $i \in [\ell, u]$ of a. ■

Example 5.6. A nontrivial precondition (a formula other than $\top$) is not always acceptable, especially if a function is public to a module. Figure 5.7 lists a more robust version of linear search. The formula

$$0 \le \ell \le i \le u < |a|$$

abbreviates

$$0 \le \ell \;\land\; \ell \le i \;\land\; i \le u \;\land\; u < |a| \;.$$

A nontrivial precondition is sometimes acceptable for a function that is private to a module. The verification method of this chapter checks that every instance of a call to such a function obeys the precondition. ■

```
@pre $0 \leq \ell \ \wedge \ u < |a| \ \wedge \ \mathsf{sorted}(a, \ell, u)$
@post $rv \ \leftrightarrow \ \exists i.\ \ell \leq i \leq u \ \wedge \ a[i] = e$
bool BinarySearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  if ($\ell > u$) return false;
  else {
    int $m := (\ell + u)$ div 2;
    if ($a[m] = e$) return true;
    else if ($a[m] < e$) return BinarySearch($a, m + 1, u, e$);
    else return BinarySearch($a, \ell, m - 1, e$);
  }
}
```

Fig. 5.8. BinarySearch with function specification

```
@pre $\top$
@post sorted($rv, 0, |rv| - 1$)
int[] BubbleSort(int[] $a_0$) {
  int[] $a := a_0$;
  for @ $\top$
    (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    for @ $\top$
      (int $j := 0$; $j < i$; $j := j + 1$) {
      if ($a[j] > a[j + 1]$) {
        int $t := a[j]$;
        $a[j] := a[j + 1]$;
        $a[j + 1] := t$;
      }
    }
  }
  return $a$;
}
```

Fig. 5.9. BubbleSort with function specification

Example 5.7. Figure 5.8 lists BinarySearch with its specification. As expected, its postcondition is identical to the postcondition of LinearSearch. However, its precondition also states that the array a is sorted.

The **sorted** predicate is defined in the combined theory of integers and arrays, $T_{\mathbb{Z}} \cup T_{\mathsf{A}}$:

$$\mathsf{sorted}(a, \ell, u) \ \Leftrightarrow \ \forall i, j.\ \ell \leq i \leq j \leq u \ \rightarrow \ a[i] \leq a[j] \ .$$

■

Example 5.8. Figure 5.9 lists BubbleSort with its specification. Given any array, the returned array is sorted. Of course, other properties are desirable and could be specified as well. For example, the returned array rv should be a permutation of the original array a_0 (see Exercise 6.5). ■

Section 5.2 presents a method for proving that a function satisfies its partial correctness specification: if the function precondition is satisfied and the function halts, then the function postcondition holds upon return. Section 5.3 discusses a method for proving that, additionally, the function always halts.

Loop Invariants

Each `for` loop and `while` loop has an attendant annotation called the **loop invariant**. A `while` loop

```
while
  @ $F$
  ($\langle condition \rangle$) {
  $\langle body \rangle$
}
```

says to apply the $\langle body \rangle$ as long as $\langle condition \rangle$ holds. The assertion F must hold at the beginning of every iteration. It is evaluated before the $\langle condition \rangle$ is evaluated, so it must hold even on the final iteration when $\langle condition \rangle$ is `false`. Therefore, on entering the $\langle body \rangle$ of the loop,

$$F \;\wedge\; \langle condition \rangle$$

must hold, and on exiting the loop,

$$F \;\wedge\; \neg\langle condition \rangle$$

must hold.

To consider a `for` loop, translate the loop

```
for
  @ $F$
  ($\langle initialize \rangle$; $\langle condition \rangle$; $\langle increment \rangle$) {
  $\langle body \rangle$
}
```

into the equivalent loop

```
$\langle initialize \rangle$;
while
  @ $F$
  ($\langle condition \rangle$) {
  $\langle body \rangle$
  $\langle increment \rangle$
}
```

F must hold after the $\langle initialize \rangle$ statement has been evaluated and, on each iteration, before the $\langle condition \rangle$ is evaluated.

```
@pre $0 \leq \ell \;\wedge\; u < |a|$
@post $rv \;\leftrightarrow\; \exists i.\ \ell \leq i \leq u \;\wedge\; a[i] = e$
bool LinearSearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  for
    @$L$ : $\ell \leq i \;\wedge\; (\forall j.\ \ell \leq j < i \;\rightarrow\; a[j] \neq e)$
    (int $i := \ell$; $i \leq u$; $i := i + 1$) {
    if ($a[i] = e$) return true;
  }
  return false;
}
```

Fig. 5.10. LinearSearch with loop invariant

Example 5.9. Figure 5.10 lists LinearSearch with a nontrivial loop invariant at L. It asserts that whenever control reaches L, the loop index is at least ℓ and that $a[j] \neq e$ for previously examined indices j. ■

Section 5.2 shows that loop invariants are crucial for constructing an inductive argument that a function obeys its specification.

Assertions

In pi, one can add an annotation anywhere. When an annotation is not a function precondition, function postcondition, or loop invariant, we call it an **assertion**. Assertions allow programmers to provide a formal comment. For example, if at the statement

$$i := i + k;$$

the programmer thinks that k is positive, then the programmer can add an assertion stating that supposition:

```
@ $k > 0$;
$i := i + k$;
```

Later, the programmer's hyothesis about k is verified with formal verification at compile time or with dynamic assertion tests at runtime.

Runtime assertions are a special class of assertions. In most programming languages, **runtime errors** include division by 0, modulo by 0, and dereference of `null`. In particular, division by 0 and modulo by 0 cause hardware exceptions, while only some languages, such as Java, catch a dereference of `null`. In pi, runtime errors include division by 0, modulo by 0, and accessing an array out of bounds. The pi compiler generates runtime assertions to catch runtime errors.

Example 5.10. Figure 5.11 lists LinearSearch with runtime assertions. The array read $a[i]$ is protected by the assertion that i is a legal index of a. ■

```
@pre $\top$
@post $\top$
bool LinearSearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  for @ $\top$
    (int $i := \ell$; $i \leq u$; $i := i + 1$) {
    @ $0 \leq i < |a|$;
    if ($a[i] = e$) return true;
  }
  return false;
}
```

Fig. 5.11. LinearSearch with runtime assertions

```
@pre $\top$
@post $\top$
bool BinarySearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  if ($\ell > u$) return false;
  else {
    @ $2 \neq 0$;
    int $m := (\ell + u)$ div $2$;
    @ $0 \leq m < |a|$;
    if ($a[m] = e$) return true;
    else {
      @ $0 \leq m < |a|$;
      if ($a[m] < e$) return BinarySearch($a, m + 1, u, e$);
      else return BinarySearch($a, \ell, m - 1, e$);
    }
  }
}
```

Fig. 5.12. BinarySearch with runtime assertions

Example 5.11. Figure 5.12 lists BinarySearch with runtime assertions. The first assertion protects the division: it asserts that $2 \neq 0$, which clearly holds. The next two assertions protect the array reads. ■

Example 5.12. Figure 5.13 lists BubbleSort with compiler-generated runtime assertions. All assertions protect array accesses. The first two runtime assertions are sufficient to protect all array accesses. Figure 5.14 lists a concise version. ■

5.2 Partial Correctness

Having specified and implemented each function of a program, we would like to prove that the functions obey their specifications (from another perspective,

```
@pre $\top$
@post $\top$
int[] BubbleSort(int[] $a_0$) {
  int[] $a := a_0$;
  for @ $\top$
    (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    for @ $\top$
      (int $j := 0$; $j < i$; $j := j + 1$) {
      @ $0 \leq j < |a|$;
      @ $0 \leq j + 1 < |a|$;
      if ($a[j] > a[j + 1]$) {
        @ $0 \leq j < |a|$;
        int $t := a[j]$;
        @ $0 \leq j < |a|$;
        @ $0 \leq j + 1 < |a|$;
        $a[j] := a[j + 1]$;
        @ $0 \leq j + 1 < |a|$;
        $a[j + 1] := t$;
      }
    }
  }
  return $a$;
}
```

Fig. 5.13. BubbleSort with runtime assertions

that their specifications reflect their actual behavior). A function is **partially correct** if when the function's precondition is satisfied on entry, its postcondition is satisfied when the function returns (if it ever does). In general, functions may not halt: a loop's guard could be incorrect, or a recursive function could fail to handle a particular base case. Section 5.3 discusses how to prove that a function always halts.

We present the **inductive assertion method** for proving that a program is partially correct. The method reduces each function and its annotations to a finite set of **verification conditions** (**VC**s), which are FOL formulae. If all of a function's VCs are valid, then the function obeys its specification. The reduction occurs in two stages: first, each function of the annotated program is broken down into a finite set of **basic paths** (Sections 5.2.1 and 5.2.2); second, each basic path generates a verification condition (Section 5.2.4). Sections 5.2.3 and 5.2.5 provide a more abstract view of the inductive assertion method.

Loops complicate proofs of partial correctness because they create an unbounded number of paths from function entry to exit. Recursive functions similarly complicate proofs. A **path** is a sequence of program statements. For loops, loop invariants cut the paths into a finite set of basic paths, while for recursive functions, the function specification of the recursive function cuts the paths. In this section, we assume that we are given function specifications

```
@pre $\top$
@post $\top$
int[] BubbleSort(int[] $a_0$) {
  int[] $a := a_0$;
  for @ $\top$
    (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    for @ $\top$
      (int $j := 0$; $j < i$; $j := j + 1$) {
      @ $0 \leq j < |a| \ \wedge\ 0 \leq j + 1 < |a|$;
      if ($a[j] > a[j + 1]$) {
        int $t := a[j]$;
        $a[j] := a[j + 1]$;
        $a[j + 1] := t$;
      }
    }
  }
  return $a$;
}
```

Fig. 5.14. BubbleSort with compressed runtime assertions

```
@pre $0 \leq \ell \ \wedge\ u < |a|$
@post $rv \ \leftrightarrow\ \exists i.\ \ell \leq i \leq u \ \wedge\ a[i] = e$
bool LinearSearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  for
    @$L$ : $\ell \leq i \ \wedge\ (\forall j.\ \ell \leq j < i \ \rightarrow\ a[j] \neq e)$
    (int $i := \ell$; $i \leq u$; $i := i + 1$) {
    if ($a[i] = e$) return true;
  }
  return false;
}
```

Fig. 5.15. LinearSearch with loop invariants

and loop invariants and concentrate on the task of generating the corresponding verification conditions. In practice, this task is performed by a verifying compiler. Chapter 6 discusses strategies for constructing specifications and loop invariants.

5.2.1 Basic Paths: Loops

A **basic path** is a sequence of instructions that begins at the function precondition or a loop invariant and ends at a loop invariant, an assertion, or the function postcondition. Moreover, a loop invariant can only occur at the beginning or the ending of a basic path. Thus, basic paths do not cross loops. The following examples illustrate the characteristics of basic paths.

Example 5.13. Figure 5.15 lists an annotated version of LinearSearch. Its basic paths are the following. The first basic path starts at the function precondition, enters the `for` loop via the initialization statement, and ends at the loop invariant L:

(1)

```
@pre $0 \leq \ell \ \wedge\ u < |a|$
$i := \ell;$
@$L:\ \ell \leq i \ \wedge\ (\forall j.\ \ell \leq j < i \ \rightarrow\ a[j] \neq e)$
```

The second basic path begins at the loop invariant at L, passes the loop guard $i \leq u$, passes the guard $a[i] = e$ of the `if` statement, executes the `return` (of `true`), and ends at the postcondition:

(2)

```
@$L:\ \ell \leq i \ \wedge\ (\forall j.\ \ell \leq j < i \ \rightarrow\ a[j] \neq e)$
assume $i \leq u$;
assume $a[i] = e$;
$rv :=$ true;
@post $rv \ \leftrightarrow\ \exists j.\ \ell \leq j \leq u \ \wedge\ a[j] = e$
```

This path exhibits two new aspects of basic paths. First, `return` statements become assignments to the special variable rv representing the return value.

Second, guards arising in program statements (in `for` loop guards, `while` loop guards, or `if` statements) become **assume statements** in basic paths. An assume statement `assume` c in a basic path means that the remainder of the basic path is executed only if the condition c holds at `assume` c. Each guard with condition c results in two assumptions: the guard holds (c) or it does not hold ($\neg c$). Therefore, each guard produces two paths with the same prefix up to the guard. They diverge on the assumption: one basic path has the statement `assume` c, and the other has the statement `assume` $\neg c$. These assumptions and the control structure of the program determine the construction of the remainder of the basic paths.

For example, the third path has the same prefix as **(2)** but makes the opposite assumption at the `if` statement guard: it assumes $a[i] \neq e$ rather than $a[i] = e$. Therefore, this path loops back around to the loop invariant:

(3)

```
@$L:\ \ell \leq i \ \wedge\ (\forall j.\ \ell \leq j < i \ \rightarrow\ a[j] \neq e)$
assume $i \leq u$;
assume $a[i] \neq e$;
$i := i + 1;$
@$L:\ \ell \leq i \ \wedge\ (\forall j.\ \ell \leq j < i \ \rightarrow\ a[j] \neq e)$
```

The final basic path has the same prefix as **(2)** and **(3)** but makes the opposite assumption at the `for` loop guard: it assumes $i > u$ rather than $i \leq u$. Therefore, this path exits the loop and returns `false`:

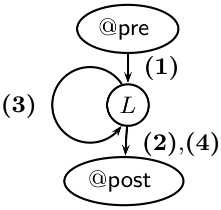

Fig. 5.16. Visualization of basic paths of LinearSearch

(4)

```
@L : $\ell \le i \ \wedge\ (\forall j.\ \ell \le j < i \ \rightarrow\ a[j] \ne e)$
assume $i > u$;
$rv$ := false;
@post $rv \ \leftrightarrow\ \exists j.\ \ell \le j \le u \ \wedge\ a[j] = e$
```

To avoid forgetting a basic path, we list paths in a depth-first order. When a guard is encountered, assume that it holds and generate the resulting paths; then assume that it does not hold and generate the resulting paths. In this example, the loop guard $i \le u$ in **(2)** is first encountered; **(2)** and **(3)** follow from the assumption that the loop guard holds, while **(4)** follows from the assumption that it does not hold. The `if` statement guard $a[i] = e$ is next encountered in **(2)**; **(2)** follows from the assumption that it holds, while **(3)** follows from the assumption that it does not hold. Figure 5.16 visualizes these basic paths. ■

Example 5.14. Figure 5.17 lists BubbleSort with loop invariants. The outer loop invariant at L_1 asserts that

- i is in the range $[-1, |a| - 1]$ (if $|a| = 0$, then i is initially -1);
- a is sorted in the range $[i, |a| - 1]$;
- and a is **partitioned** such that each element in the range $[0, i]$ is at most (less than or equal to) each element in the range $[i + 1, |a| - 1]$.

Its inner loop invariant at L_2 asserts that

- i is in the range $[1, |a| - 1]$, and j is in the range $[0, i]$;
- a is sorted in the range $[i, |a| - 1]$ as in the outer loop;
- a is partitioned as in the outer loop;
- and a is also partitioned such that each element in the range $[0, j - 1]$ is at most $a[j]$.

The **partitioned** predicate is defined in the theory $T_{\mathbb{Z}} \cup T_{\mathsf{A}}$:

$$\begin{aligned}&\mathsf{partitioned}(a, \ell_1, u_1, \ell_2, u_2)\\ &\quad\Leftrightarrow\ \forall i, j.\ \ell_1 \le i \le u_1 < \ell_2 \le j \le u_2 \ \rightarrow\ a[i] \le a[j]\ .\end{aligned}$$

```
@pre $\top$
@post sorted$(rv, 0, |rv| - 1)$
int[] BubbleSort(int[] $a_0$) {
  int[] $a := a_0$;
  for
    @$L_1$ : $\begin{bmatrix} -1 \le i < |a| \\ \wedge\ \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge\ \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$
    (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    for
      @$L_2$ : $\begin{bmatrix} 1 \le i < |a| \ \wedge\ 0 \le j \le i \\ \wedge\ \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge\ \mathsf{partitioned}(a, 0, j-1, j, j) \\ \wedge\ \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$
      (int $j := 0$; $j < i$; $j := j + 1$) {
      if ($a[j] > a[j+1]$) {
        int $t := a[j]$;
        $a[j] := a[j+1]$;
        $a[j+1] := t$;
      }
    }
  }
  return $a$;
}
```

Fig. 5.17. BubbleSort with loop invariants

Performing a depth-first exploration, the first basic path starts at the precondition and ends at the outer loop invariant at L_1:

(1)

```
@pre $\top$;
$a := a_0$;
$i := |a| - 1$;
@$L_1$ : $-1 \le i < |a| \ \wedge\ \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \ \wedge\ \mathsf{sorted}(a, i, |a|-1)$
```

The second basic path starts at L_1 and ends at the inner loop invariant at L_2 (recall that the annotation is checked after the loop initialization $j := 0$):

(2)

```
@$L_1$ : $-1 \le i < |a| \ \wedge\ \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \ \wedge\ \mathsf{sorted}(a, i, |a|-1)$
assume $i > 0$;
$j := 0$;
@$L_2$ : $\begin{bmatrix} 1 \le i < |a| \ \wedge\ 0 \le j \le i \ \wedge\ \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge\ \mathsf{partitioned}(a, 0, j-1, j, j) \ \wedge\ \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$
```

The third and fourth basic paths follow the inner loop, each handling one assumption on the guard $a[j] > a[j+1]$ of the `if` statement:

(3)

```
$@L_2 : \left[\begin{array}{l} 1 \le i < |a| \;\wedge\; 0 \le j \le i \;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \;\wedge\; \mathsf{partitioned}(a, 0, j-1, j, j) \;\wedge\; \mathsf{sorted}(a, i, |a|-1) \end{array}\right]$
assume $j < i$;
assume $a[j] > a[j+1]$;
$t := a[j]$;
$a[j] := a[j+1]$;
$a[j+1] := t$;
$j := j + 1$;
$@L_2 : \left[\begin{array}{l} 1 \le i < |a| \;\wedge\; 0 \le j \le i \;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \;\wedge\; \mathsf{partitioned}(a, 0, j-1, j, j) \;\wedge\; \mathsf{sorted}(a, i, |a|-1) \end{array}\right]$
```

(4)

```
$@L_2 : \left[\begin{array}{l} 1 \le i < |a| \;\wedge\; 0 \le j \le i \;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \;\wedge\; \mathsf{partitioned}(a, 0, j-1, j, j) \;\wedge\; \mathsf{sorted}(a, i, |a|-1) \end{array}\right]$
assume $j < i$;
assume $a[j] \le a[j+1]$;
$j := j + 1$;
$@L_2 : \left[\begin{array}{l} 1 \le i < |a| \;\wedge\; 0 \le j \le i \;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \;\wedge\; \mathsf{partitioned}(a, 0, j-1, j, j) \;\wedge\; \mathsf{sorted}(a, i, |a|-1) \end{array}\right]$
```

The fifth basic path starts at L_2, exits the inner loop, and decrements i on its way to L_1:

(5)

```
$@L_2 : \left[\begin{array}{l} 1 \le i < |a| \;\wedge\; 0 \le j \le i \;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \;\wedge\; \mathsf{partitioned}(a, 0, j-1, j, j) \;\wedge\; \mathsf{sorted}(a, i, |a|-1) \end{array}\right]$
assume $j \ge i$;
$i := i - 1$;
$@L_1 : \; -1 \le i < |a| \;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \;\wedge\; \mathsf{sorted}(a, i, |a|-1)$
```

The final basic path starts at L_1, exits the outer loop, and then exits the function, returning the (presumably sorted) array a:

(6)

```
$@L_1 : \; -1 \le i < |a| \;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \;\wedge\; \mathsf{sorted}(a, i, |a|-1)$
assume $i \le 0$;
$rv := a$;
@post sorted$(rv, 0, |rv| - 1)$
```

Figure 5.18 visualizes these basic paths. ■

Example 5.15. Figure 5.19 lists BubbleSort with runtime assertions at L_3 and a different set of loop invariants at L_1 and L_2 relevant for proving the runtime assertions. Six basic paths correspond in structure to those of Figure 5.18, although their content varies based on the new precondition, postcondi-

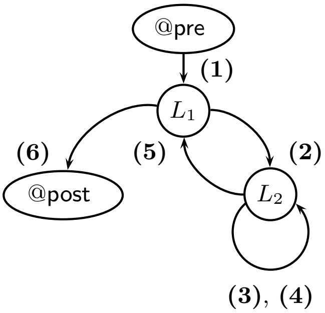

Fig. 5.18. Visualization of basic paths of BubbleSort

```
@pre $\top$
@post $\top$
int[] BubbleSort(int[] $a_0$) {
  int[] $a := a_0$;
  for
    @$L_1$ : $-1 \leq i < |a|$
    (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    for
      @$L_2$ : $0 < i < |a| \;\wedge\; 0 \leq j \leq i$
      (int $j := 0$; $j < i$; $j := j + 1$) {
      @$L_3$ : $0 \leq j < |a| \;\wedge\; 0 \leq j + 1 < |a|$;
      if ($a[j] > a[j + 1]$) {
        int $t := a[j]$;
        $a[j] := a[j + 1]$;
        $a[j + 1] := t$;
      }
    }
  }
  return $a$;
}
```

Fig. 5.19. BubbleSort with runtime assertions

tion, and loop invariants. These basic paths ignore the runtime assertion at L_3. Then one additional basic path ends at the runtime assertion:

(7)

```
@$L_2$ : $0 < i < |a| \;\wedge\; 0 \leq j \leq i$
assume $j < i$;
@$L_3$ : $0 \leq j < |a| \;\wedge\; 0 \leq j + 1 < |a|$
```

■

```
@pre $0 \leq \ell \ \wedge\ u < |a| \ \wedge\ \mathsf{sorted}(a, \ell, u)$
@post $rv \ \leftrightarrow\ \exists i.\ \ell \leq i \leq u \ \wedge\ a[i] = e$
bool BinarySearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  if $(\ell > u)$ return false;
  else {
    int $m := (\ell + u)$ div $2$;
    if $(a[m] = e)$ return true;
    else if $(a[m] < e)$ {
      $@R_1 :\ 0 \leq m + 1 \ \wedge\ u < |a| \ \wedge\ \mathsf{sorted}(a, m + 1, u)$;
      return BinarySearch$(a, m + 1, u, e)$;
    } else {
      $@R_2 :\ 0 \leq \ell \ \wedge\ m - 1 < |a| \ \wedge\ \mathsf{sorted}(a, \ell, m - 1)$;
      return BinarySearch$(a, \ell, m - 1, e)$;
    }
  }
}
```

Fig. 5.20. BinarySearch with function call assertions

5.2.2 Basic Paths: Function Calls

Like loops, recursive functions create an unbounded number of paths within programs. But just as loop invariants cut loops to produce a finite number of basic paths, function specifications cut function calls.

Recall that the function postcondition is a relation between the return value *rv* and the formal parameters. A function's postcondition summarizes the effects of calling it. We use these summaries to replace function calls in basic paths.

Remark 5.16. The postconditions of the functions of a program need only include information that is relevant for proving the given specification, so the summaries may be incomplete. Ignoring irrelevant aspects of functions reduces the size of annotations. Chapter 6 discusses techniques for developing function specifications.

The replacement of function calls by function summaries makes the listing of basic paths (and the resulting analysis described in Section 5.2.4) local to functions. Basic paths do not span multiple functions. However, recall that the function postcondition is guaranteed to hold on return only when the function precondition is satisfied on entry. To ensure that the precondition is satisfied, each instance of a function call generates an extra basic path in which the called function's precondition is asserted. An example clarifies this discussion.

Example 5.17. Figure 5.8 lists BinarySearch with its function specification. BinarySearch contains two (recursive) function calls. In Figure 5.20, each function call is protected by a **function call assertion** at R_1 and R_2. Each assertion is constructed by applying a substitution to BinarySearch's precondition

$$F[a, \ell, u, e] : \; 0 \leq \ell \;\wedge\; u < |a| \;\wedge\; \mathsf{sorted}(a, \ell, u) \; .$$

The first function call is $\mathsf{BinarySearch}(a, m + 1, u, e)$, so the function call assertion at R_1 is $F\sigma_1$, where

$$\sigma_1 : \; \{a \mapsto a, \; \ell \mapsto m + 1, \; u \mapsto u, \; e \mapsto e\} \; .$$

The notation of Section 1.5 allows us to write $F[a, m + 1, u, e]$.

The second function call is $\mathsf{BinarySearch}(a, \ell, m - 1, e)$, so the function call assertion at R_2 is $F[a, \ell, m-1, e]$. These assertions are treated in the same way as other assertions, such as runtime assertions. So far, we have the following basic paths:

(1)

```
@pre $0 \leq \ell \;\wedge\; u < |a| \;\wedge\; \mathsf{sorted}(a, \ell, u)$
assume $\ell > u$;
$rv$ := false;
@post $rv \;\leftrightarrow\; \exists i.\; \ell \leq i \leq u \;\wedge\; a[i] = e$
```

(2)

```
@pre $0 \leq \ell \;\wedge\; u < |a| \;\wedge\; \mathsf{sorted}(a, \ell, u)$
assume $\ell \leq u$;
$m$ := $(\ell + u)$ div 2;
assume $a[m] = e$;
$rv$ := true;
@post $rv \;\leftrightarrow\; \exists i.\; \ell \leq i \leq u \;\wedge\; a[i] = e$
```

(3)

```
@pre $0 \leq \ell \;\wedge\; u < |a| \;\wedge\; \mathsf{sorted}(a, \ell, u)$
assume $\ell \leq u$;
$m$ := $(\ell + u)$ div 2;
assume $a[m] \neq e$;
assume $a[m] < e$;
@$R_1$ : $0 \leq m + 1 \;\wedge\; u < |a| \;\wedge\; \mathsf{sorted}(a, m + 1, u)$
```

(5)

```
@pre $0 \leq \ell \;\wedge\; u < |a| \;\wedge\; \mathsf{sorted}(a, \ell, u)$
assume $\ell \leq u$;
$m$ := $(\ell + u)$ div 2;
assume $a[m] \neq e$;
assume $a[m] \geq e$;
@$R_2$ : $0 \leq \ell \;\wedge\; m - 1 < |a| \;\wedge\; \mathsf{sorted}(a, \ell, m - 1)$
```

Because $\mathsf{BinarySearch}$ lacks loops, each basic path starts at the function precondition.

It remains to consider paths **(4)** and **(6)**, which pass through the recursive function calls and end at the postcondition. Paths **(3)** and **(5)** end in the function call assertions at R_1 and R_2 protecting these function calls. Since they assert that the called BinarySearch's precondition holds, we can assume that the returned values obey the postcondition of BinarySearch in each of the calling contexts. Therefore, we can use the function postcondition as a summary of the function call:

(4)

```
@pre $0 \leq \ell \ \wedge\ u < |a| \ \wedge\ \mathsf{sorted}(a, \ell, u)$
assume $\ell \leq u$;
$m := (\ell + u)$ div $2$;
assume $a[m] \neq e$;
assume $a[m] < e$;
assume $v_1 \ \leftrightarrow\ \exists i.\ m + 1 \leq i \leq u \ \wedge\ a[i] = e$;
$rv := v_1$;
@post $rv \ \leftrightarrow\ \exists i.\ \ell \leq i \leq u \ \wedge\ a[i] = e$
```

The lines

```
assume $v_1 \ \leftrightarrow\ \exists i.\ m + 1 \leq i \leq u \ \wedge\ a[i] = e$;
$rv := v_1$;
```

arise as follows. First, translate the `return` statement

```
return BinarySearch$(a, m + 1, u, e)$;
```

into an assignment to rv, as usual:

```
$rv :=$ BinarySearch$(a, m + 1, u, e)$;
```

Next, given that the precondition holds (from path **(3)**), assume that the postcondition holds. Therefore, summarize the function call with a relation based on BinarySearch's postcondition,

$$G[a, \ell, u, e, rv] :\ rv \ \leftrightarrow\ \exists i.\ \ell \leq i \leq u \ \wedge\ a[i] = e\ .$$

Specifically, the relation is $G[a, m + 1, u, e, v_1]$, where v_1 is a fresh variable that captures the return value. In the basic path, assume this relation; then use the return value v_1 in the assignment:

```
assume $G[a, m + 1, u, e, v_1]$;
$rv := v_1$;
```

These are the penultimate lines of **(4)**. Hence, **(4)** replaces the function call BinarySearch$(a, m + 1, u, e)$ with a summary based on the function postcondition. Now reasoning about the basic path does not require reasoning about all of BinarySearch at once.

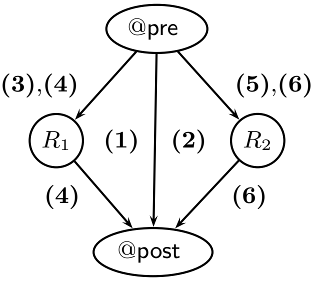

Fig. 5.21. Visualization of basic paths of BinarySearch

Construct the final basic path for the function call BinarySearch$(a, \ell, m - 1, e)$ similarly:

(6)

```
@pre $0 \le \ell \ \wedge\ u < |a| \ \wedge\ \mathsf{sorted}(a, \ell, u)$
assume $\ell \le u$;
$m := (\ell + u)$ div $2$;
assume $a[m] \ne e$;
assume $a[m] \ge e$;
assume $v_2 \ \leftrightarrow\ \exists i.\ \ell \le i \le m - 1 \ \wedge\ a[i] = e$;
$rv := v_2$;
@post $rv \ \leftrightarrow\ \exists i.\ \ell \le i \le u \ \wedge\ a[i] = e$
```

Again, v_2 is a fresh variable.

Figure 5.21 visualizes these basic paths. Paths **(4)** and **(6)** are shown to pass through locations R_1 and R_2, respectively. ■

For the general case, consider function f with prototype

```
@pre $F[p_1, \ldots, p_n]$
@post $G[p_1, \ldots, p_n, rv]$
type$_0$ f(type$_1$ $p_1$, ..., type$_n$ $p_n$)
```

Suppose that f is called in context

$w :=$ f$(e_1, \ldots, e_n)$;

where $e_1, \ldots, e_n$ are expressions. Then augment the calling context with the function call assertion:

```
@ $F[e_1, \ldots, e_n]$;
$w :=$ f$(e_1, \ldots, e_n)$;
```

Treat this new assertion the same as any assertion: it results in at least one basic path ending in

```
...
@ $F[e_1, \ldots, e_n]$
```

Finally, in basic paths that pass through the function call, replace the function call by an assumption and assignment constructed from the postcondition, where v is a fresh variable:

```
...
assume $G[e_1, \ldots, e_n, v]$;
$w := v$;
...
```

Note that rv need not have type `bool` as in BinarySearch. For example, for a function with prototype

```
@pre $\top$
@post $rv \geq x$
int g(int $x$)
```

the statement

```
$w :=$ g($n + 1$);
```

is summarized in basic paths as follows:

```
assume $v \geq n + 1$;
$w := v$;
```

5.2.3 Program States

Before presenting the final step in proving partial correctness, we formalize program state. A program **state** s is an assignment of values of the proper type to all variables. The program variables include a distinguished variable pc, the **program counter**. It holds the current location of control.

Example 5.18. The state

$$s : \ \{pc \mapsto L_1,\ a \mapsto [2;0;1],\ i \mapsto 2,\ j \mapsto 0,\ t \mapsto 2,\ rv \mapsto []\}$$

is a state of BubbleSort in which control resides at L_1. ■

A state can be extended to a logical interpretation. Suppose that T is the theory that captures the functions ($+$, $-$, *etc.*) and predicates ($=$, $<$, *etc.*) of the program. Extend a state s to a logical T-interpretation $I : (D_I, \alpha_I)$: let D_I be all values of the program types; and construct α_I by using the assignments of s and adding assignments for all logical functions and predicates so that I is a T-interpretation. Subsequently, when we say a state, we mean either the assignment of program variables to values or the extension to a T-interpretation for the appropriate theory T, depending on the context. When we write $s \models F$, we mean that $I \models F$ for a T-interpretation I extending s.

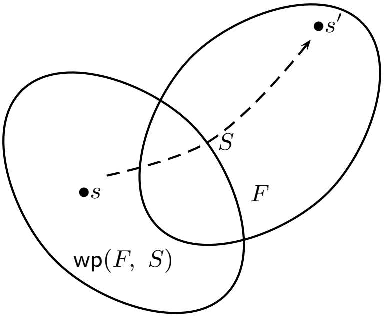

Fig. 5.22. Weakest precondition

Example 5.19. To extend

$$s:\ \{pc \mapsto L_1,\ a \mapsto [2;0;1],\ i \mapsto 2,\ j \mapsto 0,\ t \mapsto 2,\ rv \mapsto []\}$$

to a $(T_{\mathbb{Z}} \cup T_{\mathsf{A}})$-interpretation $I : (D_I, \alpha_I)$, let D_I be the set of all integers and arrays of integers; and for α_I, assign the standard functions to $\cdot[\cdot]$, $\cdot\langle\cdot \triangleleft \cdot\rangle$, $+$, $-$, *etc.* ■

5.2.4 Verification Conditions

Our goal is to reduce an annotated function to a finite set of FOL formulae, called **verification conditions**, such that their validity implies that the function's behavior agrees with its annotations. The reduction to basic paths reduces reasoning about the function to reasoning about a finite set of basic paths. The final reduction from basic paths to VCs requires a mechanism for incorporating the effects of program statements into FOL formulae. The **weakest precondition** predicate transformer is the mechanism. A **predicate transformer** p is a function

$$\mathsf{p}:\ \mathsf{FOL} \times \mathsf{stmts} \to \mathsf{FOL}$$

that maps a FOL formula $F \in \mathsf{FOL}$ and program statement $S \in \mathsf{stmts}$ to a FOL formula.

The weakest precondition $\mathsf{wp}(F,\ S)$ has the defining characteristic that if state s is such that

$$s\ \models\ \mathsf{wp}(F,\ S)$$

and if statement S is executed on state s to produce state s', then

$$s'\ \models\ F\ .$$

This situation is visualized in Figure 12.1**(a)**. The region labeled F is the set of states that satisfy F; similarly, the region labeled $\mathsf{wp}(F,\ S)$ is the set of states that satisfy $\mathsf{wp}(F,\ S)$. Every state s on which executing statement S leads to a state s' in the F region must be in the $\mathsf{wp}(F,\ S)$ region.

Define the weakest precondition for the two statement types of basic paths introduced in Section 5.2.1:

- *Assumption*: What must hold before statement $\texttt{assume}\ c$ is executed to ensure that F holds afterward? If $c \rightarrow F$ holds before, then satisfying c in $\texttt{assume}\ c$ guarantees that F holds afterward:

  $$\mathsf{wp}(F,\ \texttt{assume}\ c)\ \Leftrightarrow\ c \rightarrow F$$

- *Assignment*: What must hold before statement $v := e$ is executed to ensure that $F[v]$ holds afterward? If $F[e]$ holds before, then assigning e to v with $v := e$ makes $F[v]$ hold afterward:

  $$\mathsf{wp}(F[v],\ v := e)\ \Leftrightarrow\ F[e]$$

For a sequence of statements $S_1; \ldots; S_n$, define

$$\mathsf{wp}(F,\ S_1; \ldots; S_n)\ \Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(F,\ S_n),\ S_1; \ldots; S_{n-1})\ .$$

The weakest precondition moves a formula backward over a sequence of statements: for F to hold after executing $S_1; \ldots; S_n$, $\mathsf{wp}(F,\ S_1; \ldots; S_n)$ must hold before executing the statements. Because basic paths have only assumption and assignment statements, the definition of wp is complete.

Then the **verification condition** of basic path

```
@ $F$
$S_1$;
⋮
$S_n$;
@ $G$
```

is

$$F\ \rightarrow\ \mathsf{wp}(G,\ S_1; \ldots; S_n)\ .$$

Its validity implies that when F holds before the statements of the path are executed, then G holds afterward. Traditionally, this verification condition is denoted by the **Hoare triple**

$$\{F\}S_1; \ldots; S_n\{G\}\ .$$

Example 5.20. Consider the basic path

(1)

```
@ $x \geq 0$
$x := x + 1$;
@ $x \geq 1$
```

The VC is

$$\{x \geq 0\} x := x + 1 \{x \geq 1\} : \; x \geq 0 \;\rightarrow\; \mathsf{wp}(x \geq 1,\; x := x + 1)$$

so compute

$$\begin{array}{l} \mathsf{wp}(x \geq 1,\; x := x + 1) \\ \;\Leftrightarrow\; (x \geq 1)\{x \mapsto x + 1\} \\ \;\Leftrightarrow\; x + 1 \geq 1 \\ \;\Leftrightarrow\; x \geq 0 \end{array}$$

Simplifying the VC based on this computation produces

$$x \geq 0 \;\rightarrow\; x \geq 0 \; ,$$

which is clearly $T_{\mathbb{Z}}$-valid. ■

Example 5.21. We generate the VC corresponding to basic path **(2)** in Example 5.13 (LinearSearch):

(2)

$$\begin{array}{l} @L : \; F : \; \ell \leq i \;\wedge\; (\forall j.\ \ell \leq j < i \;\rightarrow\; a[j] \neq e) \\ S_1 : \; \texttt{assume}\ i \leq u; \\ S_2 : \; \texttt{assume}\ a[i] = e; \\ S_3 : \; rv := \texttt{true}; \\ @\mathsf{post}\ G : \; rv \;\leftrightarrow\; \exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e \end{array}$$

The VC is

$$F \;\rightarrow\; \mathsf{wp}(G,\; S_1; S_2; S_3) \; ,$$

so compute

$$\begin{array}{l} \mathsf{wp}(G,\; S_1; S_2; S_3) \\ \;\Leftrightarrow\; \mathsf{wp}(\mathsf{wp}(rv \;\leftrightarrow\; \exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e,\; rv := \texttt{true}),\; S_1; S_2) \\ \;\Leftrightarrow\; \mathsf{wp}(\texttt{true} \;\leftrightarrow\; \exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e,\; S_1; S_2) \\ \;\Leftrightarrow\; \mathsf{wp}(\exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e,\; S_1; S_2) \\ \;\Leftrightarrow\; \mathsf{wp}(\mathsf{wp}(\exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e,\; \texttt{assume}\ a[i] = e),\; S_1) \\ \;\Leftrightarrow\; \mathsf{wp}(a[i] = e \;\rightarrow\; \exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e,\; S_1) \\ \;\Leftrightarrow\; \mathsf{wp}(a[i] = e \;\rightarrow\; \exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e,\; \texttt{assume}\ i \leq u) \\ \;\Leftrightarrow\; i \leq u \;\rightarrow\; (a[i] = e \;\rightarrow\; \exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e) \end{array}$$

Replacing F and $\mathsf{wp}(G,\; S_1; S_2; S_3)$ according to the computed formula results in the VC

$$\begin{array}{l} \ell \leq i \;\wedge\; (\forall j.\ \ell \leq j < i \;\rightarrow\; a[j] \neq e) \\ \quad \rightarrow\; (i \leq u \;\rightarrow\; (a[i] = e \;\rightarrow\; \exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e)) \end{array}$$

or, equivalently,

$$\begin{array}{l} \ell \leq i \;\wedge\; (\forall j.\ \ell \leq j < i \;\rightarrow\; a[j] \neq e) \;\wedge\; i \leq u \;\wedge\; a[i] = e \\ \quad \rightarrow\; \exists j.\ \ell \leq j \leq u \;\wedge\; a[j] = e \end{array}$$

according to the equivalence

$$F_1 \wedge F_2 \;\rightarrow\; (F_3 \rightarrow (F_4 \rightarrow F_5)) \;\Leftrightarrow\; (F_1 \wedge F_2 \wedge F_3 \wedge F_4) \;\rightarrow\; F_5\ .$$

This formula is $(T_{\mathbb{Z}} \cup T_{\mathsf{A}})$-valid. ■

Example 5.22. For basic path **(3)** of Example 5.13 (LinearSearch),

(3)

$@L:\ F:\ \ell \le i \;\wedge\; (\forall j.\ \ell \le j < i \;\rightarrow\; a[j] \neq e)$
$S_1:$ `assume` $i \le u$;
$S_2:$ `assume` $a[i] \neq e$;
$S_3:\ i := i + 1$;
$@L:\ G:\ \ell \le i \;\wedge\; (\forall j.\ \ell \le j < i \;\rightarrow\; a[j] \neq e)$

we use a different strategy to compute the corresponding VC

$$F \;\rightarrow\; \mathsf{wp}(G,\ S_1; S_2; S_3)\ .$$

We collect all modifications to G during applications of wp and then apply them all at once. Compute

$$\begin{array}{l}
\mathsf{wp}(G,\ S_1; S_2; S_3) \\
\ \Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(G,\ i := i + 1),\ S_1; S_2) \\
\ \Leftrightarrow\ \mathsf{wp}(G\{i \mapsto i + 1\},\ S_1; S_2) \\
\ \Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(G\{i \mapsto i + 1\},\ \texttt{assume}\ a[i] \neq e),\ S_1) \\
\ \Leftrightarrow\ \mathsf{wp}(a[i] \neq e \;\rightarrow\; G\{i \mapsto i + 1\},\ S_1) \\
\ \Leftrightarrow\ \mathsf{wp}(a[i] \neq e \;\rightarrow\; G\{i \mapsto i + 1\},\ \texttt{assume}\ i \le u) \\
\ \Leftrightarrow\ i \le u \;\rightarrow\; a[i] \neq e \;\rightarrow\; G\{i \mapsto i + 1\}
\end{array}$$

Then the VC is

$$F \;\rightarrow\; i \le u \;\rightarrow\; a[i] \neq e \;\rightarrow\; G\{i \mapsto i + 1\}\ ,$$

or

$$\begin{array}{l}
\ell \le i \;\wedge\; (\forall j.\ \ell \le j < i \;\rightarrow\; a[j] \neq e) \;\wedge\; i \le u \;\wedge\; a[i] \neq e \\
\quad \rightarrow\; \ell \le i + 1 \;\wedge\; \forall j.\ \ell \le j < i + 1 \;\rightarrow\; a[j] \neq e\ ,
\end{array}$$

which is $(T_{\mathbb{Z}} \cup T_{\mathsf{A}})$-valid. ■

Example 5.23. Consider basic path **(2)** of Example 5.17 (BinarySearch):

(2)

$@\mathsf{pre}\ F:\ 0 \le \ell \;\wedge\; u < |a| \;\wedge\; \mathsf{sorted}(a, \ell, u)$
$S_1:$ `assume` $\ell \le u$;
$S_2:\ m := (\ell + u)$ `div` 2;
$S_3:$ `assume` $a[m] = e$;
$S_4:\ rv :=$ `true`;
$@\mathsf{post}\ G:\ rv \;\leftrightarrow\; \exists i.\ \ell \le i \le u \;\wedge\; a[i] = e$

The VC is

$$F \;\rightarrow\; \mathsf{wp}(G,\ S_1; S_2; S_3; S_4)\ ,$$

so compute

$$\begin{array}{l}
\mathsf{wp}(G,\ S_1; S_2; S_3; S_4) \\
\;\Leftrightarrow\; \mathsf{wp}(\mathsf{wp}(G,\ rv := \texttt{true}),\ S_1; S_2; S_3) \\
\;\Leftrightarrow\; \mathsf{wp}(G\{rv \mapsto \texttt{true}\},\ S_1; S_2; S_3) \\
\;\Leftrightarrow\; \mathsf{wp}(\mathsf{wp}(G\{rv \mapsto \texttt{true}\},\ \texttt{assume}\ a[m] = e),\ S_1; S_2) \\
\;\Leftrightarrow\; \mathsf{wp}(a[m] = e \;\rightarrow\; G\{rv \mapsto \texttt{true}\},\ S_1; S_2) \\
\;\Leftrightarrow\; \mathsf{wp}(\mathsf{wp}(a[m] = e \;\rightarrow\; G\{rv \mapsto \texttt{true}\},\ m := (\ell + u)\ \texttt{div}\ 2),\ S_1) \\
\;\Leftrightarrow\; \mathsf{wp}((a[m] = e \;\rightarrow\; G\{rv \mapsto \texttt{true}\})\{m \mapsto (\ell + u)\ \texttt{div}\ 2\},\ S_1) \\
\;\Leftrightarrow\; \mathsf{wp}((a[m] = e \;\rightarrow\; G\{rv \mapsto \texttt{true}\})\{m \mapsto (\ell + u)\ \texttt{div}\ 2\} \\
\qquad\quad ,\ \texttt{assume}\ \ell \leq u) \\
\;\Leftrightarrow\; \ell \leq u \;\rightarrow\; (a[m] = e \;\rightarrow\; G\{rv \mapsto \texttt{true}\})\{m \mapsto (\ell + u)\ \texttt{div}\ 2\}
\end{array}$$

Applying the substitutions produces

$$\ell \leq u \;\rightarrow\; a[(\ell + u)\ \texttt{div}\ 2] = e \;\rightarrow\; G\{rv \mapsto \texttt{true},\ m \mapsto (\ell + u)\ \texttt{div}\ 2\}\ .$$

Simplifying the VC accordingly

$$\begin{array}{l}
0 \leq \ell \;\wedge\; u < |a| \;\wedge\; \mathsf{sorted}(a, \ell, u) \;\wedge\; \ell \leq u \;\wedge\; a[(\ell + u)\ \texttt{div}\ 2] = e \\
\;\rightarrow\; \exists i.\ \ell \leq i \leq u \;\wedge\; a[i] = e
\end{array}$$

reveals that it is $(T_{\mathbb{Z}} \cup T_{\mathsf{A}})$-valid: $\ell \leq u$ from the antecedent implies that $\ell \leq (\ell + u)\ \texttt{div}\ 2 \leq u$, so that $a[(\ell + u)\ \texttt{div}\ 2] = e$ implies that $\exists i.\ \ell \leq i \leq u \;\wedge\; a[i] = e$. ■

Example 5.24. Consider basic path **(3)** of Example 5.14 (BubbleSort):

(3)

$$@L_2 :\ F : \left[\begin{array}{l}
1 \leq i < |a| \;\wedge\; 0 \leq j \leq i \\
\;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\
\;\wedge\; \mathsf{partitioned}(a, 0, j-1, j, j) \;\wedge\; \mathsf{sorted}(a, i, |a|-1)
\end{array}\right]$$

$S_1 :\ \texttt{assume}\ j < i;$
$S_2 :\ \texttt{assume}\ a[j] > a[j+1];$
$S_3 :\ t := a[j];$
$S_4 :\ a[j] := a[j+1];$
$S_5 :\ a[j+1] := t;$
$S_6 :\ j := j + 1;$

$$@L_2 :\ G : \left[\begin{array}{l}
1 \leq i < |a| \;\wedge\; 0 \leq j \leq i \\
\;\wedge\; \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\
\;\wedge\; \mathsf{partitioned}(a, 0, j-1, j, j) \;\wedge\; \mathsf{sorted}(a, i, |a|-1)
\end{array}\right]$$

The VC is

$$F \;\rightarrow\; \mathsf{wp}(G,\ S_1; S_2; S_3; S_4; S_5; S_6)$$

so compute

$$
\begin{array}{l}
\mathsf{wp}(G,\ S_1; S_2; S_3; S_4; S_5; S_6) \\
\ \Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(G,\ j := j+1),\ S_1; S_2; S_3; S_4; S_5) \\
\ \Leftrightarrow\ \mathsf{wp}(G\{j \mapsto j+1\},\ S_1; S_2; S_3; S_4; S_5) \\
\ \Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(G\{j \mapsto j+1\},\ a[j+1] := t),\ S_1; S_2; S_3; S_4) \\
\ \Leftrightarrow\ \mathsf{wp}(G\{j \mapsto j+1\}\{a \mapsto a\langle j+1 \triangleleft t\rangle\},\ S_1; S_2; S_3; S_4)
\end{array}
$$

Array assignment $a[j+1] := t$ translates to the functional substitution $\{a \mapsto a\langle j+1 \triangleleft t\rangle\}$ according to the theory of arrays, T_{A}. That is, the assignment is modeled by substituting $a\langle j+1 \triangleleft t\rangle$ for every instance a affected by the assignment. Continuing,

$$
\begin{array}{l}
\Leftrightarrow\ \mathsf{wp}(G\{j \mapsto j+1,\ a \mapsto a\langle j+1 \triangleleft t\rangle\},\ S_1; S_2; S_3; S_4) \\
\Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(G\{j \mapsto j+1,\ a \mapsto a\langle j+1 \triangleleft t\rangle\},\ a[j] := a[j+1]) \\
\qquad ,\ S_1; S_2; S_3) \\
\Leftrightarrow\ \mathsf{wp}(G\{j \mapsto j+1,\ a \mapsto a\langle j+1 \triangleleft t\rangle\}\{a \mapsto a\langle j \triangleleft a[j+1]\rangle\} \\
\qquad ,\ S_1; S_2; S_3) \\
\Leftrightarrow\ \mathsf{wp}(G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft t\rangle\},\ S_1; S_2; S_3)
\end{array}
$$

The array assignment S_4 similarly translates to a substitution. Composing the two substitutions produces the substitution in which the doubly-modified array $a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft t\rangle$ replaces a. Then

$$
\begin{array}{l}
\Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft t\rangle\},\ t := a[j]) \\
\qquad ,\ S_1; S_2) \\
\Leftrightarrow\ \mathsf{wp}(G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft t\rangle\}\{t \mapsto a[j]\} \\
\qquad ,\ S_1; S_2) \\
\Leftrightarrow\ \mathsf{wp}(G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle,\ t \mapsto a[j]\} \\
\qquad ,\ S_1; S_2)
\end{array}
$$

Here, the composition of substitutions results in applying $\{t \mapsto a[j]\}$ to the term $a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft t\rangle$, which contains t. To finish,

$$
\begin{array}{l}
\Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle,\ t \mapsto a[j]\} \\
\qquad\quad ,\ \texttt{assume}\ a[j] > a[j+1]) \\
\qquad ,\ S_1) \\
\Leftrightarrow\ \mathsf{wp}(a[j] > a[j+1] \\
\qquad\quad \rightarrow\ G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle,\ t \mapsto a[j]\} \\
\qquad ,\ S_1) \\
\Leftrightarrow\ \mathsf{wp}(a[j] > a[j+1] \\
\qquad\quad \rightarrow\ G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle,\ t \mapsto a[j]\} \\
\qquad ,\ \texttt{assume}\ j < i) \\
\Leftrightarrow\ j < i\ \rightarrow\ a[j] > a[j+1] \\
\qquad \rightarrow\ G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle,\ t \mapsto a[j]\} \\
\Leftrightarrow\ j < i\ \wedge\ a[j] > a[j+1] \\
\qquad \rightarrow\ G\{j \mapsto j+1,\ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle,\ t \mapsto a[j]\}
\end{array}
$$

Finally, applying the substitution

$$\sigma : \ \{j \mapsto j+1, \ a \mapsto a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, \ t \mapsto a[j]\}$$

to G produces $G\sigma$:

$$\begin{array}{l}1 \leq i < |a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle| \ \land \ 0 \leq j+1 \leq i \\ \ \land \ \mathsf{partitioned}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, 0, i, i+1, \\ \qquad\qquad |a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle| - 1) \\ \ \land \ \mathsf{partitioned}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, 0, j, j+1, j+1) \\ \ \land \ \mathsf{sorted}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, i, |a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle| - 1)\end{array}$$

The **length function** $|\cdot|$ obeys the following axiom:

$$\forall a, i, e. \ |a\langle i \triangleleft e\rangle| = |a| \qquad \text{(array length)}$$

In other words, modifying an array's elements does not change its size. Under this axiom, $G\sigma$ is equivalent to

$$\begin{array}{l}1 \leq i < |a| \ \land \ 0 \leq j+1 \leq i \\ \ \land \ \mathsf{partitioned}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, 0, i, i+1, |a| - 1) \\ \ \land \ \mathsf{partitioned}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, 0, j, j+1, j+1) \\ \ \land \ \mathsf{sorted}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, i, |a| - 1)\end{array}$$

Thus, the VC is

$$\begin{array}{l}\left[\begin{array}{l}1 \leq i < |a| \ \land \ 0 \leq j \leq i \\ \ \land \ \mathsf{partitioned}(a, 0, i, i+1, |a| - 1) \\ \ \land \ \mathsf{partitioned}(a, 0, j-1, j, j) \\ \ \land \ \mathsf{sorted}(a, i, |a| - 1) \\ \ \land \ j < i \ \land \ a[j] > a[j+1]\end{array}\right] \\ \rightarrow \left[\begin{array}{l}1 \leq i < |a| \ \land \ 0 \leq j+1 \leq i \\ \ \land \ \mathsf{partitioned}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, 0, i, i+1, |a| - 1) \\ \ \land \ \mathsf{partitioned}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, 0, j, j+1, j+1) \\ \ \land \ \mathsf{sorted}(a\langle j \triangleleft a[j+1]\rangle\langle j+1 \triangleleft a[j]\rangle, i, |a| - 1)\end{array}\right]\end{array}$$

which is $(T_{\mathbb{Z}} \cup T_{\mathsf{A}})$-valid. ■

5.2.5 *P*-Invariant and *P*-Inductive

Let us consider the inductive assertion method abstractly.

A **program** is a set of functions with some distinguished **entry function**. Let P be a program with distinguished entry function f such that f has function precondition F_{pre} and initial location L_0. A **computation** of program P, or a P-computation, is a sequence of states

$$s_0, s_1, s_2, \ldots$$

such that

1. $s_0[pc] = L_0$ and $s_0 \models F_{\mathsf{pre}}$;
2. and for each index i, s_{i+1} is a result of executing the instruction at $s_i[pc]$ on state s_i.

The notation $s_i[pc]$ indicates the value of pc given by state s_i. A computation may be finite or infinite. Infinite computations arise when a program contains a function that does not always halt (either through recursion or looping).

A formula F annotating location L of program P is **P-invariant** (also, is an invariant of P, or is a P-invariant) iff for all P-computations $s_0, s_1, s_2, \ldots$ and for each index i, if $s_i[pc] = L$ then $s_i \models F$. The annotations of P are P-invariant iff each annotation is P-invariant. This definition is not implementable: checking if F is P-invariant requires checking an infinite number of P-computations in general, even if all computations are finite.

Instead, we use the inductive assertion method introduced in this chapter. If all verification conditions generated from program P are T-valid (in the proper theory T), then the annotations are **P-inductive** (also, are inductive P-invariants) and therefore P-invariant. In summary, we have the following theorem.

Theorem 5.25 (Verification Conditions). *If for every basic path*

```
$@L_i : \ F$
$S_1;$
$\vdots$
$S_n;$
$@L_j : \ G$
```

of program P, the verification condition

$$\{F\}S_1; \ldots; S_n\{G\}$$

is valid (in the appropriate theory), then the annotations are P-invariant. In particular, the annotations are P-inductive.

In other words, when P's annotations are P-inductive, each function of P obeys its specification.

Henceforth, when P is obvious from the context, we say invariant instead of P-invariant and inductive instead of P-inductive.

5.3 Total Correctness

Partial correctness is just one step in proving the **total correctness** of a function or program. Total correctness of a function asserts that if the input satisfies the function precondition, then the function *eventually halts* and produces output that satisfies its function postcondition. In other words, total correctness requires proving partial correctness and that the function always halts on input satisfying its precondition. We focus now on the latter task.

Proving function termination is based on well-founded relations (see Chapter 4). Define a set S with a well-founded relation $\prec$. Then find a function δ mapping program states to S such that δ decreases according to $\prec$ along every basic path. Since $\prec$ is well-founded, there cannot exist an infinite sequence of program states; otherwise, they would map to an infinite decreasing sequence in S. The function δ is called a **ranking function**.

Choosing Well-Founded Relations and Ranking Functions

The concept of well-founded relations is the fundamental principle of this method. The ranking function itself is really just a convenience. One could directly construct a well-founded relation over the program states and show that the output state of each basic path is less than the input state according to this relation. However, it is often conceptually easier to find a function that maps states to a known set S with a known well-founded relation $\prec$. For example, we usually choose as S the set of n-tuples of natural numbers and as $\prec$ the lexicographic extension $<_n$ of $<$ to n-tuples of natural numbers, where n varies according to the application.

As with partial correctness, we consider total correctness in the context of loops first and then in the context of recursion. Section 6.2 examines a program with both loops and recursion.

Example 5.26. Figure 5.23 lists BubbleSort with ranking annotations. It contains one new type of annotation: $\downarrow\ (i+1, i+1)$ and $\downarrow\ (i+1, i-j)$ assert that the functions $(i+1, i+1)$ and $(i+1, i-j)$, respectively, are ranking functions. These functions map states of BubbleSort onto pairs of natural numbers $S : \mathbb{N}^2$ with well-founded relation $<_2$. Intuitively, we have captured two separate arguments. The outer loop eventually finishes because i decreases to 0; hence $i+1$ decreases as well. Why do we use $i+1$ rather than i? When $|a| = 0$, the initial assignment to i is -1. While $i+1$ is always nonnegative, i is not; and recall that we want to map into the natural numbers. The inner loop halts because j increases to i; hence $i-j$ decreases to 0. Therefore, our intuition tells us that $i+1$ is important for the outer loop, and $i-j$ is important for the inner loop.

Placing these two functions $i+1$ and $i-j$ together as a pair $(i+1, i-j)$ provides the annotation for the inner loop. We expect $i+1$ to remain constant while the inner loop is executing and decreasing $i-j$. For the annotation of the outer loop, we note that $i+1 > i-j = i-0 = i$ on entry to the inner loop, so that $(i+1, i+1) >_2 (i+1, i-j)$.

The loop annotations assert that the ranking functions $(i+1, i+1)$ and $(i+1, i-j)$ map program states to pairs of natural numbers. Hence, we need to prove that the loop annotations are inductive using the inductive assertion

```
@pre $\top$
@post $\top$
int[] BubbleSort(int[] $a_0$) {
  int[] $a := a_0$;
  for
    @$L_1 :\ i + 1 \geq 0$
    $\downarrow (i + 1,\ i + 1)$
    (int $i := |a| - 1;\ i > 0;\ i := i - 1$) {
    for
      @$L_2 :\ i + 1 \geq 0 \ \wedge\ i - j \geq 0$
      $\downarrow (i + 1,\ i - j)$
      (int $j := 0;\ j < i;\ j := j + 1$) {
      if ($a[j] > a[j + 1]$) {
        int $t := a[j]$;
        $a[j] := a[j + 1]$;
        $a[j + 1] := t$;
      }
    }
  }
  return $a$;
}
```

Fig. 5.23. BubbleSort with annotations to prove termination

method. We leave this step to the reader. It only remains to prove that the functions decrease along each basic path.

The relevant basic paths are the following:

(1)

```
@$L_1 :\ i + 1 \geq 0$
$\downarrow L_1 :\ (i + 1, i + 1)$
assume $i > 0$;
$j := 0$;
$\downarrow L_2 :\ (i + 1, i - j)$
```

(2)

```
@$L_2 :\ i + 1 \geq 0 \ \wedge\ i - j \geq 0$
$\downarrow L_2 :\ (i + 1, i - j)$
assume $j < i$;
assume $a[j] > a[j + 1]$;
$t := a[j]$;
$a[j] := a[j + 1]$;
$a[j + 1] := t$;
$j := j + 1$;
$\downarrow L_2 :\ (i + 1, i - j)$
```

(3)

```
$@L_2 :\ i + 1 \geq 0\ \wedge\ i - j \geq 0$
$\downarrow L_2 :\ (i + 1, i - j)$
assume $j < i$;
assume $a[j] \leq a[j + 1]$;
$j := j + 1$;
$\downarrow L_2 :\ (i + 1, i - j)$
```

(4)

```
$@L_2 :\ i + 1 \geq 0\ \wedge\ i - j \geq 0$
$\downarrow L_2 :\ (i + 1, i - j)$
assume $j \geq i$;
$i := i - 1$;
$\downarrow L_1 :\ (i + 1, i + 1)$
```

The paths entering and exiting the outer loop at L_1 are not relevant for the termination argument. The entering path does not begin with a ranking function annotation, so there is nothing to prove. The exiting path leads to the `return` statement.

For termination purposes, paths **(2)** and **(3)** can be treated the same:

```
$@L_2 :\ i + 1 \geq 0\ \wedge\ i - j \geq 0$
$\downarrow L_2 :\ (i + 1, i - j)$
assume $j < i$;
...
$j := j + 1$;
$\downarrow L_2 :\ (i + 1, i - j)$
```

The excluded statements do not impact the value of the ranking functions. ■

Verification Conditions

A basic path beginning and ending with a ranking function

```
$@\ F$
$\downarrow\ \delta[\overline{x}]$
$S_1$;
⋮
$S_k$;
$\downarrow\ \kappa[\overline{x}]$
```

induces a verification condition

$$F \;\rightarrow\; \mathsf{wp}(\kappa \prec \delta[\overline{x}_0],\; S_1; \cdots ; S_k)\{\overline{x}_0 \mapsto \overline{x}\} \;.$$

In words, we must prove that the value of κ after executing statements $S_1; \cdots ; S_k$ is less than the value of δ before executing the statements. Therefore, we rename the variables of δ to something new ($\overline{x}_0$ in this case), compute the weakest precondition of $\kappa \prec \delta[\overline{x}_0]$ across the statements $S_1; \cdots ; S_k$, and then rename the new variables back to their original names ($\overline{x}$ in this case). This renaming process preserves the value of $\delta[\overline{x}]$ in its context. The annotation F can provide extra invariant information with which to prove this relation.

Example 5.27. Let us return to the proof of Example 5.26 that BubbleSort halts. Path **(1)** induces the following verification condition:

$$i+1 \geq 0 \;\wedge\; i > 0 \;\rightarrow\; (i+1, i-0) <_2 (i+1, i+1) \;,$$

which is valid. Paths **(2)** and **(3)** induce the verification condition:

$$i+1 \geq 0 \;\wedge\; i-j \geq 0 \;\wedge\; j < i \;\rightarrow\; (i+1, i-(j+1)) <_2 (i+1, i-j) \;,$$

which is valid. Finally, **(4)** induces the verification condition

$$i+1 \geq 0 \;\wedge\; i-j \geq 0 \;\wedge\; j \geq i \;\rightarrow\; ((i-1)+1, (i-1)+1) <_2 (i+1, i-j) \;,$$

which is also valid. Hence, BubbleSort always halts. Combined with the proof of the sortedness property, we can now say that BubbleSort is totally correct with respect to its specification: it always halts and returns a sorted array.

Let us work through the construction of the final verification condition for basic path **(4)**. First, replace i and j with i_0 and j_0, respectively, in the function annotating L_2: (i_0+1, i_0-j_0). Then compute

$$\begin{array}{l}\mathsf{wp}((i+1, i+1) <_2 (i_0+1, i_0-j_0),\; \texttt{assume}\ j \geq i;\ i := i-1) \\ \quad \Leftrightarrow\; \mathsf{wp}(((i-1)+1, (i-1)+1) <_2 (i_0+1, i_0-j_0),\; \texttt{assume}\ j \geq i) \\ \quad \Leftrightarrow\; j \geq i \;\rightarrow\; (i, i) <_2 (i_0+1, i_0-j_0)\end{array}$$

Now, having preserved the original value of $(i+1, i-j)$ at L_2, replace i_0 and j_0 with i and j, respectively:

$$j \geq i \;\rightarrow\; (i, i) <_2 (i+1, i-j) \;.$$

Noting that **(4)** begins by asserting $i+1 \geq 0 \;\wedge\; i-j \geq 0$, we have our verification condition:

$$i+1 \geq 0 \;\wedge\; i-j \geq 0 \;\wedge\; j \geq i \;\rightarrow\; (i, i) <_2 (i+1, i-j) \;.$$

In this proof, the loop annotations (other than the ranking functions) do not have any bearing on the termination argument. Their purpose is only to prove that the given functions map to the natural numbers. ■

```
@pre $u - \ell + 1 \geq 0$
@post $\top$
$\downarrow u - \ell + 1$
bool BinarySearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  if ($\ell > u$) return false;
  else {
    int $m := (\ell + u)$ div 2;
    if ($a[m] = e$) return true;
    else if ($a[m] < e$) return BinarySearch($a, m + 1, u, e$);
    else return BinarySearch($a, \ell, m - 1, e$);
  }
}
```

Fig. 5.24. BinarySearch always halts

Besides loops, recursion is another source of nontermination and hence requires ranking annotations as well. Again, we break the termination argument down to the level of basic paths.

Example 5.28. Figure 5.24 lists the BinarySearch function. $u - \ell + 1$ maps the formal parameters of BinarySearch to $S : \mathbb{N}$ with well-founded relation $<$. Why do we use $u - \ell + 1$ as the ranking function? Intuitively, the interval $[\ell, u]$ shortens at each level of recursion, so $u - \ell$ is a good guess at a ranking function. However, it may be that $\ell > u$ so that $u - \ell$ does not map into $\mathbb{N}$; but in this case $\ell = u + 1$, so $u - \ell + 1$ does map into $\mathbb{N}$.

We now need to prove two properties of $u - \ell + 1$:

1. Because $u - \ell + 1$ has type int, we must prove that $u - \ell + 1$ in fact maps to $\mathbb{N}$: whenever $u - \ell + 1$ is evaluated at function entry, it must be the case that $u - \ell + 1 \geq 0$.
2. We must prove that $u - \ell + 1$ decreases from function entry to each recursive call.

The function precondition asserts the first property, so we need only prove that the function specification is inductive as usual. To prove the second property, we reduce the argument to basic paths: $u - \ell + 1$ should decrease across each basic path.

Convince yourself that the annotations other than the ranking argument are inductive.

We now consider the ranking argument. The relevant basic paths of the function are the following:

(1)

```
@pre $u - \ell + 1 \geq 0$
$\downarrow\ u - \ell + 1$
assume $\ell \leq u$;
$m$ := $(\ell + u)$ div 2;
assume $a[m] \neq e$;
assume $a[m] < e$;
$\downarrow\ u - (m + 1) + 1$
```

(2)

```
@pre $u - \ell + 1 \geq 0$
$\downarrow\ u - \ell + 1$
assume $\ell \leq u$;
$m$ := $(\ell + u)$ div 2;
assume $a[m] \neq e$;
assume $a[m] \geq e$;
$\downarrow\ (m - 1) - \ell + 1$
```

Two other basic paths exist from function entry to the first two `return` statements; however, as the recursion ends at each, they are irrelevant to the termination argument.

The basic paths induce two verification conditions. Before examining them, notice that the `assume` statements about $a[m]$ are irrelevant to the termination argument. Now, the first VC is

$$\begin{array}{l} u - \ell + 1 \geq 0 \ \wedge \ \ell \leq u \ \wedge \ \cdots \\ \quad \rightarrow \ u - (((\ell + u) \ \texttt{div} \ 2) + 1) + 1 < u - \ell + 1 \ , \end{array}$$

where $\cdots$ elides the literals involving $a[m]$. It is $T_{\mathbb{Z}}$-valid. The VC

$$\begin{array}{l} u - \ell + 1 \geq 0 \ \wedge \ \ell \leq u \ \wedge \ \cdots \\ \quad \rightarrow \ (((\ell + u) \ \texttt{div} \ 2) - 1) - \ell + 1 < u - \ell + 1 \end{array}$$

for the second basic path is also $T_{\mathbb{Z}}$-valid, so BinarySearch halts on all input in which ℓ is initially at most $u + 1$.

Section 6.2 provides an alternative to using the awkward ranking function $u - \ell + 1$. Additionally, the argument proves termination on all input. ■

5.4 Summary

This chapter introduces the specification and verification of sequential programs. It covers:

- The programming language pi.

- *Program specification.* Specifying a program involves writing *function preconditions* and *function postconditions* as first-order assertions. A function precondition asserts on which inputs the function is defined, while a function postcondition asserts the form of the returned data. Other annotations: loop invariants, assertions, runtime assertions.
- *Partial correctness*, which guarantees that if a function halts and the input satisfied the function precondition, then its returned value satisfies the function postcondition. Partial correctness is proved via an inductive argument. Additional annotations strengthen the inductive hypothesis. Basic paths, program state, verification conditions, inductive invariants.
- *Total correctness*, which guarantees additionally that the program or function always halts. Total correctness requires mapping, via a ranking function, program states to a domain with a well-founded relation and proving that the mapped values in the domain decrease as computation progresses. Typically, proving termination requires additional partial correctness annotations.

Chapter 6 discusses strategies for applying the techniques of this chapter.

The methods introduced in this chapter are fundamental to verification of software and hardware. However, we present them in a simple context. Decades of research have made much more possible.

The validity of verification conditions listed in examples or produced by programs in the chapter are all decided using the decision procedures discussed in Part II. We have focused on specifications that can be expressed in the fragments of theories introduced in Chapter 3. More complex specifications may require general mechanical theorem proving. However, mechanical theorem provers often rely on the decision procedures of Part II when possible for speed and to minimize human interaction.

Chapter 12 introduces algorithms for deducing annotations.

Bibliographic Remarks

Formally proving program correctness has been a subject of active research for five decades. McCarthy argues in [59, 58] for a “mathematical science of computation”. Floyd [34] and Hoare [39] introduce the main concepts for proving property invariance and termination. In particular, they develop *Floyd-Hoare logic.* Manna describes a verification style similar to ours [52]. The *weakest precondition* predicate transformer was first formalized by Dijkstra [28].

King describes in his thesis [50] the idea of a *verifying compiler*, which generates and proves during compilation the verification conditions that arise from program annotations. See [27] for a discussion of the *Extended Static Checker*, a verifying compiler for Java.

```
@pre $p(a_0)$
@post sorted$(rv, 0, |rv| - 1)$
int[] InsertionSort(int[] $a_0$) {
  int[] $a := a_0$;
  for
    @ $r_1(a, a_0, i, j)$
    (int $i := 1$; $i < |a|$; $i := i + 1$) {
    int $t := a[i]$;
    for
      @ $r_2(a, a_0, i, j)$
      (int $j := i - 1$; $j \geq 0$; $j := j - 1$) {
      if $(a[j] \leq t)$ break;
      $a[j + 1] := a[j]$;
    }
    $a[j + 1] := t$;
  }
  return $a$;
}
```

Fig. 5.25. InsertionSort for Exercise 5.1**(a)**

Exercises

5.1 (Basic paths). For each of the following functions, replace each @pre $\top$ with a fresh predicate p over the function parameters, each @post $\top$ with a fresh predicate q over rv and the function parameters, and each @ $\top$ with a fresh predicate r over the function variables. As an example, see Figure 5.25 for the replacements for part **(a)**. Then list the basic paths.

(a) InsertionSort of Figure 6.8.
(b) merge of Figure 6.9.
(c) ms of Figure 6.9.

5.2 (Weakest precondition). Compute the following formulae:

(a) wp$(x \geq 0,\ x := x - k;\ \texttt{assume}\ k \leq 1)$
(b) wp$(x \geq 0,\ \texttt{assume}\ k \leq x;\ x := x - k)$
(c) wp$(x \geq 0,\ x := x - k;\ \texttt{assume}\ k \leq x)$
(d) wp$(x + 2y \geq 3,\ x := x + 1;\ \texttt{assume}\ x > 0;\ y := y + x)$

5.3 (Verification condition generation). Generate the VCs for the following basic paths:

(1)

```
@ $x > 0$;
$x := x - k$;
assume $k \leq 1$;
@ $x \geq 0$;
```

(2)

```
@ $\top$;
assume $k \leq x$;
$x := x - k$;
@ $x \geq 0$;
```

(3)

```
@ $\top$;
$x := x - k$;
assume $k \leq x$;
@ $x \geq 0$;
```

(4)

```
@ $k \geq 0$;
$x := x - k$;
assume $k \leq x$;
@ $x \geq 0$;
```

(5)

```
@ $y \geq 0$;
$x := x + 1$;
assume $x > 0$;
$y := y + x$;
@ $x + 2y \geq 3$;
```

Which are $T_{\mathbb{Z}}$-valid? Which are $T_{\mathbb{Q}}$-valid?

5.4 (Verification conditions). For each basic path generated in Exercise 5.1, list the corresponding VCs.

5.5 (Public functions). Example 5.6 asserts that a function that is accessible outside a module should have a reasonable function precondition, such as $\top$. Implement verified public wrapper functions to LinearSearch and BinarySearch for searching an entire array. The function preconditions should be reasonable.

5.6 (The `div` function). Integer division, even by a constant, is not a function of $T_{\mathbb{Z}}$; however, it is useful for reasoning about programs like BinarySearch. Show how basic paths that include the `div` function can be altered to use only standard linear arithmetic. *Hint*: Use an additional `assume` statement. How does this change affect the resulting VCs?

5.7 (Ackermann function). Implement the *ack* function of Example 4.9 as a pi program and prove that it always halts.

6

Program Correctness: Strategies

> *The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program.*
>
> — Robert W. Floyd
>
> *Assigning Meanings to Programs*, 1967

As in other applications of mathematical induction (see Example 4.1), the main challenge in applying the inductive assertion method is discovering extra information to make the inductive argument succeed. Consider a typical partial correctness property that asserts that a function's output satisfies some relation with its input. Assuming this property as the inductive hypothesis does not provide any information about how the function behaves between entry and exit. It is a weak hypothesis. Therefore, one must provide more information about the function in the form of additional program annotations. Section 6.1 discusses strategies for discovering this additional information. Section 6.2 applies the strategies to prove that QuickSort always returns a sorted array.

6.1 Developing Inductive Annotations

The machinery presented in Section 5.2 automatically reduces an annotated function to a finite set of verification conditions. Furthermore, the decision procedures discussed in Part II of this text make deciding the validity of many verification conditions an automatable task. For example, all verification conditions in this text fall within decidable fragments of FOL, unless otherwise noted. Thus, determining if a program's annotations are inductive can be automated in many cases. Ongoing research in decision procedures continually expands the set of annotations that produce decidable verification conditions.

Developing annotations is a different matter. As expected, writing a function specification requires human ingenuity, just as implementing the function requires it. Of course, simple assertions, such as that a program is free of runtime errors, can be generated automatically.

Writing the loop invariants also requires human ingenuity. A certain level of human intervention is acceptable: the programmer ought to know certain facts about her/his code. Loop invariants often capture insights into how the code works and what it accomplishes. Developing the implementation and annotations simultaneously results in more robust systems. Finally, annotations formally document code, facilitating better development in team projects.

However, Section 5.2.5 points out a fundamental limitation of the inductive assertion method of program verification: loop invariants must be inductive for the corresponding verification conditions to be valid, not just invariant. Consequently, the programmer can assert many facts that are indeed invariant; yet if the annotations are not inductive, the facts cannot be proved.

Much research addresses automatic (inductive) invariant discovery. For example, algorithms exist for discovering linear and polynomial relations among integer and real variables. Such invariants can, for example, provide loop index bounds, prove the lack of division by 0, or prove that an index into an array is within bounds. Other methods exist for discovering the "shape" of memory in programming languages with pointers, allowing, for example, the partially automated analysis of linked lists. One of the most important roles of automatic invariant discovery is strengthening the programmer's annotations into inductive annotations. Chapter 12 introduces invariant generation procedures. However, no set of algorithms will ever fully replace humans in writing verified software.

In this section, we suggest structured techniques for developing inductive annotations to prove partial correctness. We emphasize that the methods are just heuristics: human ingenuity is still the most important ingredient in forming proofs.

6.1.1 Basic Facts

To begin a proof, include basic facts in loop invariants. Basic facts include loop index ranges and other "obvious" facts. To be inductive, complex assertions usually require these basic facts. We illustrate the development of basic facts through several examples.

Example 6.1. Consider the loop of LinearSearch(see also Figure 5.1):

```
for
  $@L :\ \top$
  (int $i := \ell;\ i \leq u;\ i := i + 1$) {
  if ($a[i] = e$) return true;
}
```

Based on the initialization of i, the loop guard, and that i is only modified by being incremented in the loop update, we know that at L,

$$\ell \leq i \leq u + 1\ .$$

Notice the upper bound. It is a common mistake to forget that on the final iteration, the loop guard is not true. Our basic annotation of the loop is the following:

```
for
  @$L$ :  $\ell \leq i \leq u + 1$
  (int $i := \ell$;  $i \leq u$;  $i := i + 1$) {
  if ($a[i] = e$) return true;
}
```

■

Example 6.2. Consider the loops of BubbleSort (see also Figure 5.3):

```
for
  @$L_1$ :  $\top$
  (int $i := |a| - 1$;  $i > 0$;  $i := i - 1$) {
  for
    @$L_2$ :  $\top$
    (int $j := 0$;  $j < i$;  $j := j + 1$) {
    if ($a[j] > a[j + 1]$)  {
      int $t := a[j]$;
      $a[j] := a[j + 1]$;
      $a[j + 1] := t$;
    }
  }
}
```

The outer loop index i ranges according to

$$-1 \leq i < |a|$$

Why -1? If $|a| = 0$, then $|a| - 1 = -1$ so that i is initially -1. Keep in mind that "corner cases" like this one are just as important as normal cases (and perhaps even more important when considering correctness: corner cases are often the source of bugs). In the inner loop, the range of i is more restricted:

$$0 < i < |a|$$

because of the outer loop guard.

In the inner loop, j ranges according to

$$0 \leq j \leq i\ .$$

Therefore, our basic annotation of the two loops is the following:

```
for
  $@L_1 : \; -1 \le i < |a|$
  (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
  for
    $@L_2 : \; 0 < i < |a| \; \land \; 0 \le j \le i$
    (int $j := 0$; $j < i$; $j := j + 1$) {
    if ($a[j] > a[j + 1]$) {
      int $t := a[j]$;
      $a[j] := a[j + 1]$;
      $a[j + 1] := t$;
    }
  }
}
```

Note that the loops modify just the elements of a, not a itself. Therefore, we could add the annotation

$$|a| = |a_0|$$

to both loops. Such an annotation would be useful if the postcondition asserted, for example, that

$$|rv| = |a_0| \; .$$

For the property that we address ($\mathsf{sorted}(rv, 0, |rv| - 1)$), this annotation is not useful. ■

6.1.2 The Precondition Method

Basic facts provide a foundation for more interesting information. The **precondition method** (also called the "backward substitution" or "backward propagation" method) is a strategy for developing more interesting information in a structured way. Again, we emphasize that the method is a heuristic, not an algorithm: it provides some guidance for the human rather than replacing the human's intuition and ingenuity.

The precondition method consists of the following steps:

1. Identify a fact F that is known at one location L in the function ($@L : \; F$) but that is not supported by annotations earlier in the function.
2. Repeat:
   a) Compute the weakest preconditions of F backward through the function, ending at loop invariants or at the beginning of the function.
   b) At each new annotation location L', generalize the new facts to new formula F' ($@L' : \; F'$).

We illustrate the technique through examples.

Example 6.3. Consider the loop of LinearSearch (see also Figure 5.1), annotated with basic facts:

```
for
  @$L$ : $\ell \le i \le u + 1$
  (int $i := \ell$; $i \le u$; $i := i + 1$) {
  if ($a[i] = e$) return true;
}
return false;
```

The postcondition of LinearSearch is

$$rv \;\leftrightarrow\; \exists i.\ \ell \le i \le u \;\wedge\; a[i] = e\ .$$

Consider basic path **(4)** of Example 5.13 but with the current loop invariant substituted for the first assertion:

(4)

@L : F_1 : $\ell \le i \le u + 1$
S_1 : assume $i > u$;
S_2 : rv := false;
@post F_2 : $rv \;\leftrightarrow\; \exists j.\ \ell \le j \le u \;\wedge\; a[j] = e$

Note that we continue to number basic paths as they were numbered in Example 5.13. The VC

$$\{F_1\}S_1;S_2\{F_2\} :\ \ell \le i \le u + 1 \;\wedge\; i > u \;\rightarrow\; \neg(\exists j.\ \ell \le j \le u \;\wedge\; a[j] = e)$$

is not $(T_{\mathbb{Z}} \cup T_{\mathsf{A}})$-valid. Essentially, the antecedent does not assert anything useful about the content of a. Write the consequent as

$$F :\ \forall j.\ \ell \le j \le u \;\rightarrow\; a[j] \ne e$$

by pushing in the negation. F says that if LinearSearch exits via S_2, then no element of a in the range $[\ell, u]$ is e. But F is not supported by the current loop invariant at L. In short, F_2 is a fact that is not supported by earlier annotations.

Having identified an unsupported fact, we compute preconditions. To propagate F_2 back to the loop invariant, compute

$$\begin{array}{l} \mathsf{wp}(F_2,\ S_1;S_2) \\ \;\Leftrightarrow\; \mathsf{wp}(\mathsf{wp}(F_2,\ rv := \texttt{false}),\ S_1) \\ \;\Leftrightarrow\; \mathsf{wp}(F_2\{rv \mapsto \texttt{false}\},\ S_1) \\ \;\Leftrightarrow\; \mathsf{wp}(F_2\{rv \mapsto \texttt{false}\},\ \texttt{assume}\ i > u) \\ \;\Leftrightarrow\; i > u \;\rightarrow\; F_2\{rv \mapsto \texttt{false}\} \\ \;\Leftrightarrow\; i > u \;\rightarrow\; \forall j.\ \ell \le j \le u \;\rightarrow\; a[j] \ne e \end{array}$$

The final formula

$$G : \; i > u \;\rightarrow\; \forall j.\; \ell \leq j \leq u \;\rightarrow\; a[j] \neq e \;,$$

in particular the antecedent $i > u$, and some intuition suggests the generalization

$$G' : \; \forall j.\; \ell \leq j < i \;\rightarrow\; a[j] \neq e \;.$$

We compute one backward iteration through the loop to increase our confidence:

(3)

```
@L :  H : ?
S_1 :  assume i ≤ u;
S_2 :  assume a[i] ≠ e;
S_3 :  i := i + 1;
@L :  G :  i > u  →  ∀j. ℓ ≤ j ≤ u  →  a[j] ≠ e
```

Then

$$\begin{aligned}
&\mathsf{wp}(G,\; S_1; S_2; S_3)\\
&\;\Leftrightarrow\; \mathsf{wp}(\mathsf{wp}(G,\; i := i+1),\; S_1; S_2)\\
&\;\Leftrightarrow\; \mathsf{wp}(G\{i := i+1\},\; S_1; S_2)\\
&\;\Leftrightarrow\; \mathsf{wp}(\mathsf{wp}(G\{i := i+1\},\; \texttt{assume}\; a[i] \neq e),\; S_1)\\
&\;\Leftrightarrow\; \mathsf{wp}(a[i] \neq e \;\rightarrow\; G\{i := i+1\},\; S_1)\\
&\;\Leftrightarrow\; \mathsf{wp}(a[i] \neq e \;\rightarrow\; G\{i := i+1\},\; \texttt{assume}\; i \leq u)\\
&\;\Leftrightarrow\; i \leq u \;\rightarrow\; a[i] \neq e \;\rightarrow\; G\{i := i+1\}\\
&\;\Leftrightarrow\; i \leq u \;\wedge\; a[i] \neq e \;\wedge\; i+1 > u \;\rightarrow\; \forall j.\; \ell \leq j \leq u \;\rightarrow\; a[j] \neq e\\
&\;\Leftrightarrow\; i = u \;\wedge\; a[u] \neq e \;\rightarrow\; \forall j.\; \ell \leq j \leq u \;\rightarrow\; a[j] \neq e\\
&\;\Leftrightarrow\; i = u \;\wedge\; a[u] \neq e \;\rightarrow\; \forall j.\; \ell \leq j \leq u-1 \;\rightarrow\; a[j] \neq e\\
&\;\Leftrightarrow\; i = u \;\wedge\; a[u] \neq e \;\rightarrow\; \forall j.\; \ell \leq j \leq i-1 \;\rightarrow\; a[j] \neq e
\end{aligned}$$

To obtain the second-to-last line from the third-to-last, note that the antecedent already asserts that $a[u] \neq e$; hence, its occurrence as the case $j = u$ of $\forall j.\; \ell \leq j \leq u \cdots$ is redundant. The final line is realized by applying the equality $i = u$ to the upper bound on j. As we suspected, it seems that the right bound on j should be related to the progress of i, rather than being fixed to u. This observation from computing the weakest precondition matches our intuition. One trick to generalize assertions is to *replace fixed terms (bounds, indices, etc.) with terms that evolve according to the loop counter.*

Thus, we settle on the formula

$$G' : \; \forall j.\; \ell \leq j < i \;\rightarrow\; a[j] \neq e \;.$$

That is, all previously checked entries of a do not equal e. We add this assertion to the loop invariant:

```
for
   @L :  ℓ ≤ i ≤ u + 1  ∧  (∀j. ℓ ≤ j < i  →  a[j] ≠ e)
   (int i := ℓ;  i ≤ u;  i := i + 1) {
   if (a[i] = e) return true;
}
```

The result is similar to the annotation in Figure 5.15. Generating and checking the corresponding VCs reveals that the annotations are inductive. ■

Example 6.4. Consider the version of BinarySearch of Figure 5.12 that contains runtime assertions but has only a trivial function specification $\top$. Using the precondition method, we infer a function precondition that makes the annotations inductive. Contexts that call BinarySearch are then forced to obey this function precondition, guaranteeing a lack of runtime errors.

Consider the path from function entry to the assertion protecting the array access:

$(\cdot)$

```
@pre $H$ : ?
$S_1$ : assume $\ell \le u$;
$S_2$ : $m := (\ell + u)$ div 2;
@ $F$ : $0 \le m < |a|$
```

Compute

$$\begin{aligned}
&\mathsf{wp}(F,\ S_1; S_2)\\
&\Leftrightarrow\ \mathsf{wp}(\mathsf{wp}(F,\ m := (\ell + u)\ \texttt{div}\ 2),\ S_1)\\
&\Leftrightarrow\ \mathsf{wp}(F\{m \mapsto (\ell + u)\ \texttt{div}\ 2\},\ S_1)\\
&\Leftrightarrow\ \mathsf{wp}(F\{m \mapsto (\ell + u)\ \texttt{div}\ 2\},\ \texttt{assume}\ \ell \le u)\\
&\Leftrightarrow\ \ell \le u\ \rightarrow\ F\{m \mapsto (\ell + u)\ \texttt{div}\ 2\}\\
&\Leftrightarrow\ \ell \le u\ \rightarrow\ 0 \le (\ell + u)\ \texttt{div}\ 2 < |a|\\
&\Leftarrow\ 0 \le \ell\ \wedge\ u < |a|
\end{aligned}$$

The final line implies the penultimate line, for if $0 \le \ell\ \wedge\ u < |a|$ and $\ell \le u$, then both $0 \le \ell < |a|$ and $0 \le u < |a|$; hence, their mean is also in the range $[0, |a| - 1]$. Therefore, it is guaranteed that

$$0 \le \ell\ \wedge\ u < |a|\ \rightarrow\ \mathsf{wp}(F,\ S_1; S_2)$$

is $T_{\mathbb{Z}}$-valid.

The formula $0 \le \ell\ \wedge\ u < |a|$ appears as the function precondition in Figure 6.1. The annotations are inductive, proving that the runtime assertion $0 \le m < |a|$ holds in every execution of BinarySearch in which the precondition $0 \le \ell\ \wedge\ u < |a|$ is satisfied. ■

Example 6.5. Consider the following code fragment of BubbleSort (see also Figure 5.3)

```
@pre $0 \le \ell \ \wedge\ u < |a|$
@post $\top$
bool BinarySearch(int[] $a$, int $\ell$, int $u$, int $e$) {
  if ($\ell > u$) return false;
  else {
    @ $2 \neq 0$;
    int $m := (\ell + u)$ div $2$;
    @ $0 \le m < |a|$;
    if ($a[m] = e$) return true;
    else if ($a[m] < e$) return BinarySearch($a, m+1, u, e$);
    else return BinarySearch($a, \ell, m-1, e$);
  }
}
```

Fig. 6.1. BinarySearch with runtime assertions

```
for
  @$L_1$ : $-1 \le i < |a|$
  (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
  for
    @$L_2$ : $0 < i < |a| \ \wedge\ 0 \le j \le i$
    (int $j := 0$; $j < i$; $j := j + 1$) {
    if ($a[j] > a[j+1]$) {
      int $t := a[j]$;
      $a[j] := a[j+1]$;
      $a[j+1] := t$;
    }
  }
}
return $a$;
```

and its postcondition

$$F:\ \mathsf{sorted}(rv, 0, |rv| - 1)\ .$$

Consider the path

(6)

```
@$L_1$ : $G$ : ?
$S_1$ : assume $i \le 0$;
$S_2$ : $rv := a$;
@post $F$ : sorted($rv, 0, |rv| - 1$)
```

Computing $\mathsf{wp}(F,\ S_1; S_2)$ produces the formula

$$F':\ i \le 0\ \rightarrow\ \mathsf{sorted}(a, 0, |a| - 1)\ ,$$

which tells us (not surprisingly) that a should be sorted upon exiting the outer loop. Observe the index variable of the outer loop: it starts at $|a| - 1$

and decrements down to 0. Therefore, recalling the trick to *replace fixed terms (bounds, indices, etc.) with terms that evolve according to the loop counter* suggests the following generalization of F':

$$G : \ \mathsf{sorted}(a, i, |a| - 1) \ .$$

G trivially holds upon entering the outer loop; moreover, it follows from the behavior of i that progress is made by working down the array. The outer loop invariant L_1 should include G. Thus, we have

$$@L_1 : \ -1 \le i < |a| \ \land \ \mathsf{sorted}(a, i, |a| - 1)$$

so far.

Propagate G via wp to the inner loop along the path from the exit of the inner loop L_2 to the top of the outer loop L_1:

(5)

$@L_2 : \ H : ?$
$S_1 : \ \texttt{assume}\ j \ge i;$
$S_2 : \ i := i - 1;$
$@L_1 : \ G : \ \mathsf{sorted}(a, i, |a| - 1)$

The result at L_2 is the formula

$$H' : \ j \ge i \ \rightarrow \ \mathsf{sorted}(a, i - 1, |a| - 1) \ ,$$

which states that when the inner loop has *finished*, the range $[i - 1, |a| - 1]$ is sorted. Immediately generalizing H' to

$$H'' : \ \mathsf{sorted}(a, i - 1, |a| - 1)$$

is too strong. For suppose H'' were to annotate the inner loop at L_2, and consider the path

(2)

$@L_1 : \ G : \ \mathsf{sorted}(a, i, |a| - 1)$
$S_1 : \ \texttt{assume}\ i > 0;$
$S_2 : \ j := 0;$
$@L_2 : \ H'' : \ \mathsf{sorted}(a, i - 1, |a| - 1)$

Computing

$$G \ \rightarrow \ \mathsf{wp}(H'', \ \texttt{assume}\ i > 0; \ j := 0)$$

produces

$$\mathsf{sorted}(a, i, |a| - 1) \ \land \ i > 0 \ \rightarrow \ \mathsf{sorted}(a, i - 1, |a| - 1) \ ,$$

which is not $(T_{\mathbb{Z}} \cup T_{\mathsf{A}})$-valid. All we know at L_1 (with respect to sortedness of a) is $G : \ \mathsf{sorted}(a, i, |a| - 1)$. Essentially, $\mathsf{sorted}(a, i - 1, |a| - 1)$ at L_2 is a special case that definitely holds only when the inner loop has finished. Therefore, we generalize H' to the weaker assertion $H : \mathsf{sorted}(a, i, |a| - 1)$, which claims that a smaller subrange of a is sorted.

At this point, we have annotated the loops of BubbleSort as follows:

```
for
  $@L_1 : \ -1 \leq i < |a| \ \wedge \ \mathsf{sorted}(a, i, |a| - 1)$
  (int $i := |a| - 1;\ i > 0;\ i := i - 1$) {
  for
    $@L_2 : \ 0 < i < |a| \ \wedge \ 0 \leq j \leq i \ \wedge \ \mathsf{sorted}(a, i, |a| - 1)$
    (int $j := 0;\ j < i;\ j := j + 1$) {
    if ($a[j] > a[j + 1]$) {
      int $t := a[j]$;
      $a[j] := a[j + 1]$;
      $a[j + 1] := t$;
    }
  }
}
```

The resulting VCs are not valid. Further annotations require some insight on our part, which leads us to the next section. ■

6.1.3 A Strategy

In general, proofs require insights beyond generalizing formulae obtained through the precondition method. We adopt the following strategy when proving partial correctness.

First, decompose the function specification into atomic properties. Then analyze each atomic property. For example, to prove that BubbleSort returns a sorted array that is a permutation of the input, study the sortedness property and permutation property separately. In some cases, several atomic properties may have to be examined together to complete the proof. For each basic property, apply the following steps:

1. Assert basic facts (Section 6.1.1).
2. Repeat:
   a) Use the precondition method to propagate annotations (Section 6.1.2).
   b) Formalize an insight.

The second step suggests applying the precondition method until nothing more can be learned. Then pause, understand another essential fact about the program, and resume applying the precondition method.

While Chapter 12 discusses the foundations of algorithms for automatically generating inductive annotations, the reader should be aware that even the best of these algorithms cannot approach the abilities of a human. Take heart! Experience has shown that students quickly become adept at annotating programs.

Example 6.6. We resume our analysis of BubbleSort from Example 6.5. Some cogitation (and observation of sample traces; see Figure 5.4) suggests that BubbleSort exhibits the following behavior: the inner loop propagates the largest value of the unsorted region to the right side of the unsorted region, thus expanding the sorted region. At every iteration, j is the index of the largest value found so far. In other words, all values in the range $[0, j-1]$ are at most $a[j]$:

$$F : \ \mathsf{partitioned}(a, 0, j-1, j, j) \ .$$

This observations should be added as an annotation at L_2. Having gained new insight into BubbleSort, we return to the precondition method and propagate F back to the outer loop at L_1 along the path

(2)

```
@$L_1$ : $H$ : ?
$S_1$ : assume $i > 0$;
$S_2$ : $j := 0$;
@$L_2$ : $F$ : $\mathsf{partitioned}(a, 0, j-1, j, j)$
```

resulting in the new annotation

$$\mathsf{wp}(F,\ S_1; S_2) : \ i > 0 \ \rightarrow \ \mathsf{partitioned}(a, 0, -1, 0, 0)$$

at L_1. The result is trivially valid according to the definition of partitioned, so it does not contribute any new information. Thus, we finish this round of Step 2 with the annotations

```
for
  @$L_1$ : $-1 \le i < |a| \ \wedge \ \mathsf{sorted}(a, i, |a|-1)$
  (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
  for
    @$L_2$ : $\begin{bmatrix} 0 < i < |a| \ \wedge \ 0 \le j \le i \\ \wedge \ \mathsf{partitioned}(a, 0, j-1, j, j) \ \wedge \ \mathsf{sorted}(a, i, |a|-1) \end{bmatrix}$
    (int $j := 0$; $j < i$; $j := j + 1$) {
    if ($a[j] > a[j+1]$) {
      int $t := a[j]$;
      $a[j] := a[j+1]$;
      $a[j+1] := t$;
    }
  }
}
```

The annotations are not yet inductive.

Some further meditation enlightens us with the following: the sorted region must contain the largest elements of a, for the inner loop has assiduously moved the largest element of the unsorted region to the sorted region. In other words, the sorted range $[i+1, |a|-1]$ contains the largest elements of a:

$$G : \ \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \ .$$

This observation should be added as an annotation at L_1. Now we propagate G from L_1 to L_2. Recall from Example 6.5 that our propagation of $\mathsf{sorted}(a, i, |a|-1)$ to the inner loop was unsuccessful. Similarly,

$$\begin{array}{l} \mathsf{wp}(G, \ \texttt{assume}\ j \geq i; \ i := i-1) \\ \ \Leftrightarrow \ j \geq i \ \rightarrow \ \mathsf{partitioned}(a, 0, i-1, i, |a|-1) \end{array}$$

for the path

(5)

$@L_2 : \ H : ?$
$\texttt{assume}\ j \geq i;$
$i := i-1;$
$@L_1 : \ G : \ \mathsf{partitioned}(a, 0, i, i+1, |a|-1)$

cannot be generalized to

$$\mathsf{partitioned}(a, 0, i-1, i, |a|-1) \ .$$

Instead, consider the path from L_1 to L_2:

(2)

$@L_1 : \ G : \ \mathsf{partitioned}(a, 0, i, i+1, |a|-1)$
$S_1 : \ \texttt{assume}\ i > 0;$
$S_2 : \ j := 0;$
$@L_2 : \ H : ?$

Find the strongest formula H that can annotate the inner loop such that the VC

$$\mathsf{partitioned}(a, 0, i, i+1, |a|-1) \ \rightarrow \ \mathsf{wp}(H, \ S_1; S_2)$$

for the path is valid. In other words, seek a formula H annotating the inner loop that is supported by the annotation G of the outer loop. The strongest such formula is G itself.

These new annotations result in Figure 5.17. ■

6.2 Extended Example: QuickSort

In this section, we bring together the concepts studied in this and the last chapters through a single example. We prove that QuickSort always halts and returns a sorted array. We argue at the level of annotations, leaving a computer or the reader to check the VCs.

Figure 6.2 lists the high-level functions of QuickSort. QuickSort is a wrapper function (or public interface) for the recursive function qsort, which sorts array a_0 in the range $[\ell, u]$. As in BubbleSort, the first line of qsort assigns a_0 to an

```
typedef struct qs {
  int $pivot$;
  int[] $array$;
} qs;

@pre $\top$
@post sorted($rv, 0, |rv| - 1$)
int[] QuickSort(int[] $a$) {
  return qsort($a, 0, |a| - 1$);
}

@pre $\top$
@post $\top$
int[] qsort(int[] $a_0$, int $\ell$, int $u$) {
  int[] $a := a_0$;
  if ($\ell \geq u$) return $a$;
  else {
    qs $p :=$ partition($a, \ell, u$);
    $a := p.array$;
    $a :=$ qsort($a, \ell, p.pivot - 1$);
    $a :=$ qsort($a, p.pivot + 1, u$);
    return $a$;
  }
}
```

Fig. 6.2. Main functions of QuickSort

array a because qsort modifies a (recall that pi does not allow parameters to be modified). The qs data structure holds the two data that the partition function, listed in Figure 6.3, returns: the pivot index *pivot* and the partitioned array *array*.

One level of recursion of qsort works as follows. If $\ell \geq u$, then the trivial range $[\ell, u]$ of a_0 is already sorted. Otherwise, partition chooses a *pivot index* $pi \in [\ell, u]$, remembering the *pivot value* $a[pi]$ as pv. It then swaps cells pi and u of a so that the randomly chosen pivot now appears on the right side of the $[\ell, u]$ subarray. random has the following prototype:

```
@pre $\ell \leq u$
@post $\ell \leq rv \leq u$
int random(int $\ell$, int $u$);
```

The for loop of partition partitions a such that all elements at most pv are on the left and all elements greater than pv are on the right. Within the loop, $j < u$, so that the pivot value pv, stored in $a[u]$, remains untouched. When the loop finishes, if $i < u - 1$, then the value $a[i + 1]$ is the first value greater than pv; otherwise, all elements of a are at most pv. Finally, partition

```
@pre $\top$
@post $\top$
qs partition(int[] $a_0$, int $\ell$, int $u$) {
  int[] $a := a_0$;
  int $pi :=$ random$(\ell, u)$;
  int $pv := a[pi]$;
  $a[pi] := a[u]$;
  $a[u] := pv$;

  int $i := \ell - 1$;
  for @ $\top$
    (int $j := \ell$; $j < u$; $j := j + 1$) {
    if $(a[j] \leq pv)$ {
      $i := i + 1$;
      $t := a[i]$;
      $a[i] := a[j]$;
      $a[j] := t$;
    }
  }

  $t := a[i + 1]$;
  $a[i + 1] := a[u]$;
  $a[u] := t$;
  return
    { $pivot = i + 1$;
      $a = a$;
    };
}
```

Fig. 6.3. QuickSort's partition function

swaps the pivot value $a[u]$ with $a[i + 1]$ so that a is partitioned as follows in the range $[\ell, u]$: cells to the left of $i + 1$ have value at most pv; $a[i + 1] = pv$; and cells to the right of $i + 1$ have value greater than pv. It returns the pivot index $i + 1$ and the partitioned array a via an instance of the qs data type.

Finally, qsort recursively sorts the subarrays to the left and to the right of the pivot index.

Figure 6.4 presents a sample trace. In the first line, partition chooses the second cell as the pivot and swaps it with cell u. The subsequent six lines follow the partition's loop as it partitions elements according to pv. The penultimate line shows the swap that brings the pivot element into the pivot position. The final line shows the state of the array when it is returned to qsort. qsort calls itself recursively on the two indicated subarrays. We encourage the reader to understand QuickSort and the sample trace before reading further.

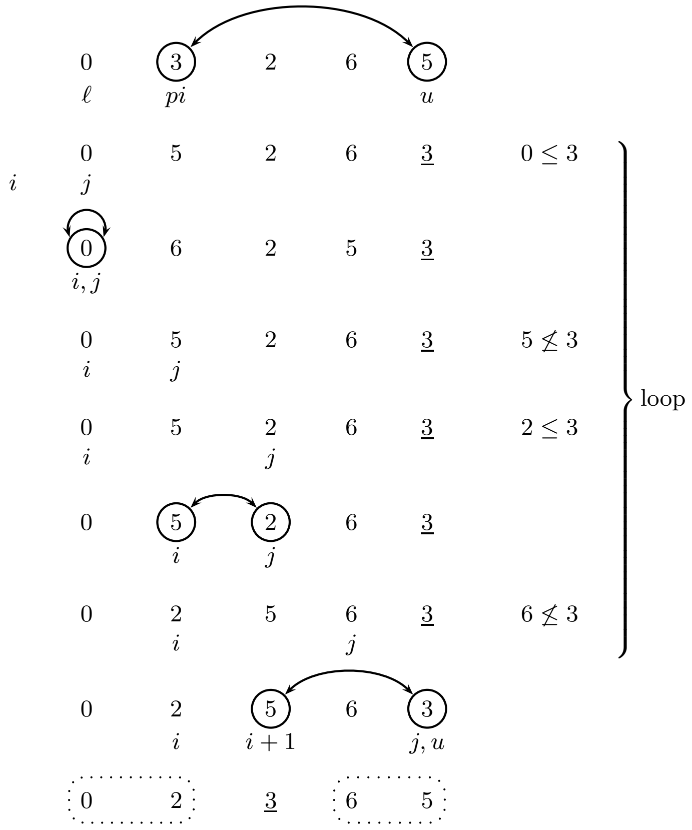

Fig. 6.4. Sample execution of QuickSort

6.2.1 Partial Correctness

We prove that if QuickSort halts, then it returns a sorted array. First, we develop the function specifications for qsort and partition so that QuickSort and qsort have inductive annotations. We then leave the annotation of the loop of partition as Exercise 6.3.

First, annotate qsort and partition with their function specifications. To avoid runtime errors, the function preconditions should include

$$0 \leq \ell \;\wedge\; u < |a_0| \;.$$

Next, while the returned array is not the same as the input array a_0 in either function, we know that their lengths are the same:

$$|rv| = |a_0| \;.$$

We also observe that neither qsort nor partition modify the array outside of the range $[\ell, u]$. Thus, we note in the function postcondition that

$$\mathsf{beq}(rv, a_0, 0, \ell - 1) \ \wedge\ \mathsf{beq}(rv, a_0, u + 1, |a_0| - 1) \ .$$

The **bounded equality** predicate beq is defined

$$\mathsf{beq}(a, b, k_1, k_2) \ \Leftrightarrow\ \forall i.\ k_1 \le i \le k_2 \ \rightarrow\ a[i] = b[i]$$

in the theory $T_{\mathbb{Z}} \cup T_{\mathsf{A}}$. It asserts that two arrays are equal in the index range $[k_1, k_2]$.

The annotations for partition vary slightly because of its return type:

$$\begin{aligned}&|rv.array| = |a_0| \ \wedge\ \mathsf{beq}(rv.array, a_0, 0, \ell - 1)\\ &\ \wedge\ \mathsf{beq}(rv.array, a_0, u + 1, |a_0| - 1)\end{aligned}$$

Since partition returns an integer (*pivot*) and an array (*array*) in a qs structure, the postcondition asserts facts about *rv.array*.

To finish with qsort, formalize that qsort sorts the range $[\ell, u]$ of the given array:

$$\mathsf{sorted}(rv, \ell, u) \ .$$

So far, then, we have specified qsort as follows:

```
@pre $0 \le \ell \ \wedge\ u < |a_0|$
@post $\begin{bmatrix}|rv| = |a_0| \ \wedge\ \mathsf{beq}(rv, a_0, 0, \ell - 1) \ \wedge\ \mathsf{beq}(rv, a_0, u + 1, |a_0| - 1)\\ \wedge\ \mathsf{sorted}(rv, \ell, u)\end{bmatrix}$
int[] qsort(int[] $a_0$, int $\ell$, int $u$)
```

To finish the specification of partition, recall that partition is intended to return an array that is partitioned around the pivot element. Therefore, let us formalize the description that we gave above. Observe that the specification of random guarantees that

$$\ell \le rv.pivot \le u \ ,$$

as desired. Now, for the left subarray,

$$\forall i.\ \ell \le i < rv.pivot \ \rightarrow\ rv.array[i] \le rv.array[rv.pivot] \ ,$$

or, as a partition,

$$\mathsf{partitioned}(rv.array, \ell, rv.pivot - 1, rv.pivot, rv.pivot) \ ,$$

while for the right subarray,

$$\forall i.\ rv.pivot < i \le u \ \rightarrow\ rv.array[rv.pivot] < rv.array[i] \ .$$

We weaken this assertion slightly to

$$\mathsf{partitioned}(rv.array, rv.pivot, rv.pivot, rv.pivot + 1, u) \ ,$$

which does not capture the strict inequality but is more convenient for reasoning. For partition, we have thus specified the following:

$$
\begin{array}{l}
\text{@pre } 0 \le \ell \ \land\ u < |a_0| \\
\text{@post} \begin{bmatrix}
|rv.array| = |a_0| \ \land\ \mathsf{beq}(rv.array, a_0, 0, \ell - 1) \\
\quad \land\ \mathsf{beq}(rv.array, a_0, u + 1, |a_0| - 1) \\
\quad \land\ \ell \le rv.pivot \le u \\
\quad \land\ \mathsf{partitioned}(rv.array, \ell, rv.pivot - 1, rv.pivot, rv.pivot) \\
\quad \land\ \mathsf{partitioned}(rv.array, rv.pivot, rv.pivot, rv.pivot + 1, u)
\end{bmatrix} \\
\mathsf{qs}\ \mathsf{partition}(\texttt{int}[]\ a_0,\ \texttt{int}\ \ell,\ \texttt{int}\ u)
\end{array}
$$

Let us step back a moment. Essentially, we have specified that qsort does not modify the array outside of the range $[\ell, u]$. Regarding the subarray given by $[\ell, u]$, all we have asserted is that it is sorted in the returned array. Focus on the recursive calls to qsort: is knowing that the ranges $[\ell, p.pivot - 1]$ and $[p.pivot + 1, u]$ of a are sorted enough to conclude that the range $[\ell, u]$ is sorted when a is returned? In other words, is the VC corresponding to the following basic path valid? The basic path follows the path from the precondition to the second `return` statement, using the function call abstraction introduced in Section 5.2.1 to abstract away functions calls:

$(\cdot)$

$$
\begin{array}{l}
\text{@pre } 0 \le \ell \ \land\ u < |a_0| \\
a := a_0; \\
\texttt{assume}\ \ell < u; \\
\texttt{assume} \begin{bmatrix}
|v_1.array| = |a| \ \land\ \mathsf{beq}(v_1.array, a, 0, \ell - 1) \\
\quad \land\ \mathsf{beq}(v_1.array, a, u + 1, |a| - 1) \\
\quad \land\ \ell \le v_1.pivot \le u \\
\quad \land\ \mathsf{partitioned}(v_1.array, \ell, v_1.pivot - 1, v_1.pivot, v_1.pivot) \\
\quad \land\ \mathsf{partitioned}(v_1.array, v_1.pivot, v_1.pivot, v_1.pivot + 1, u)
\end{bmatrix}; \\
p := v_1; \\
a := p.array; \\
\texttt{assume} \begin{bmatrix}
|v_2| = |a| \ \land\ \mathsf{beq}(v_2, a, 0, \ell - 1) \ \land\ \mathsf{beq}(v_2, a, p.pivot, |a| - 1) \\
\quad \land\ \mathsf{sorted}(v_2, \ell, p.pivot - 1)
\end{bmatrix}; \\
a := v_2; \\
\texttt{assume} \begin{bmatrix}
|v_3| = |a| \ \land\ \mathsf{beq}(v_3, a, 0, p.pivot) \ \land\ \mathsf{beq}(v_3, a, u + 1, |a| - 1) \\
\quad \land\ \mathsf{sorted}(v_3, p.pivot + 1, u)
\end{bmatrix}; \\
a := v_3; \\
rv := a; \\
@ \begin{bmatrix}
|rv| = |a_0| \ \land\ \mathsf{beq}(rv, a_0, 0, \ell - 1) \ \land\ \mathsf{beq}(rv, a_0, u + 1, |a_0| - 1) \\
\quad \land\ \mathsf{sorted}(rv, \ell, u)
\end{bmatrix}
\end{array}
$$

The corresponding VC is not valid. The assumptions about v_1, v_2, and v_3 are not strong enough to imply that the range $[\ell, u]$ of rv is sorted.

The standard approach to addressing this problem is to reason simultaneously that qsort returns an array that is a permutation of its input array (a permuted array contains the same elements as the original array but possibly in a different order). However, reasoning about permutations presents a problem. A straightforward formalization of permutation is not possible in FOL, instead requiring second-order logic. We could assert that the output is a **weak permutation** of the input: all values occurring in the input array occur in the output array but possibly with a varying number of occurrences. Formally,

$$\forall e.\ (\exists i.\ a_0[i] = e) \ \leftrightarrow\ (\exists j.\ rv[j] = e)\ .$$

In Exercise 6.5, we ask the reader to explore an approximation to weak permutation.

However, that the elements are permuted is a stronger statement than necessary to prove sortedness. Instead, notice that QuickSort imposes a larger partitioning of the intermediate arrays than we have previously observed in our analysis. At every level of recursion of qsort, the elements of a_0 in the range $[\ell, u]$ are at least the elements to their left and at most the elements to their right. Formally, we strengthen the specification of qsort as follows:

$$
\begin{array}{l}
\mathsf{@pre}\left[\begin{array}{l}
0 \le \ell \ \wedge\ u < |a_0| \\
\ \wedge\ \mathsf{partitioned}(a_0, 0, \ell-1, \ell, u) \\
\ \wedge\ \mathsf{partitioned}(a_0, \ell, u, u+1, |a_0|-1)
\end{array}\right] \\
\mathsf{@post}\left[\begin{array}{l}
|rv| = |a_0| \ \wedge\ \mathsf{beq}(rv, a_0, 0, \ell-1) \ \wedge\ \mathsf{beq}(rv, a_0, u+1, |a_0|-1) \\
\ \wedge\ \mathsf{partitioned}(rv, 0, \ell-1, \ell, u) \\
\ \wedge\ \mathsf{partitioned}(rv, \ell, u, u+1, |rv|-1) \\
\ \wedge\ \mathsf{sorted}(rv, \ell, u)
\end{array}\right] \\
\texttt{int}[]\ \mathsf{qsort}(\texttt{int}[]\ a_0,\ \texttt{int}\ \ell,\ \texttt{int}\ u)
\end{array}
$$

Of course, now the specification of partition must be strengthened to carry this reasoning through the main basic path of qsort:

$$
\begin{array}{l}
\mathsf{@pre}\left[\begin{array}{l}
0 \le \ell \ \wedge\ u < |a_0| \\
\ \wedge\ \mathsf{partitioned}(a_0, 0, \ell-1, \ell, u) \\
\ \wedge\ \mathsf{partitioned}(a_0, \ell, u, u+1, |a_0|-1)
\end{array}\right] \\
\mathsf{@post}\left[\begin{array}{l}
|rv.array| = |a_0| \ \wedge\ \mathsf{beq}(rv.array, a_0, 0, \ell-1) \\
\ \wedge\ \mathsf{beq}(rv.array, a_0, u+1, |a_0|-1) \\
\ \wedge\ \mathsf{partitioned}(rv.array, 0, \ell-1, \ell, u) \\
\ \wedge\ \mathsf{partitioned}(rv.array, \ell, u, u+1, |rv.array|-1) \\
\ \wedge\ \ell \le rv.pivot \le u \\
\ \wedge\ \mathsf{partitioned}(rv.array, \ell, rv.pivot-1, rv.pivot, rv.pivot) \\
\ \wedge\ \mathsf{partitioned}(rv.array, rv.pivot, rv.pivot, rv.pivot+1, u)
\end{array}\right] \\
\mathsf{qs}\ \mathsf{partition}(\texttt{int}[]\ a_0,\ \texttt{int}\ \ell,\ \texttt{int}\ u)
\end{array}
$$

That is, partition preserves this partitioning even as it manipulates the elements in the range $[\ell, u]$. Indeed, partition itself imposes the necessary partitioning for the next level of recursion, which we already observed earlier as the final partitioned assertions of the function postcondition.

The annotations of QuickSort and qsort are inductive. Exercise 6.3 asks the reader to finish the proof by annotating the `for` loop of partition so that the annotations of partition are also inductive.

6.2.2 Total Correctness

To prove total correctness — that QuickSort actually returns a sorted arrays — we need to prove that QuickSort always halts. Our implementation of QuickSort has both recursive behavior in function qsort and looping behavior in function partition. We must analyze both possible sources of nontermination.

To prove that the loop in partition always halts, let us use the obvious ranking function of $\delta_1 : u - j$ suggested by the structure of the loop. δ_1 clearly maps the program state to $\mathbb{Z}$, but we prove the stronger fact that δ_1 actually maps the program state to $\mathbb{N}$ with well-founded relation $<$. In particular, we prove that

- $u - j \geq 0$ is a loop invariant;
- and $u - j$ decreases on each iteration.

Annotating the loop with the bounds on j suggested by the loop structure proves that the loop always halts:

```
for
  @$L_1$ : $\ell \leq j \;\wedge\; j \leq u$
  $\downarrow \delta_1$ : $u - j$
  (int $j := \ell$; $j < u$; $j := j + 1$)
```

Proving that the recursion of qsort always halts is superficially more difficult. The argument that we would like to make is that $u - \ell$ decreases on each recursive call, which requires proving that the pivot value returned by partition lies within the range $[\ell, u]$.

Observe, however, that $u - \ell$ may be negative when qsort is called with $\ell > u$. But in this case, $\ell = u + 1$, for either $|a_0| = 0$, and qsort was called from QuickSort; or $p.pivot = \ell$ or $p.pivot = u$, and qsort was called recursively. More generally, we can establish that $u - \ell + 1 \geq 0$ is an invariant of qsort. Hence, $\delta_2 : u - \ell + 1$ is our proposed ranking function that maps the program states to $\mathbb{N}$ with well-founded relation $<$.

Figure 6.5 formalizes the arguments that δ_1 and δ_2 are ranking functions. Notice that bounds on i are proved as loop invariants at L_1. These bounds imply that $rv.pivot$ lies within the range $[\ell, u]$ as required.

One trick that would avoid reasoning about the case in which $\ell > u$ is to cut the recursion at a point within qsort rather than at function entry. Figure 6.6 provides an alternate argument in which the ranking function labels the

```
@pre $u - \ell + 1 \geq 0$
@post $\top$
$\downarrow \delta_2 :\ u - \ell + 1$
int[] qsort(int[] $a_0$, int $\ell$, int $u$) {
  int[] $a := a_0$;
  if $(\ell \geq u)$ return $a$;
  else {
    qs $p :=$ partition$(a, \ell, u)$;
    $a := p.array$;
    $a :=$ qsort$(a, \ell, p.pivot - 1)$;
    $a :=$ qsort$(a, p.pivot + 1, u)$;
    return $a$;
  }
}

@pre $\ell \leq u$
@post $\ell \leq rv.pivot\ \wedge\ rv.pivot \leq u$
qs partition(int[] $a_0$, int $\ell$, int $u$) {
  ⋮
  int $i := \ell - 1$;
  for
    @$L_1 :\ \ell \leq j\ \wedge\ j \leq u\ \wedge\ \ell - 1 \leq i\ \wedge\ i < j$
    $\downarrow \delta_1 :\ u - j$
    (int $j := \ell$; $j < u$; $j := j + 1$) {
    ⋮
  }
  ⋮
  return
    { $pivot\ =\ i + 1$;
      $a\ =\ a$;
    };
}
```

Fig. 6.5. QuickSort always halts

else branch in qsort. The first branch terminates the recursion. partition is annotated as in Figure 6.5.

6.3 Summary

This chapter presents strategies for specifying and proving the correctness of sequential programs. It covers:

- *Strategies* for proving partial correctness. The need for strengthening annotations. Basic facts; the precondition method.

```
@pre $\top$
@post $\top$
int[] qsort(int[] $a_0$, int $\ell$, int $u$) {
  int[] $a := a_0$;
  if ($\ell \geq u$) return $a$;
  else {
    $\downarrow \delta_3 :\ u - \ell$
    qs $p :=$ partition$(a, \ell, u)$;
    $a := p.array$;
    $a :=$ qsort$(a, \ell, p.pivot - 1)$;
    $a :=$ qsort$(a, p.pivot + 1, u)$;
    return $a$;
  }
}
```

Fig. 6.6. Alternate argument that QuickSort always halts

```
@pre $\top$
@post $\forall i.\ 0 \leq i < |rv| \ \rightarrow\ rv[i] \geq 0$
int[] abs(int[] $a_0$) {
  int[] $a := a_0$;
  for @ $\top$
    (int $i := 0$; $i < |a|$; $i := i + 1$) {
    if ($a[i] < 0$) {
      $a[i] :=\ - a[i]$;
    }
  }
  return $a$;
}
```

Fig. 6.7. Computing the absolute value of elements of a_0

- A full proof that QuickSort returns a sorted array.

Bibliographic Remarks

QuickSort was discovered by Tony Hoare, who also proposed specifying and verifying programs using FOL [39].

Exercises

6.1 (Absolute value). Prove the partial correctness of abs in Figure 6.7. That is, annotate the function; list basic paths and verification conditions; and argue that the VCs are valid.

```
@pre $\top$
@post sorted$(rv, 0, |rv| - 1)$
int[] InsertionSort(int[] $a_0$) {
  int[] $a := a_0$;
  for @ $\top$
    (int $i := 1$; $i < |a|$; $i := i + 1$) {
    int $t := a[i]$;
    for @ $\top$
      (int $j := i - 1$; $j \geq 0$; $j := j - 1$) {
      if ($a[j] \leq t$) break;
      $a[j + 1] := a[j]$;
    }
    $a[j + 1] := t$;
  }
  return $a$;
}
```

Fig. 6.8. InsertionSort

6.2 (InsertionSort). Prove the partial correctness of InsertionSort. That is, annotate the function; list basic paths and verification conditions; and argue that the VCs are valid. See Figure 6.8.

As in other languages, the `break` statement moves control to the loop exit.

6.3 (QuickSort). Finish the proof of the sortedness property of QuickSort. That is, annotate partition of Figure 6.3 so that its precondition and postcondition annotations given at the end of Section 6.2 are inductive.

6.4 (MergeSort). Prove the partial correctness of MergeSort. See Figure 6.9. The function merge uses the pi keyword `new`, which allocates an array of the specified size. Therefore, it is known after the allocation to buf that $|buf| = u - \ell + 1$.

First, deduce the function specifications for ms and merge by focusing on MergeSort and ms. Prove MergeSort and ms correct with respect to these annotations. Then analyze merge.

Since MergeSort is fairly long, you need not list basic paths and VCs. Just present MergeSort with its inductive annotations.

6.5 (Weak permutation). Define **weak permutation** as follows:

$$\forall e.\ (\exists i.\ a[i] = e) \ \leftrightarrow \ (\exists j.\ b[j] = e)\ . \tag{6.1}$$

Unfortunately, the decision procedures for arrays discussed in Chapter 11 cannot decide the validity of VCs arising from wperm annotations, as such VCs fall outside of the studied fragments of T_{A}. Instead, we describe an approximation.

Consider annotating BubbleSort as in Figure 6.10. The `define` keyword defines a global constant. In this case, e is defined to have some nondeterministic value; that is, e stands for an arbitrary integer. The annotations then use this e in wperm literals, where wperm is defined as follows:

$$\mathsf{wperm}(a, a_0, e) \;\Leftrightarrow\; (\exists i.\; a[i] = e) \;\leftrightarrow\; (\exists j.\; a_0[j] = e)\ .$$

Compared to the full definition (6.1) of weak permutation, wperm does not have a universally quantified variable e; instead, it uses a given expression e, in this case the global constant e.

(a) Argue that the annotations of Figure 6.10 are inductive. That is, list the VCs and argue their validity.

(b) Argue that the annotations imply that BubbleSort actually satisfies the weakest permutation property. That is, prove that the validity of the VC

$$\mathsf{wperm}(a, a_0, e) \;\wedge\; a' = \ldots \;\rightarrow\; \mathsf{wperm}(a', a_0, e)$$

implies the validity of the VC

$$\begin{array}{l}(\forall e.\; (\exists i.\; a[i] = e) \;\leftrightarrow\; (\exists j.\; a_0[j] = e)) \;\wedge\; a' = \ldots \\ \quad\quad \rightarrow\; (\forall e.\; (\exists i.\; a'[i] = e) \;\leftrightarrow\; (\exists j.\; a_0[j] = e))\ .\end{array}$$

(c) Can this approximation be used to prove the weakest permutation property of

- **(i)** InsertionSort (Figure 6.8)?
- **(ii)** MergeSort (Figure 6.9)?
- **(iii)** QuickSort (Section 6.2)?

If so, prove it. If not, explain why not.

6.6 (Sets with arrays). Implement an API (**application programming interface**) for manipulating sets. The underlying data structure of the implementation is arrays.

(a) Prove the correctness of the union function of Figure 6.11 by adding inductive annotations.

(b) Implement and specify and prove the correctness of an intersection function, which takes two arrays a_0 and b_0 and returns the intersection of the sets they represent.

(c) Implement and specify and prove the correctness of a subset function, which takes two arrays a_0 and b_0 and returns `true` iff the first set, represented by a_0, is a subset of the second set, represented by b_0.

6.7 (Sets with sorted arrays). Implement an API for manipulating sets. The underlying data structure of the implementation is sorted arrays.

(a) Prove the correctness of the union function of Figure 6.12 by adding inductive annotations.

(b) Implement and specify and prove the correctness of an intersection function, which takes two sorted arrays a_0 and b_0 and returns the intersection of the sets they represent as a sorted set.

(c) Implement and specify and prove the correctness of a subset function, which takes two sorted arrays a_0 and b_0 and returns `true` iff the first set, represented by a_0, is a subset of the second set, represented by b_0.

6.8 (QuickSort halts). Provide the basic paths and verification conditions for the proof of Section 6.2.2 that QuickSort always halts.

6.9 (Intuitive ranking functions). Following the proof that the recursion of qsort halts, move the location of the ranking function annotations in the following functions to produce more intuitive arguments:

(a) BinarySearch, Figure 5.2

(b) BubbleSort, Figure 5.3

(c) InsertionSort, Figure 6.8

6.10 (Fewer annotations). Notice in the annotated BubbleSort of Figure 5.17 that there are only a finite number of basic paths from function entry to L_2, from L_2 to function exit, and from L_2 back to itself.

(a) List these basic paths.

(b) Annotate only the inner loop of BubbleSort so that the VCs corresponding to the basic paths of **(a)** are valid.

(c) Treat InsertionSort of Figure 5.25 similarly.

(d) Similarly, annotate only the inner loop of BubbleSort with a ranking annotation.

(e) Treat InsertionSort similarly.

```
@pre $\top$
@post sorted$(rv, 0, |rv| - 1)$
int[] MergeSort(int[] $a$) {
  return ms$(a, 0, |a| - 1)$;
}

@pre $\top$
@post $\top$
int[] ms(int[] $a_0$, int $\ell$, int $u$) {
  int[] $a := a_0$;
  if $(\ell \geq u)$ return $a$;
  else {
    int $m := (\ell + u)$ div 2;
    $a :=$ ms$(a, \ell, m)$;
    $a :=$ ms$(a, m + 1, u)$;
    $a :=$ merge$(a, \ell, m, u)$;
    return $a$;
  }
}

@pre $\top$
@post $\top$
int[] merge(int[] $a_0$, int $\ell$, int $m$, int $u$) {
  int[] $a := a_0$, $buf :=$ new int$[u - \ell + 1]$;
  int $i := \ell$, $j := m + 1$;
  for @ $\top$
    (int $k := 0$; $k < |buf|$; $k := k + 1$) {
    if $(i > m)$ {
      $buf[k] := a[j]$;
      $j := j + 1$;
    } else if $(j > u)$ {
      $buf[k] := a[i]$;
      $i := i + 1$;
    } else if $(a[i] \leq a[j])$ {
      $buf[k] := a[i]$;
      $i := i + 1$;
    } else {
      $buf[k] := a[j]$;
      $j := j + 1$;
    }
  }
  for @ $\top$
    ($k := 0$; $k < |buf|$; $k := k + 1$) {
    $a[\ell + k] := buf[k]$;
  }
  return $a$;
}
```

Fig. 6.9. MergeSort

```
define int $e = ?$;

@pre $\top$
@post wperm$(a, a_0, e)$
int[] BubbleSort(int[] $a_0$) {
  int[] $a := a_0$;
  for
    @$L_1$ : $-1 \leq i < |a| \ \wedge$ wperm$(a, a_0, e)$
    (int $i := |a| - 1$; $i > 0$; $i := i - 1$) {
    for
      @$L_2$ : $0 \leq j < i \ \wedge \ i < |a| \ \wedge$ wperm$(a, a_0, e)$
      (int $j := 0$; $j < i$; $j := j + 1$) {
      if $(a[j] > a[j + 1])$ {
        int $t := a[j]$;
        $a[j] := a[j + 1]$;
        $a[j + 1] := t$;
      }
    }
  }
  return $a$;
}
```

Fig. 6.10. BubbleSort with annotations for weak permutation

```
define int $e = ?$;

@pre $\top$
@post $\left[\begin{array}{l}(\exists i.\ 0 \leq i < |rv| \ \wedge \ rv[i] = e) \\ \quad \leftrightarrow \ (\exists i.\ 0 \leq i < |a_0| \ \wedge \ a_0[i] = e) \ \vee \ (\exists i.\ 0 \leq i < |b_0| \ \wedge \ b_0[i] = e)\end{array}\right]$
int[] union(int[] $a_0$, int[] $b_0$) {
  int[] $u :=$ new int$[|a_0| + |b_0|]$;
  int $j := 0$;
  for @ $\top$
    (int $i = 0$; $i < |a_0|$; $i := i + 1$) {
    $u[j] := a_0[i]$;
    $j := j + 1$;
  }
  for @ $\top$
    (int $i = 0$; $i < |b_0|$; $i := i + 1$) {
    $u[j] := b_0[i]$;
    $j := j + 1$;
  }
  return $u$;
}
```

Fig. 6.11. Function union of the linear set implementation

```
define int $e = ?$;

@pre sorted$(a_0, 0, |a_0| - 1) \;\wedge\;$ sorted$(b_0, 0, |b_0| - 1)$
@post $\left[\begin{array}{l}\mathsf{sorted}(rv, 0, |rv| - 1) \\ \wedge \left[\begin{array}{l}(\exists i.\ 0 \le i < |a_0| \;\wedge\; a_0[i] = e) \;\vee\; (\exists i.\ 0 \le i < |b_0| \;\wedge\; b_0[i] = e) \\ \leftrightarrow\; (\exists i.\ 0 \le i < |rv| \;\wedge\; rv[i] = e)\end{array}\right]\end{array}\right]$
int[] union(int[] $a_0$, int[] $b_0$) {
  int[] $u$ := new int[$|a_0| + |b_0|$];
  int $i$ := 0, $j$ := 0;
  for @ $\top$
    (int $k = 0$; $k < |u|$; $k := k + 1$) {
    if ($i \ge |a_0|$) {
      $u[k] := b_0[j]$;
      $j := j + 1$;
    }
    else if ($j \ge |b_0|$) {
      $u[k] := a_0[i]$;
      $i := i + 1$;
    }
    else if ($a_0[i] \le b_0[j]$) {
      $u[k] := a_0[i]$;
      $i := i + 1$;
    }
    else {
      $u[k] := b_0[j]$;
      $j := j + 1$;
    }
  }
  return $u$;
}
```

Fig. 6.12. Function union of the sorted set implementation

10

Combining Decision Procedures

The expressions which arise in program manipulation often do not fall within any... naturally defined theories — they usually involve mixed *terms containing functions and predicates from several theories.*

— Greg Nelson and Derek C. Oppen
Simplification by Cooperating Decision Procedures, 1979

Chapters 7–9 consider decision procedures for theories that each formalize just one data type. Yet almost all formulae in Chapter 5 are formulae of union theories. For example, many assert facts in $T_{\mathbb{Z}} \cup T_{\mathsf{A}}$ about arrays of integers indexed by integers. Additionally, the decision procedure for the array property fragment of $T_{\mathsf{A}}^{\mathbb{Z}}$ that we discuss in Chapter 11 requires a procedure for the quantifier-free fragment of $T_{\mathbb{Z}} \cup T_{\mathsf{A}}$. Can we reuse the decision procedures of Chapters 7–9 to decide satisfiability of formulae in union theories, or must we invent a new procedure for each combination?

Fortunately, there is a general result for quantifier-free fragments of union theories that allows us to reuse the procedures. This chapter discusses the **Nelson-Oppen combination method** for constructing decision procedures for union theories from decision procedures for individual theories. Section 10.1 introduces the method and discusses its limitations. Then Section 10.2 presents a nondeterministic version, for which correctness is proved in Section 10.4; and Section 10.3 presents the more practical deterministic version.

In this chapter, decision procedures for individual theories apply just to quantifier-free fragments. We rely on Cooper's method with all optimizations for considering quantifier-free $\Sigma_{\mathbb{Z}}$-formulae. Procedures for the other theories already apply only to their quantifier-free fragments.

10.1 Combining Decision Procedures

Consider two theories T_1 and T_2 over signatures Σ_1 and Σ_2, respectively. For the quantifier-free fragments of T_1 and T_2, we have decision procedures P_1 and P_2. How do we decide satisfiability in the quantifier-free fragment of $T_1 \cup T_2$?

Example 10.1. Consider the $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F : \ 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2) \ .$$

Chapter 9 describes a decision procedure for T_{E}, while Chapter 7 presents a decision procedure for $T_{\mathbb{Z}}$. We would like to combine these decision procedures to decide the $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-satisfiability of F and other quantifier-free $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formulae. ■

The **Nelson-Oppen combination method** (**N-O** method) combines decision procedures for the quantifier-free fragments of several theories into one decision procedure for the quantifier-free fragment of the union theory. In our presentation of the N-O method, we usually discuss combining two theories and their decision procedures; however, the N-O method can combine an arbitrary number of theories and procedures. Additionally, we restrict ourselves to considering conjunctive formulae; however, the satisfiability of arbitrary (quantifier-free) formulae can be considered by converting to DNF and checking each disjunct.

Besides being restricted to quantifier-free formulae, the N-O method has two additional restrictions. First, the signatures Σ_1 and Σ_2 can only share equality $=$:

$$\Sigma_1 \cap \Sigma_2 = \{=\} \ .$$

Second, the theories T_1 and T_2 must be stably infinite.

A theory T with signature Σ is **stably infinite** if for every quantifier-free Σ-formula F, if F is T-satisfiable, then there exists some T-interpretation that satisfies F and has a domain of infinite cardinality. We illustrate this concept with two example theories.

Example 10.2. Consider the theory $T_{a,b}$ with signature

$$\Sigma_2 : \ \{a, \ b, \ =\} \ ,$$

where both a and b are constants, and axiom

1. $\forall x. \ x = a \ \vee \ x = b$ (two)

Because of axiom (two), every $T_{a,b}$-interpretation $I : (D_I, \alpha_I)$ is such that the domain D_I has at most two elements: $|D_I| \leq 2$. Hence, $T_{a,b}$ is not stably infinite. ■

Example 10.3. We prove that T_{E} is stably infinite. Consider the T_{E}-satisfiable quantifier-free Σ_{E}-formula F with arbitrary satisfying T_{E}-interpretation $I : (D_I, \alpha_I)$ in which α_I maps $=$ to $=_I$. Let A be any infinite set disjoint from D_I. Then construct new interpretation $J : (D_J, \alpha_J)$:

- $D_J = D_I \cup A$

- $\alpha_J = \{= \mapsto =_J,\ \ldots\}$, where for $\mathsf{v}_1, \mathsf{v}_2 \in D_J$,

$$\mathsf{v}_1 =_J \mathsf{v}_2 \ \overset{\text{def}}{=} \begin{cases} \mathsf{v}_1 =_I \mathsf{v}_2 & \text{if } \mathsf{v}_1, \mathsf{v}_2 \in D_I \\ \top & \text{if } \mathsf{v}_1 \text{ is the same element as } \mathsf{v}_2 \\ \bot & \text{otherwise} \end{cases}$$

J is a T_{E}-interpretation satisfying F with infinite domain. Hence, T_{E} is stably infinite. ■

The other theories discussed in this book are also stably infinite.

Example 10.4. Consider the quantifier-free conjunctive $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F:\ 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2) \ .$$

The signatures of T_{E} and $T_{\mathbb{Z}}$ only share $=$. Also, both theories are stably infinite. Hence, the N-O combination of the decision procedures for T_{E} and $T_{\mathbb{Z}}$ decides the $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-satisfiability of F.

Intuitively, F is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-unsatisfiable. For the first two literals imply $x = 1 \ \vee \ x = 2$ so that $f(x) = f(1) \ \vee \ f(x) = f(2)$. Yet the last two literals contradict this conclusion. ■

10.2 Nelson-Oppen Method: Nondeterministic Version

In this section, we discuss the nondeterministic version of the N-O method. While simple to present, it suffers from high complexity. Section 10.3 reformulates the method to be deterministic and efficient.

Consider a quantifier-free conjunctive $(\Sigma_1 \cup \Sigma_2)$-formula F. The N-O method proceeds in two steps.

10.2.1 Phase 1: Variable Abstraction

The variable abstraction phase transforms a quantifier-free conjunctive formula F into two quantifier-free conjunctive formulae, a Σ_1-formula F_1 and a Σ_2-formula F_2, such that F and $F_1 \wedge F_2$ are $(T_1 \cup T_2)$-equisatisfiable. That is, F is $(T_1 \cup T_2)$-satisfiable iff $F_1 \wedge F_2$ is $(T_1 \cup T_2)$-satisfiable. F_1 and F_2 are linked via a set of shared variables.

For term t, let $\mathsf{hd}(t)$ be the root symbol; *e.g.*, $\mathsf{hd}(f(x)) = f$. Then for $i, j \in \{1, 2\}$ and $i \neq j$, repeat the following transformations as long as possible:

1. if function $f \in \Sigma_i$ and $\mathsf{hd}(t) \in \Sigma_j$,

$$F[f(t_1,\ \ldots,\ t,\ \ldots,\ t_n)] \quad \Longrightarrow \quad F[f(t_1,\ \ldots,\ w,\ \ldots,\ t_n)] \ \wedge \ w = t$$

2. if predicate $p \in \Sigma_i$ and $\mathsf{hd}(t) \in \Sigma_j$,

$$F[p(t_1,\ \ldots,\ t,\ \ldots,\ t_n)] \quad \Longrightarrow \quad F[p(t_1,\ \ldots,\ w,\ \ldots,\ t_n)] \ \wedge \ w = t$$

3. if $\mathsf{hd}(s) \in \Sigma_i$ and $\mathsf{hd}(t) \in \Sigma_j$,

$$F[s = t] \quad \Longrightarrow \quad F[w = t] \ \wedge \ w = s$$

w is a fresh variable in each application of a transformation. Transformation 3 also applies to $s \neq t$ literals: replace $F[s \neq t]$ with $F[w \neq t] \ \wedge \ w = s$.

After applying the transformations, each literal of the resulting formula falls entirely within the signature of one of the two theories (or possibly within each if it is just an equality $x = y$ or a disequality $x \neq y$ between variables: such literals are in every signature since they do not have symbols other than $=$). Divide the literals into two sets, one for each theory. These sets are not disjoint when there is a literal that is an equality or disequality between variables. Then return the conjunction of each set.

Example 10.5. Consider $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F: \ 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2) \ .$$

Since $f \in \Sigma_{=}$ and $1 \in \Sigma_{\mathbb{Z}}$, replace $f(1)$ by $f(w_1)$ and add $w_1 = 1$ by transformation 1. Similarly, replace $f(2)$ by $f(w_2)$ and add $w_2 = 2$.

Now, the literals

$$1 \leq x, \ x \leq 2, \ w_1 = 1, \text{ and } w_2 = 2$$

are $T_{\mathbb{Z}}$-literals, while the literals

$$f(x) \neq f(w_1) \text{ and } f(x) \neq f(w_2)$$

are T_{E}-literals. Hence, construct the $\Sigma_{\mathbb{Z}}$-formula

$$F_{\mathbb{Z}}: \ 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ w_1 = 1 \ \wedge \ w_2 = 2$$

and the Σ_{E}-formula

$$F_{\mathsf{E}}: \ f(x) \neq f(w_1) \ \wedge \ f(x) \neq f(w_2) \ .$$

$F_{\mathbb{Z}}$ and F_{E} share the variables x, w_1, and w_2. $F_{\mathbb{Z}} \wedge F_{\mathsf{E}}$ is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-equisatisfiable to F. ■

Example 10.6. Consider the $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F: \ f(x) = x + y \ \wedge \ x \leq y + z \ \wedge \ x + z \leq y \ \wedge \ y = 1 \ \wedge \ f(x) \neq f(2) \ .$$

Intuitively, F is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-satisfiable: consider an interpretation in which $x = 0$, $y = 1$, $z = 1$, $f(0) = 1$, and $f(2) = 2$.

In the first literal, $\mathsf{hd}(f(x)) = f \in \Sigma_{\mathsf{E}}$ and $\mathsf{hd}(x + y) = + \in \Sigma_{\mathbb{Z}}$; thus, by transformation 3, replace the literal with

$$w_1 = x + y \ \wedge \ w_1 = f(x) \ .$$

In the last literal, $f \in \Sigma_{\mathsf{E}}$ but $2 \in \Sigma_{\mathbb{Z}}$, so by transformation 1, replace it with

$$f(x) \neq f(w_2) \ \wedge \ w_2 = 2 \ .$$

Now, separating the literals results in two formulae:

$$F_{\mathbb{Z}} : \ w_1 = x + y \ \wedge \ x \leq y + z \ \wedge \ x + z \leq y \ \wedge \ y = 1 \ \wedge \ w_2 = 2$$

is a $\Sigma_{\mathbb{Z}}$-formula, and

$$F_{\mathsf{E}} : \ w_1 = f(x) \ \wedge \ f(x) \neq f(w_2)$$

is a Σ_{E}-formula. The conjunction $F_{\mathbb{Z}} \wedge F_{\mathsf{E}}$ is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-equisatisfiable to F. ■

10.2.2 Phase 2: Guess and Check

Phase 1 separates $(\Sigma_1 \cup \Sigma_2)$-formula F into two formulae, Σ_1-formula F_1, and Σ_2-formula F_2. F_1 and F_2 are linked by a set of shared variables. Let

$$V = \mathsf{shared}(F_1, F_2) = \mathsf{free}(F_1) \cap \mathsf{free}(F_2)$$

be the shared variables of F_1 and F_2. Let E be an equivalence relation over V. The **arrangement** $\alpha(V, E)$ of V induced by E is the formula

$$\alpha(V, E) : \bigwedge_{u,v \ \in \ V. \ uEv} u = v \ \wedge \bigwedge_{u,v \ \in \ V. \ \neg(uEv)} u \neq v \ ,$$

which asserts that variables related by E are equal and that variables unrelated by E are not equal. The formula F is $(T_1 \cup T_2)$-satisfiable iff there exists an equivalence relation E of V such that

- $F_1 \wedge \alpha(V, E)$ is T_1-satisfiable, and
- $F_2 \wedge \alpha(V, E)$ is T_2-satisfiable.

Otherwise, F is $(T_1 \cup T_2)$-unsatisfiable.

Example 10.7. Consider $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F : \ 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(2) \ .$$

Phase 1 separates this formula into the $\Sigma_{\mathbb{Z}}$-formula

$$F_{\mathbb{Z}} : \ 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ w_1 = 1 \ \wedge \ w_2 = 2$$

and the Σ_{E}-formula

$$F_{\mathsf{E}} : \ f(x) \neq f(w_1) \ \wedge \ f(x) \neq f(w_2) \ ,$$

with

$$V = \mathsf{shared}(F_{\mathbb{Z}}, F_{\mathsf{E}}) = \{x, w_1, w_2\} \ .$$

There are 5 equivalence relations to consider, which we list by stating the partitions:

1. $\{\{x, w_1, w_2\}\}$, *i.e.*, $x = w_1 = w_2$: $F_{\mathsf{E}} \wedge \alpha(V, E)$ is T_{E}-unsatisfiable because it cannot be the case that both $x = w_1$ and $f(x) \neq f(w_1)$.
2. $\{\{x, w_1\}, \{w_2\}\}$, *i.e.*, $x = w_1$, $x \neq w_2$: $F_{\mathsf{E}} \wedge \alpha(V, E)$ is T_{E}-unsatisfiable because it cannot be the case that both $x = w_1$ and $f(x) \neq f(w_1)$.
3. $\{\{x, w_2\}, \{w_1\}\}$, *i.e.*, $x = w_2$, $x \neq w_1$: $F_{\mathsf{E}} \wedge \alpha(V, E)$ is T_{E}-unsatisfiable because it cannot be the case that both $x = w_2$ and $f(x) \neq f(w_2)$.
4. $\{\{x\}, \{w_1, w_2\}\}$, *i.e.*, $x \neq w_1$, $w_1 = w_2$: $F_{\mathbb{Z}} \wedge \alpha(V, E)$ is $T_{\mathbb{Z}}$-unsatisfiable because it cannot be the case that both $w_1 = w_2$ and $w_1 = 1 \ \wedge \ w_2 = 2$.
5. $\{\{x\}, \{w_1\}, \{w_2\}\}$, *i.e.*, $x \neq w_1$, $x \neq w_2$, $w_1 \neq w_2$: $F_{\mathbb{Z}} \wedge \alpha(V, E)$ is $T_{\mathbb{Z}}$-unsatisfiable because it cannot be the case that both $x \neq w_1 \ \wedge \ x \neq w_2$ and $x = w_1 = 1 \vee x = w_2 = 2$ (since $1 \leq x \leq 2$ implies that $x = 1 \vee x = 2$ in $T_{\mathbb{Z}}$).

Hence, F is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-unsatisfiable. ■

Example 10.8. Consider the $(\Sigma_{\mathsf{cons}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F : \ \mathsf{car}(x) + \mathsf{car}(y) = z \ \wedge \ \mathsf{cons}(x, z) \neq \mathsf{cons}(y, z) \ .$$

After two applications of transformation 1, Phase 1 separates F into the Σ_{cons}-formula

$$F_{\mathsf{cons}} : \ w_1 = \mathsf{car}(x) \ \wedge \ w_2 = \mathsf{car}(y) \ \wedge \ \mathsf{cons}(x, z) \neq \mathsf{cons}(y, z)$$

and the $\Sigma_{\mathbb{Z}}$-formula

$$F_{\mathbb{Z}} : \ w_1 + w_2 = z \ ,$$

with

$$V = \mathsf{shared}(F_{\mathsf{cons}}, F_{\mathbb{Z}}) = \{z, w_1, w_2\} \ .$$

Consider the equivalence relation E given by the partition

$$\{\{z\}, \{w_1\}, \{w_2\}\} \ .$$

The arrangement

$$\alpha(V, E) : \ z \neq w_1 \ \wedge \ z \neq w_2 \ \wedge \ w_1 \neq w_2$$

satisfies both F_{cons} and $F_{\mathbb{Z}}$: $F_{\mathsf{cons}} \wedge \alpha(V, E)$ is T_{cons}-satisfiable, and $F_{\mathbb{Z}} \wedge \alpha(V, E)$ is $T_{\mathbb{Z}}$-satisfiable. Hence, F is $(T_{\mathsf{cons}} \cup T_{\mathbb{Z}})$-satisfiable. ■

10.2.3 Practical Efficiency

Phase 2 is formulated as "guess and check": first, guess an equivalence relation E, then check the induced arrangement. Unfortunately, the number of equivalence relations increases significantly with the number of shared variables. The number of equivalence relations is given by the sequence of **Bell numbers**, which grows super-exponentially. For example, just 12 shared variables induce over four million equivalence relations. Hence, the guess-and-check method is impractical.

However, there is no need to guess the entire equivalence relation at once; instead, construct it incrementally, as the following example illustrates:

Example 10.9. In Example 10.6, Phase 1 separates the $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F:\ f(x) = x + y \ \wedge\ x \le y + z \ \wedge\ x + z \le y \ \wedge\ y = 1 \ \wedge\ f(x) \ne f(2)$$

into $\Sigma_{\mathbb{Z}}$-formula

$$F_{\mathbb{Z}}:\ w_1 = x + y \ \wedge\ x \le y + z \ \wedge\ x + z \le y \ \wedge\ y = 1 \ \wedge\ w_2 = 2$$

and Σ_{E}-formula

$$F_{\mathsf{E}}:\ w_1 = f(x) \ \wedge\ f(x) \ne f(w_2)$$

Then

$$V = \mathsf{shared}(F_{\mathbb{Z}}, F_{\mathsf{E}}) = \{x, w_1, w_2\}\ .$$

We attempt to construct an arrangement.

1. Suppose $x = w_1$. But then $w_1 = x + y$ of $F_{\mathbb{Z}}$ implies that $y = 0$, yet $F_{\mathbb{Z}}$ asserts that $y = 1$. Hence, $x \ne w_1$.
2. $F_{\mathbb{Z}} \ \wedge\ x \ne w_1$ and $F_{\mathsf{E}} \ \wedge\ x \ne w_1$ are $T_{\mathbb{Z}}$- and T_{E}-satisfiable, respectively.
3. Suppose $x = w_2$. But $f(x) \ne f(w_2)$ of F_{E} contradicts this supposition. Hence, $x \ne w_2$.
4. $F_{\mathbb{Z}} \ \wedge\ x \ne w_1 \ \wedge\ x \ne w_2$ and $F_{\mathsf{E}} \ \wedge\ x \ne w_1 \ \wedge\ x \ne w_2$ are $T_{\mathbb{Z}}$- and T_{E}-satisfiable, respectively.
5. Suppose $w_1 = w_2$. No contradiction exists.

We discovered the arrangement

$$x \ne w_1 \ \wedge\ x \ne w_2 \ \wedge\ w_1 = w_2\ ,$$

so F is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-satisfiable. ■

Readers interested in implementing a simple Nelson-Oppen-based decision procedure could consider this incremental-construction “optimization” of the nondeterministic method. However, in practice, implementations are based on the deterministic method described in the next section.

10.3 Nelson-Oppen Method: Deterministic Version

Phase 1 of the deterministic version is the same as in the nondeterministic version.

Phase 2 of the nondeterministic method (both the guess-and-check method and the optimized incremental construction) proposes a set of equalities and disequalities and then lets each decision procedure P_i check the set with the corresponding formula F_i. In contrast, Phase 2 of the deterministic version asks the decision procedures P_1 and P_2 to propagate information in the form of new equalities.

A **convex** theory is particularly well-suited for propagating equalities. Section 10.3.1 discusses convex theories. Then Section 10.3.2 presents the deterministic Nelson-Oppen method.

10.3.1 Convex Theories

If a conjunctive formula in a convex theory implies a disjunction of equalities between variables, then it actually implies a single equality. Formally, consider a quantifier-free conjunctive Σ-formula F and a disjunction

$$G: \bigvee_{i=1}^{n} u_i = v_i \ , \tag{10.1}$$

for variables u_i and v_i. Theory T is **convex** if for every such F and G, if

$$F \ \Rightarrow \ \bigvee_{i=1}^{n} u_i = v_i$$

then

$$F \ \Rightarrow \ u_i = v_i \quad \text{for some } i \in \{1, \ldots, n\} \ .$$

If F implies G, then F actually implies one of the disjuncts of G.

Intuitively, F cannot be “covered” by any disjunction of equalities — no matter how many — if no single equality covers F (F is covered by a formula if F implies it). This intuition is especially apparent for vector spaces (Section 8.2): a plane cannot be covered by a finite disjunction of lines; it cannot even be covered by a finite disjunction of other planes unless at least one of the planes is the plane itself.

Example 10.10. The theory of integers $T_{\mathbb{Z}}$ is not convex. For consider the quantifier-free conjunctive $\Sigma_{\mathbb{Z}}$-formula

$$F: \ 1 \leq z \ \wedge \ z \leq 2 \ \wedge \ u = 1 \ \wedge \ v = 2 \ .$$

Then

$$F \;\Rightarrow\; z = u \;\vee\; z = v \;,$$

but neither

$$F \;\Rightarrow\; z = u \quad \text{nor} \quad F \;\Rightarrow\; z = v \;.$$

■

Example 10.11. The theory of arrays T_{A} is not convex. For consider the quantifier-free conjunctive Σ_{A}-formula

$$F:\; a\langle i \triangleleft v\rangle[j] = v \;.$$

Then

$$F \;\Rightarrow\; i = j \;\vee\; a[j] = v \;,$$

but neither

$$F \;\Rightarrow\; i = j \quad \text{nor} \quad F \;\Rightarrow\; a[j] = v \;.$$

■

Example 10.12. ★ The theory of rationals $T_{\mathbb{Q}}$ is convex, as it is convex in a geometric sense (see Chapter 8).

Each equality $u_i = v_i$ of the disjunction G of (10.1) is geometrically convex, but G itself is not. Consider, for example,

$$H:\; x = y \;\vee\; x = z \;.$$

Let S_H be the set of points satisfying H. The point $(x, y, z) = (0, 0, 1)$ is included in S_H, as is the point $(1, 0, 1)$. However, the average of the two points, $(\frac{1}{2}, 0, 1)$ (choosing $\lambda = \frac{1}{2}$), is not in S_H. Indeed, choose any two points

$$(u, u, v_1) \quad \text{and} \quad (w, v_2, w)$$

from $S_{x=y}$ and $S_{x=z}$, respectively, such that neither is in their intersection $S_{x=y=z}$ (*i.e.*, $v_1 \neq u$ and $v_2 \neq w$). Then for any $\lambda \in (0, 1)$, the point

$$(\lambda u + (1 - \lambda)w,\; \lambda u + (1 - \lambda)v_2,\; \lambda v_1 + (1 - \lambda)w)$$

is neither in $S_{x=y}$ nor in $S_{x=z}$.

Suppose, then, that $F \Rightarrow G : \bigvee_{i=1}^{n} u_i = v_i$, but for no $i \in \{1, \ldots, n\}$ does $F \Rightarrow u_i = v_i$. Then it must be the case that there are two points s_1 and s_2 of S_F in separate subsets $S_{u_i=v_i}$, $S_{u_j=v_j}$, $i \neq j$, of S_G. By the argument above, the points on the line segment between s_1 and s_2 are not in S_G and thus not in S_F. Then F is not geometrically convex, a contradiction.

Thus, $T_{\mathbb{Q}}$ is convex. ■

Exercise 10.5 asks the reader to prove that the theories T_{E} and T_{cons} are also convex.

10.3.2 Phase 2: Equality Propagation

Recall that the nondeterministic version guesses an equivalence relation E over the shared variables V and checks that both $F_1 \wedge \alpha(V, E)$ is T_1-satisfiable and $F_2 \wedge \alpha(V, E)$ is T_2-satisfiable. If it finds a satisfying equivalence relation E, it declares that F is $(T_1 \cup T_2)$-satisfiable. This method suffers from the enormous number of equivalence relations that are possible even over small sets of shared variables. In the deterministic version, a central manager asks the decision procedures P_1 and P_2 to report any new implied equalities between shared variables. It then adds this new information to the already discovered equalities and propagates it to the other decision procedure. This method is efficient.

In the context of already discovered equalities $\mathcal{E}$, a decision procedure P_i for a convex theory T_i discovers a new equality $u = v$, for shared variables u and v, when

$$F_i \ \wedge\ \mathcal{E} \ \Rightarrow\ u = v\ .$$

The central manager then propagates this new equality to the other decision procedure.

If T_j is not convex, P_j discovers a new disjunction of equalities S when

$$F_j \ \wedge\ \mathcal{E} \ \Rightarrow \bigvee_{u_i = v_i \ \in\ S} (u_i = v_i)\ ,$$

for shared variables u_i and v_i. In this case, the central manager must split the disjunction and search along multiple branches. Each branch assumes one of the disjuncts. The search along a branch ends either when a full arrangement is discovered (so the original formula is $(T_1 \cup T_2)$-satisfiable; see below) or when all sub-branches end in contradiction (T_i-unsatisfiability for some i). In the latter case, the central manager tries another branch. If no branches remain to try, then the central manager declares the original formula to be $(T_1 \cup T_2)$-unsatisfiable.

If at some point, neither P_1 nor P_2 finds a new equality (or a disjunction of equalities in the non-convex case), then the central manager concludes that the given formula is $(T_1 \cup T_2)$-satisfiable. For if $\mathcal{E}$ is the set of all learned equalities, S is the set of all possible remaining equalities, and

$$F_1 \ \wedge\ \mathcal{E} \ \not\Rightarrow \bigvee_{u_i = v_i \ \in\ S} (u_i = v_i) \quad \text{and} \quad F_2 \ \wedge\ \mathcal{E} \ \not\Rightarrow \bigvee_{u_i = v_i \ \in\ S} (u_i = v_i)\ ,$$

(which must hold when no new disjunctions of equalities are discovered), then

$$F_1 \ \wedge\ \mathcal{E}\ \wedge \bigwedge_{u_i = v_i \ \in\ S} (u_i \neq v_i) \quad \text{and} \quad F_2 \ \wedge\ \mathcal{E}\ \wedge \bigwedge_{u_i = v_i \ \in\ S} (u_i \neq v_i)$$

are T_1-satisfiable and T_2-satisfiable, respectively. Hence, the discovered arrangement is

$$\alpha(V, E) = \mathcal{E} \ \wedge \bigwedge_{u_i = v_i \ \in \ S} (u_i \neq v_i) \ ,$$

and F is $(T_1 \cup T_2)$-satisfiable.

Example 10.13. Consider the $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Q}})$-formula

$$F : \ f(f(x) - f(y)) \neq f(z) \ \wedge \ x \leq y \ \wedge \ y + z \leq x \ \wedge \ 0 \leq z \ .$$

F is $(T_{\mathsf{E}} \cup T_{\mathbb{Q}})$-unsatisfiable: the final three literals imply that $z = 0$ and $x = y$, so that $f(x) = f(y)$. But then from the first literal, $f(0) \neq f(0)$ since both $f(x) - f(y)$ and z equal 0.

Phase 1 separates F into two formulae. According to transformation 1, it replaces $f(x)$ by u, $f(y)$ by v, and $u - v$ by w, resulting in Σ_{E}-formula

$$F_{\mathsf{E}} : \ f(w) \neq f(z) \ \wedge \ u = f(x) \ \wedge \ v = f(y)$$

and $\Sigma_{\mathbb{Q}}$-formula

$$F_{\mathbb{Q}} : \ x \leq y \ \wedge \ y + z \leq x \ \wedge \ 0 \leq z \ \wedge \ w = u - v \ ,$$

with

$$V = \mathsf{shared}(F_{\mathsf{E}}, F_{\mathbb{Q}}) = \{x, y, z, u, v, w\} \ .$$

Recall that T_{E} and $T_{\mathbb{Q}}$ are convex theories. The decision procedure $P_{\mathbb{Q}}$ for $T_{\mathbb{Q}}$ discovers

$$F_{\mathbb{Q}} \ \Rightarrow \ x = y$$

from $x \leq y \ \wedge \ y + z \leq x \ \wedge \ 0 \leq z$, so

$$\mathcal{E}_1 : \ x = y \ .$$

Then P_{E} discovers the new congruence $f(x) = f(y)$ from $x = y$, so that

$$F_{\mathsf{E}} \ \wedge \ \mathcal{E}_1 \ \Rightarrow \ u = v \ ,$$

yielding

$$\mathcal{E}_2 : \ x = y \ \wedge \ u = v \ .$$

But then

$$F_{\mathbb{Q}} \ \wedge \ \mathcal{E}_2 \ \Rightarrow \ z = w$$

since $w = u - v = 0$, according to $u = v$, and $z = 0$. Propagating this equality back to P_{E} via

$$\mathcal{E}_3 : \ x = y \ \wedge \ u = v \ \wedge \ z = w$$

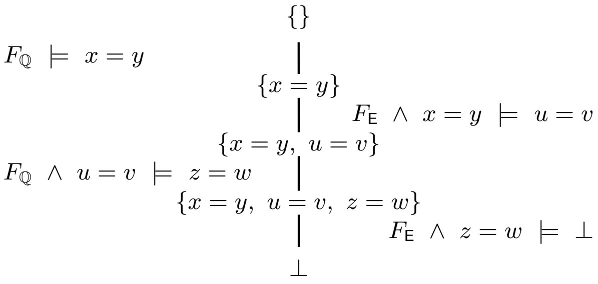
$\{\}$
$F_{\mathbb{Q}} \models x = y$
$\{x = y\}$
$F_{\mathsf{E}} \wedge x = y \models u = v$
$\{x = y,\ u = v\}$
$F_{\mathbb{Q}} \wedge u = v \models z = w$
$\{x = y,\ u = v,\ z = w\}$
$F_{\mathsf{E}} \wedge z = w \models \bot$
$\bot$

Fig. 10.1. Summary of Example 10.13

reveals the contradiction

$$F_{\mathsf{E}} \wedge \mathcal{E}_3 \Rightarrow \bot \ ;$$

in particular, $z = w$ contradicts $f(w) \neq f(z)$. Therefore, F is $(T_{\mathsf{E}} \cup T_{\mathbb{Q}})$-unsatisfiable.

Since both T_{E} and $T_{\mathbb{Q}}$ are convex, no case splitting was required.

Figure 10.1 summarizes this argument. The left and right halves list deductions made in $T_{\mathbb{Q}}$ and T_{E}, respectively. The sets in the middle are the deduced sets of shared equalities. The deductions terminate with $\bot$, indicating that F is $(T_{\mathsf{E}} \cup T_{\mathbb{Q}})$-unsatisfiable. ■

Example 10.14. Consider the $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F:\ 1 \leq x \ \wedge\ x \leq 2 \ \wedge\ f(x) \neq f(1) \ \wedge\ f(x) \neq f(2)\ .$$

While T_{E} is convex, $T_{\mathbb{Z}}$ is not. Thus, we should expect some case splits.

According to transformation 1, Phase 1 replaces $f(1)$ by $f(w_1)$ and $f(2)$ by $f(w_2)$, resulting in the $\Sigma_{\mathbb{Z}}$-formula

$$F_{\mathbb{Z}}:\ 1 \leq x \ \wedge\ x \leq 2 \ \wedge\ w_1 = 1 \ \wedge\ w_2 = 2$$

and the Σ_{E}-formula

$$F_{\mathsf{E}}:\ f(x) \neq f(w_1) \ \wedge\ f(x) \neq f(w_2)\ ,$$

with

$$V = \mathsf{shared}(F_{\mathbb{Z}}, F_{\mathsf{E}}) = \{x, w_1, w_2\}\ .$$

Immediately, $P_{\mathbb{Z}}$ recognizes that

$$F_{\mathbb{Z}} \Rightarrow x = w_1 \ \vee\ x = w_2\ ,$$

since $1 \leq x \leq 2$ implies that either $x = 1$ or $x = 2$. Hence, case split on these two disjuncts. For the first case, propagate

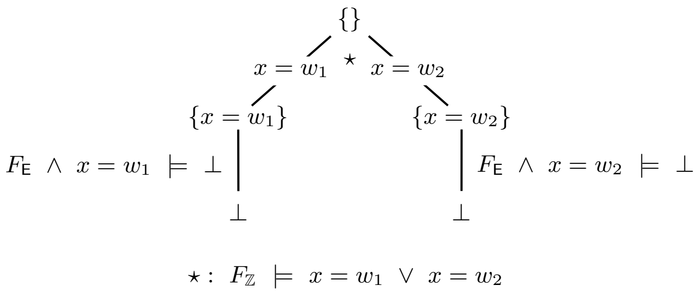
$\{\}$
$x = w_1$
$\star$
$x = w_2$
$\{x = w_1\}$
$\{x = w_2\}$
$F_{\mathsf{E}} \wedge x = w_1 \models \bot$
$F_{\mathsf{E}} \wedge x = w_2 \models \bot$
$\bot$
$\bot$
$\star : F_{\mathbb{Z}} \models x = w_1 \vee x = w_2$

Fig. 10.2. Summary of Example 10.14

$$\mathcal{E}_1^a : \ x = w_1$$

to P_{E}, which discovers that

$$F_{\mathsf{E}} \ \wedge \ \mathcal{E}_1^a \ \Rightarrow \ \bot \ ,$$

as $x = w_1$ contradicts $f(x) \neq f(w_1)$.

For the second case,

$$\mathcal{E}_1^b : \ x = w_2 \ .$$

Again, P_{E} discovers that

$$F_{\mathsf{E}} \ \wedge \ \mathcal{E}_1^b \ \Rightarrow \ \bot \ ,$$

as $x = w_2$ contradicts $f(x) \neq f(w_2)$.

As all branches end in contradiction, F is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-unsatisfiable.

Figure 10.2 summarizes this argument. Unlike in Example 10.13 and Figure 10.1, the nonconvexity of $T_{\mathbb{Z}}$ causes the argument to branch along two possibilities. Each branch ends in a contradiction. ■

Example 10.15. Consider the $(\Sigma_{\mathsf{E}} \cup \Sigma_{\mathbb{Z}})$-formula

$$F : \ 1 \leq x \ \wedge \ x \leq 3 \ \wedge \ f(x) \neq f(1) \ \wedge \ f(x) \neq f(3) \ \wedge \ f(1) \neq f(2) \ .$$

Applying transformation 1 of Phase 1 three times produces the $\Sigma_{\mathbb{Z}}$-formula

$$F_{\mathbb{Z}} : \ 1 \leq x \ \wedge \ x \leq 3 \ \wedge \ w_1 = 1 \ \wedge \ w_2 = 2 \ \wedge \ w_3 = 3$$

and the Σ_{E}-formula

$$F_{\mathsf{E}} : \ f(x) \neq f(w_1) \ \wedge \ f(x) \neq f(w_3) \ \wedge \ f(w_1) \neq f(w_2) \ ,$$

with

$$V = \mathsf{shared}(F_{\mathbb{Z}}, F_{\mathsf{E}}) = \{x, w_1, w_2, w_3\} \ .$$

From $1 \leq x \leq 3$, $P_{\mathbb{Z}}$ discovers that

$$F_{\mathbb{Z}} \ \Rightarrow \ x = w_1 \ \vee \ x = w_2 \ \vee \ x = w_3 \ .$$

Recall that $T_{\mathbb{Z}}$ is not convex. On case

$$\mathcal{E}_1^a : \ x = w_1 \ ,$$

P_{E} finds that

$$F_{\mathsf{E}} \ \wedge \ \mathcal{E}_1^a \ \Rightarrow \ \bot$$

because of $f(x) \neq f(w_1)$. On case

$$\mathcal{E}_1^b : \ x = w_2 \ ,$$

neither $P_{\mathbb{Z}}$ nor P_{E} discovers any contradiction or new equality. That is,

$$F_{\mathbb{Z}} \ \wedge \ \mathcal{E}_1^b \ \not\Rightarrow \ x = w_1 \ \vee \ x = w_3 \ \vee \ w_1 = w_2 \ \vee \ w_1 = w_3 \ \vee \ w_2 = w_3$$

and

$$F_{\mathsf{E}} \ \wedge \ \mathcal{E}_1^b \ \not\Rightarrow \ x = w_1 \ \vee \ x = w_3 \ \vee \ w_1 = w_2 \ \vee \ w_1 = w_3 \ \vee \ w_2 = w_3 \ ;$$

or, in other words,

$$F_{\mathbb{Z}} \ \wedge \ \mathcal{E}_1^b \ \wedge \ x \neq w_1 \ \wedge \ x \neq w_3 \ \wedge \ w_1 \neq w_2 \ \wedge \ w_1 \neq w_3 \ \wedge \ w_2 \neq w_3$$

is $T_{\mathbb{Z}}$-satisfiable, and

$$F_{\mathsf{E}} \ \wedge \ \mathcal{E}_1^b \ \wedge \ x \neq w_1 \ \wedge \ x \neq w_3 \ \wedge \ w_1 \neq w_2 \ \wedge \ w_1 \neq w_3 \ \wedge \ w_2 \neq w_3$$

is T_{E}-satisfiable. Thus, F is $(T_{\mathsf{E}} \cup T_{\mathbb{Z}})$-satisfiable.

Figure 10.3 summarizes this argument. The middle branch terminates with a satisfying arrangement. We did not actually explore the right branch. ■

10.3.3 Equality Propagation: Implementation

Equality propagation can be implemented somewhat efficiently without modifying the individual decision procedures. For convex theory T_j, test each possible equality $u_i = v_i$. Suppose that F_j is the Σ_j-formula constructed in Phase 1 and $\mathcal{E}$ is the conjunction of equalities discovered so far. Then check if any equality $u_i = v_i$ is implied:

$$F_j \ \wedge \ \mathcal{E} \ \Rightarrow \ u_i = v_i \ .$$

Any implied equality should be propagated to the other theories.

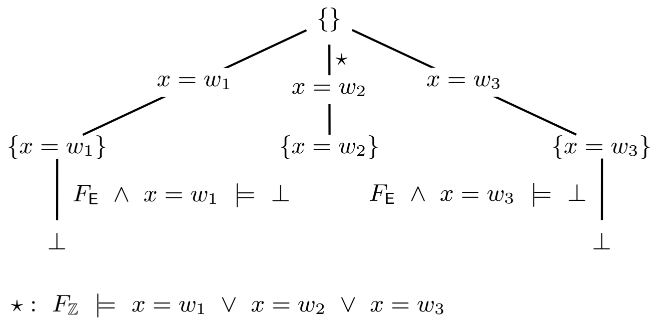
{}
★
$x = w_1$
$x = w_2$
$x = w_3$
$\{x = w_1\}$
$\{x = w_2\}$
$\{x = w_3\}$
$F_\mathsf{E} \wedge x = w_1 \models \bot$
$F_\mathsf{E} \wedge x = w_3 \models \bot$
$\bot$
$\bot$
$\star : F_\mathbb{Z} \models x = w_1 \vee x = w_2 \vee x = w_3$

Fig. 10.3. Summary of Example 10.15

This procedure is not applicable to a non-convex theory T_k. A procedure for a non-convex theory must be able to find disjunctions of equalities that are implied by a Σ_k-formula F_k. Moreover, the disjunctions should be as small as possible since the Nelson-Oppen method must branch on each disjunct. A disjunction is **minimal** if it is implied by F_k and if each smaller disjunction is not implied by F_k.

A simple procedure to find a minimal disjunction is based on the observation that any disjunction that contains a minimal disjunction — which is implied by F_k by definition — is also implied by F_k. Therefore, we can strip off extra disjuncts one-by-one. First, consider the disjunction of all equalities at once. If it is not implied, then no subset is implied either, so we are done. Otherwise, drop each equality in turn: if the remaining disjunction is still implied by F_k, continue with this smaller disjunction; otherwise, restore the equality and continue. When all equalities have been considered, the resulting disjunction is minimal. This procedure requires checking T_k-satisfiability $O(|V|^2)$ times, where V is the set of shared variables. Exercise 10.4 asks the reader to describe a procedure based on binary search that requires asymptotically fewer satisfiability checks when the final disjunction is small relative to the disjunction of all equalities.

10.4 ★Correctness of the Nelson-Oppen Method

In this section, we prove the correctness of the Nelson-Oppen combination method. We reason at the level of arrangements, which is more suited to the nondeterministic version of the method. However, Section 10.3 shows how to construct an arrangement in the deterministic version, as well, so the following proof can be extended to the deterministic version. We also focus on the second phase of the nondeterministic procedure, which chooses an arrangement if one exists. We thus assume that the variable abstraction phase is correct: it produces formulae F_1 and F_2 such that $F_1 \wedge F_2$ is $(T_1 \cup T_2)$-equivalent to the given $(\Sigma_1 \cup \Sigma_2)$-formula F.

A theory **has equality** (or is a theory **with equality**) if its signature includes the binary predicate $=$ and its axioms imply reflexivity, symmetry, and transitivity of equality. The **pure equality** fragment of a theory with equality is composed of formulae that are possibly quantified Boolean combinations of equalities between variables.

Theorem 10.16 (Sound & Complete). *Consider stably infinite theories T_1 and T_2 such that $\Sigma_1 \cap \Sigma_2 = \{=\}$. For conjunctive quantifier-free Σ_1-formula F_1 and conjunctive quantifier-free Σ_2-formula F_2, $F_1 \wedge F_2$ is $(T_1 \cup T_2)$-satisfiable iff there exists an arrangement $K = \alpha(\mathsf{shared}(F_1, F_2), E)$ such that $F_1 \wedge K$ is T_1-satisfiable and $F_2 \wedge K$ is T_2-satisfiable.*

Soundness is straightforward. Suppose that $F_1 \wedge F_2$ is $(T_1 \cup T_2)$-satisfiable with satisfying $(T_1 \cup T_2)$-interpretation I. Extract from I the equivalence relation E such that the arrangement $K = \alpha(\mathsf{shared}(F_1, F_2), E)$ is satisfied by I. Then $F_1 \wedge K$ and $F_2 \wedge K$ are both satisfied by I, which can be viewed as both a T_1-interpretation and a T_2-interpretation, so that they are T_1-satisfiable and T_2-satisfiable, respectively.

Completeness is more complicated. Let $K = \alpha(\mathsf{shared}(F_1, F_2), E)$ be an arrangement such that $F_1 \wedge K$ and $F_2 \wedge K$ are T_1-satisfiable and T_2-satisfiable, respectively. Suppose that $F_1 \wedge F_2$ is $(T_1 \cup T_2)$-unsatisfiable. We derive a contradiction.

The outline of the proof is the following. Because $F_1 \wedge F_2$ is $(T_1 \cup T_2)$-unsatisfiable, we know that F_1 implies $\neg F_2$ in $T_1 \cup T_2$. An adaptation of the **Craig Interpolation Lemma** (Theorem 2.38) tells us that there is a quantifier-free formula H such that F_1 implies H over all infinite T_1-interpretations (T_1-interpretations with infinite domains) and F_2 implies $\neg H$ over all infinite T_2-interpretations: H interpolates between F_1 and F_2. We then show that the arrangement K implies H, which means that F_2 implies $\neg K$ over all infinite T_2-interpretations. In other words, no infinite T_2-interpretation satisfies $F_2 \wedge K$. Yet if T_2 is stably infinite and $F_2 \wedge K$ is T_2-satisfiable as assumed, then $F_2 \wedge K$ is satisfied by some infinite T_2-interpretation, a contradiction.

We now present the details of the proof. First, because we are considering only stably infinite theories, we need only consider interpretations with infinite domains. For we can extend a T_1- or T_2-interpretation with a finite domain to a T_1- or T_2-interpretation with an infinite domain. Therefore, define $\Rightarrow^*$ as a weaker form of implication: $F \Rightarrow^* G$ iff G is true on every interpretation I that has an infinite domain and that satisfies F. Similarly, weaken $\Leftrightarrow$ to $\Leftrightarrow^*$. If $F \Rightarrow^* G$, we say that F **weakly implies** G; if $F \Leftrightarrow^* G$, we say that F is **weakly equivalent** to G.

Recall from Section 2.7.4 the following theorem.

Theorem 10.17 (Compactness Theorem). *A countable set of first-order formulae S is simultaneously satisfiable iff the conjunction of every finite subset is satisfiable.*

Since $F_1 \wedge F_2$ is $(T_1 \cup T_2)$-unsatisfiable, the Compactness Theorem tells us that there exist a conjunction S_1 of a finite subset of axioms of T_1 and a conjunction S_2 of a finite subset of axioms of T_2 such that $S_1 \wedge F_1 \wedge S_2 \wedge F_2$ is (first-order) unsatisfiable. Choose S_1 and S_2 to include the axioms that imply reflexivity, symmetry, and transitivity of equality. Then, rearranging, we have that

$$S_1 \;\wedge\; F_1 \;\Rightarrow\; \neg S_2 \;\vee\; \neg F_2 \;. \tag{10.2}$$

Recall from Section 2.7.4 the following theorem.

Theorem 10.18 (Craig Interpolation Lemma). *If $F_1 \Rightarrow F_2$, then there exists a formula H such that $F_1 \Rightarrow H$, $H \Rightarrow F_2$, and each free variable, function symbol, and predicate symbol of H appears in F_1 and F_2.*

Hence, from implication (10.2), there exists an interpolant H' such that $\mathsf{free}(H') = \mathsf{shared}(F_1, F_2)$ and

$$S_1 \;\wedge\; F_1 \;\Rightarrow\; H' \quad \text{and} \quad S_2 \;\wedge\; H' \;\Rightarrow\; \neg F_2 \;.$$

The latter implication is derived by rearranging $H' \Rightarrow \neg S_2 \vee \neg F_2$. Because $=$ is the only predicate or function shared between $S_1 \wedge F_1$ and $S_2 \wedge F_2$, H' is of a special form: its atoms are equalities between variables of $\mathsf{shared}(F_1, F_2)$. However, H' may have quantifiers. We prove next that in fact a "weak" quantifier-free interpolant H exists.

Lemma 10.19 (Weak Quantifier Elimination for Pure Equality). *Consider any stably infinite theory T with equality. For each pure equality formula F, there exists a quantifier-free pure equality formula F' such that F is weakly T-equivalent to F'.*

Proof. Consider pure equality formula $\exists x.\; G[x, \overline{y}]$, where G is quantifier-free with free variables x and $\overline{y}$. Define

$$G_0 :\;\; G\{x = x \mapsto \mathsf{true},\; x = y_1 \mapsto \mathsf{false},\; \ldots,\; x = y_n \mapsto \mathsf{false}\}$$

and, for $i \in \{1, \ldots, n\}$,

$$G_i :\;\; G\{x \mapsto y_i\} \;.$$

We claim that $\exists x.\; G$ is weakly T-equivalent to

$$G' :\;\; G_0 \;\vee\; G_1 \;\vee\; \cdots \;\vee\; G_n \;.$$

For G' asserts that x is either equal to some free variable y_i or not. Because we consider only interpretations with infinite domains, it is always possible for x not to equal any y_i.

By Section 7.1, we have a weak quantifier elimination procedure over the pure equality fragment of T. It is weak because equivalence is only guaranteed to hold on infinite interpretations. ■

Example 10.20. Consider the pure equality formula

$$F:\ x \neq y\ \wedge\ (\forall z.\ z = x\ \vee\ z = y)\ .$$

For eliminating z, consider the negation of the second conjunct,

$$G:\ \exists z.\ z \neq x\ \wedge\ z \neq y\ ,$$

for which we have

$$G_0:\ \neg\bot\ \wedge\ \neg\bot\ \Leftrightarrow\ \top$$

and

$$G_x:\ x \neq x\ \wedge\ x \neq y\ \Leftrightarrow\ \bot \qquad G_y:\ y \neq x\ \wedge\ y \neq y\ \Leftrightarrow\ \bot\ .$$

Then

$$G':\ G_0\ \vee\ G_x\ \vee\ G_y\ \Leftrightarrow\ \top\ .$$

Substituting into F, we have

$$x \neq y\ \wedge\ \neg(\top)\ \Leftrightarrow\ \bot\ .$$

Hence, over infinite interpretations satisfying the axioms of equality, F is equivalent to $\bot$.

However, in an interpretation with a two-element domain that satisfies the equality axioms, F is not equivalent to $\bot$, but rather to $x \neq y$. For if $x \neq y$ on such an interpretation, then every element is equal either to x or to y. ■

Continuing the main theorem, we claim that there exists a quantifier-free pure equality formula H over $\mathsf{shared}(F_1, F_2)$ such that

$$S_1\ \wedge\ F_1\ \Rightarrow^*\ H \quad \text{and} \quad S_2\ \wedge\ H\ \Rightarrow^*\ \neg F_2\ .$$

For by Lemma 10.19, a quantifier-free pure equality formula H exists such that H is weakly equivalent to the Craig interpolant H' in any stably infinite theory with equality.

For the next step, recall from the beginning of the proof that $F_1 \wedge K$ is T_1-satisfiable and $F_2 \wedge K$ is T_2-satisfiable, where $K = \alpha(\mathsf{shared}(F_1, F_2), E)$ is an arrangement. We thus know that

$$S_1\ \wedge\ F_1\ \wedge\ K \quad \text{and} \quad S_2\ \wedge\ F_2\ \wedge\ K$$

are (first-order) satisfiable. Moreover, as T_1 and T_2 are stably infinite, each of these formulae has an interpretation with an infinite domain.

Now, K is a conjunction of equalities and disequalities between pairs of variables of $\mathsf{shared}(F_1, F_2)$. Moreover, by the definition of an arrangement, K is as strong as possible: no additional equality literals L over $\mathsf{shared}(F_1, F_2)$

can be added to K without either K and $K \wedge L$ being equivalent in a theory with equality or $K \wedge L$ being unsatisfiable in a theory with equality. Based on this observation, construct the formula K' by conjoining additional equality literals: for each pair of variables $u, v \in \mathsf{shared}(F_1, F_2)$, conjoin either $u = v$ or $u \neq v$, depending on which maintains the satisfiability of K' in a theory with equality. Now, since $S_1 \wedge F_1 \wedge K$ is satisfiable, then so is $S_1 \wedge F_1 \wedge K'$, indeed by the same interpretations.

We claim that the DNF representation of H must include K' or a (conjunctive) subformula of K' as a disjunct. Suppose not; then every disjunct of the DNF representation of H contradicts the satisfying interpretations of $S_1 \wedge F_1 \wedge K'$, of which at least one exists. Therefore, $K' \Rightarrow H$, and — because K and K' are equivalent in a theory with equality — $K \Rightarrow H$. In other words, the discovered arrangement K is a special case of the weak interpolant H.

To finish, we have

$$S_2 \;\wedge\; H \;\Rightarrow^*\; \neg F_2 \;,$$

or, rearranging,

$$S_2 \;\wedge\; F_2 \;\Rightarrow^*\; \neg H \;.$$

From $K \Rightarrow H$, we have $\neg H \Rightarrow \neg K$, so

$$S_2 \;\wedge\; F_2 \;\Rightarrow^*\; \neg K \;.$$

But this weak implication contradicts that $S_2 \wedge F_2 \wedge K$ is satisfied by some infinite interpretation. Thus, $F_1 \wedge F_2$ is actually $(T_1 \cup T_2)$-satisfiable, and the Nelson-Oppen method is correct.

10.5 ★Complexity

Assume that T_1 and T_2 are stably infinite theories such that $\Sigma_1 \cap \Sigma = \{=\}$. Also, they have decision procedures P_1 and P_2 for their respective conjunctive quantifier-free fragments.

Theorem 10.21 (Complexity: Convex Theories). *If convex theories T_1 and T_2 have* PTIME *decision procedures P_1 and P_2, then the Nelson-Oppen combination based on equality propagation is a* PTIME *decision procedure for the conjunctive quantifier-free fragment of $T_1 \cup T_2$.*

Theorem 10.22 (Complexity: Non-Convex Theories). *If T_1 and T_2 have* NPTIME *decision procedures P_1 and P_2, then the Nelson-Oppen combination based on equality propagation is an* NPTIME *decision procedure for the conjunctive quantifier-free fragment of $T_1 \cup T_2$.*

10.6 Summary

Combining decision procedures in a general and efficient manner is crucial for most applications. This chapter covers the Nelson-Oppen combination method, in particular:

- The *nondeterministic Nelson-Oppen method.* Three requirements: the theories only share =; the theories are stably infinite; and the considered formula is quantifier-free. Variable abstraction, separation into theory-specific formulae. Shared variables, equivalence relations over shared variables, arrangements.
- The *deterministic Nelson-Oppen method.* Convex theories. Equality propagation.
- *Correctness* of the Nelson-Oppen method, which follows from the Craig Interpolation Lemma of Chapter 2.
- *Complexity.* When the individual decision procedures are convex and run in polynomial time, the combination procedure runs in polynomial time.

The Nelson-Oppen combination method provides a general means of reasoning simultaneously about the theories studied in this book using the individual decision procedures. Being able to reason in union theories is crucial. For example, almost all of the verification conditions of Chapters 5 and 6 are expressed in multiple signatures.

Bibliographic Remarks

Nelson and Oppen describe the Nelson-Oppen combination method [65]. Their original proof of correctness was flawed; Oppen presents a corrected proof in [70], and Nelson presents a corrected proof in [64]. Oppen also proves in [70] the complexity results that we state. Tinelli and Harandi present an alternate proof of correctness in [92]. Our correctness proof derives from that of Nelson and Oppen. See [56] for another presentation of the method and its correctness.

Another general combination method that has received much attention is that of Shostak [84]. See the work of Ruess and Shankar [78] for a correct presentation of the method.

Exercises

10.1 (DP for combinations). For each of the following formulae, identify the combination of theories in which it lies. To avoid ambiguity, prefer $T_{\mathbb{Z}}$ to $T_{\mathbb{Q}}$. Then apply the N-O method using the appropriate decision procedures. Use either the nondeterministic or deterministic version. Provide a level of detail as in the examples of the chapter.

(a) $1 \leq x \ \wedge \ x \leq 2 \ \wedge \ \mathsf{cons}(1, y) \neq \mathsf{cons}(x, y) \ \wedge \ \mathsf{cons}(2, y) \neq \mathsf{cons}(x, y)$
(b) $a[i] \geq 1 \ \wedge \ a[i] + x \leq 2 \ \wedge \ x > 0 \ \wedge \ x = i \ \wedge \ a\langle x \triangleleft 2\rangle[i] \neq 1$

10.2 (Deterministic N-O). Apply the deterministic N-O method to the following formulae. Prefer $T_{\mathbb{Q}}$ to $T_{\mathbb{Z}}$.

(a) $1 \leq x \ \wedge \ x \leq 2 \ \wedge \ \mathsf{cons}(1, y) \neq \mathsf{cons}(x, y) \ \wedge \ \mathsf{cons}(2, y) \neq \mathsf{cons}(x, y)$
(b) $x + y = z \ \wedge \ f(z) = z \ \wedge \ f(x + y) \neq z$
(c) $g(x + y, z) = f(g(x, y)) \ \wedge \ x + z = y \ \wedge \ z \geq 0 \ \wedge \ x \geq y$
$\wedge \ g(x, x) = z \ \wedge \ f(z) \neq g(2x, 0)$

10.3 (★Equality propagation in T_{E}). Section 10.3.3 explains general techniques for propagating equalities. However, some decision procedures are easily modified to propagate new equalities. Describe such a modification of the congruence closure algorithm of Chapter 9.

10.4 (★Equality propagation). Consider conjunctive Σ-formula F of nonconvex theory T and the disjunction of equalities

$$G : \bigvee_{i=1}^{n} u_i = v_i$$

such that $F \Rightarrow G$. Describe a procedure based on binary search that discovers a minimal disjunction G' of the equalities of G that is implied by F. If the procedure returns a disjunction with m equalities, then it should have invoked the decision procedure for T at most $O(m \lg n)$ times. *Hint*: The solution is related to the solution of Exercise 8.1*(e)*.

10.5 (★Convex theories). Prove that the following theories are convex:

(a) T_{E}
(b) T_{cons}

10.6 (★Complexity). Prove the complexity results about the N-O method.

(a) Theorem 10.21.
(b) Theorem 10.22.