

Second Edition

Computability, Complexity, and Languages

*Fundamentals of
Theoretical Computer
Science*

Martin D. Davis
Ron Sigal
Elaine J. Weyuker

Σ



Π



Second Edition

Computability, Complexity, and Languages

*Fundamentals of
Theoretical Computer Science*

Martin D. Davis

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
New York, New York

Ron Sigal

Departments of Mathematics and Computer Science
Yale University
New Haven, Connecticut

Elaine J. Weyuker

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
New York, New York



ACADEMIC PRESS
Harcourt, Brace & Company
Boston San Diego New York
London Sydney Tokyo Toronto

This book is printed on acid-free paper 

Copyright © 1994, 1983 by Academic Press, Inc.

All rights reserved

No part of this publication may be reproduced or
transmitted in any form or by any means, electronic
or mechanical, including photocopy, recording, or
any information storage and retrieval system, without
permission in writing from the publisher.

ACADEMIC PRESS, INC.
525 B Street, Suite 1900, San Diego, CA 92101-4495

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24-28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data

Davis, Martin 1928-

Computability, complexity, and languages: fundamentals of
theoretical computer science /Martin D. Davis, Ron Sigal,
Elaine J. Weyuker. --2nd ed.

p. cm. --(Computer science and applied mathematics)

Includes bibliographical references and index.

ISBN 0-12-206382-1

1. Machine theory. 2. Computational complexity. 3. Formal
languages. I. Sigal, Ron. II. Weyuker, Elaine J. III. Title.

IV. Series.

QA267.D38 1994

511.3—dc20

93-26807

CIP

Printed in the United States of America
94 95 96 97 98 BC 9 8 7 6 5 4 3 2 1

*To the memory of Helen and Harry Davis
and to*

*Hannah and Herman Sigal
Sylvia and Marx Weyuker*

*Virginia Davis, Dana Latch, Thomas Ostrand
and to*

Rachel Weyuker Ostrand

Contents

<i>Preface</i>	xiii
<i>Acknowledgments</i>	xvii
<i>Dependency Graph</i>	xix
1 Preliminaries	1
1. Sets and n -tuples	1
2. Functions	3
3. Alphabets and Strings	4
4. Predicates	5
5. Quantifiers	6
6. Proof by Contradiction	8
7. Mathematical Induction	9
Part 1 <i>Computability</i>	15
2 Programs and Computable Functions	17
1. A Programming Language	17
2. Some Examples of Programs	18
3. Syntax	25
4. Computable Functions	28
5. More about Macros	32

3 Primitive Recursive Functions	39
1. Composition	39
2. Recursion	40
3. PRC Classes	42
4. Some Primitive Recursive Functions	44
5. Primitive Recursive Predicates	49
6. Iterated Operations and Bounded Quantifiers	52
7. Minimalization	55
8. Pairing Functions and Gödel Numbers	59
4 A Universal Program	65
1. Coding Programs by Numbers	65
2. The Halting Problem	68
3. Universality	70
4. Recursively Enumerable Sets	78
5. The Parameter Theorem	85
6. Diagonalization and Reducibility	88
7. Rice's Theorem	95
*8. The Recursion Theorem	97
*9. A Computable Function That Is Not Primitive Recursive	105
5 Calculations on Strings	113
1. Numerical Representation of Strings	113
2. A Programming Language for String Computations	121
3. The Languages \mathcal{S} and \mathcal{S}_n	126
4. Post-Turing Programs	129
5. Simulation of \mathcal{S}_n in \mathcal{T}	135
6. Simulation of \mathcal{T} in \mathcal{S}	140
6 Turing Machines	145
1. Internal States	145
2. A Universal Turing Machine	152
3. The Languages Accepted by Turing Machines	153
4. The Halting Problem for Turing Machines	157
5. Nondeterministic Turing Machines	159
6. Variations on the Turing Machine Theme	162
7 Processes and Grammars	169
1. Semi-Thue Processes	169
2. Simulation of Nondeterministic Turing Machines by Semi-Thue Processes	171

3. Unsolvable Word Problems	176
4. Post's Correspondence Problem	181
5. Grammars	186
6. Some Unsolvable Problems Concerning Grammars	191
*7. Normal Processes	192
8 Classifying Unsolvable Problems	197
1. Using Oracles	197
2. Relativization of Universality	201
3. Reducibility	207
4. Sets r.e. Relative to an Oracle	211
5. The Arithmetic Hierarchy	215
6. Post's Theorem	217
7. Classifying Some Unsolvable Problems	224
8. Rice's Theorem Revisited	230
9. Recursive Permutations	231
Part 2 <i>Grammars and Automata</i>	235
9 Regular Languages	237
1. Finite Automata	237
2. Nondeterministic Finite Automata	242
3. Additional Examples	247
4. Closure Properties	249
5. Kleene's Theorem	253
6. The Pumping Lemma and Its Applications	260
7. The Myhill–Nerode Theorem	263
10 Context-Free Languages	269
1. Context-Free Grammars and Their Derivation Trees	269
2. Regular Grammars	280
3. Chomsky Normal Form	285
4. Bar-Hillel's Pumping Lemma	287
5. Closure Properties	291
*6. Solvable and Unsolvable Problems	297
7. Bracket Languages	301
8. Pushdown Automata	308
9. Compilers and Formal Languages	323

11 Context-Sensitive Languages	327
1. The Chomsky Hierarchy	327
2. Linear Bounded Automata	330
3. Closure Properties	337
Part 3 <i>Logic</i>	345
12 Propositional Calculus	347
1. Formulas and Assignments	347
2. Tautological Inference	352
3. Normal Forms	353
4. The Davis–Putnam Rules	360
5. Minimal Unsatisfiability and Subsumption	366
6. Resolution	367
7. The Compactness Theorem	370
13 Quantification Theory	375
1. The Language of Predicate Logic	375
2. Semantics	377
3. Logical Consequence	382
4. Herbrand’s Theorem	388
5. Unification	399
6. Compactness and Countability	404
*7. Gödel’s Incompleteness Theorem	407
*8. Unprovability of the Satisfiability Problem in Predicate Logic	410
Part 4 <i>Complexity</i>	417
14 Abstract Complexity	419
1. The Blum Axioms	419
2. The Gap Theorem	425
3. Preliminary Form of the Speedup Theorem	428
4. The Speedup Theorem Concluded	435
15 Polynomial-Time Computability	439
1. Rates of Growth	439
2. P versus NP	443
3. Cook’s Theorem	451
4. Other NP-Complete Problems	457

Part 5 <i>Semantics</i>	465
16 Approximation Orderings	467
1. Programming Language Semantics	467
2. Partial Orders	472
3. Complete Partial Orders	475
4. Continuous Functions	486
5. Fixed Points	494
17 Denotational Semantics of Recursion Equations	505
1. Syntax	505
2. Semantics of Terms	511
3. Solutions to W-Programs	520
4. Denotational Semantics of W-Programs	530
5. Simple Data Structure Systems	539
6. Infinitary Data Structure Systems	544
18 Operational Semantics of Recursion Equations	557
1. Operational Semantics for Simple Data Structure Systems	557
2. Computable Functions	575
3. Operational Semantics for Infinitary Data Structure Systems	584
<i>Suggestions for Further Reading</i>	593
<i>Notation Index</i>	595
<i>Index</i>	599

Preface

Theoretical computer science is the mathematical study of models of computation. As such, it originated in the 1930s, well before the existence of modern computers, in the work of the logicians Church, Gödel, Kleene, Post, and Turing. This early work has had a profound influence on the practical and theoretical development of computer science. Not only has the Turing machine model proved basic for theory, but the work of these pioneers presaged many aspects of computational practice that are now commonplace and whose intellectual antecedents are typically unknown to users. Included among these are the existence in principle of all-purpose (or universal) digital computers, the concept of a program as a list of instructions in a formal language, the possibility of interpretive programs, the duality between software and hardware, and the representation of languages by formal structures, based on productions. While the spotlight in computer science has tended to fall on the truly breathtaking technological advances that have been taking place, important work in the foundations of the subject has continued as well. It is our purpose in writing this book to provide an introduction to the various aspects of theoretical computer science for undergraduate and graduate students that is sufficiently comprehensive that the professional literature of treatises and research papers will become accessible to our readers.

We are dealing with a very young field that is still finding itself. Computer scientists have by no means been unanimous in judging which

parts of the subject will turn out to have enduring significance. In this situation, fraught with peril for authors, we have attempted to select topics that have already achieved a polished classic form, and that we believe will play an important role in future research.

In this second edition, we have included new material on the subject of programming language semantics, which we believe to be established as an important topic in theoretical computer science. Some of the material on computability theory that had been scattered in the first edition has been brought together, and a few topics that were deemed to be of only peripheral interest to our intended audience have been eliminated. Numerous exercises have also been added. We were particularly pleased to be able to include the answer to a question that had to be listed as open in the first edition. Namely, we present Neil Immerman's surprisingly straightforward proof of the fact that the class of languages accepted by linear bounded automata is closed under complementation.

We have assumed that many of our readers will have had little experience with mathematical proof, but that almost all of them have had substantial programming experience. Thus the first chapter contains an introduction to the use of proofs in mathematics in addition to the usual explanation of terminology and notation. We then proceed to take advantage of the reader's background by developing computability theory in the context of an extremely simple abstract programming language. By systematic use of a macro expansion technique, the surprising power of the language is demonstrated. This culminates in a universal program, which is written in all detail on a single page. By a series of simulations, we then obtain the equivalence of various different formulations of computability, including Turing's. Our point of view with respect to these simulations is that it should not be the reader's responsibility, at this stage, to fill in the details of vaguely sketched arguments, but rather that it is our responsibility as authors to arrange matters so that the simulations can be exhibited simply, clearly, and completely.

This material, in various preliminary forms, has been used with undergraduate and graduate students at New York University, Brooklyn College, The Scuola Matematica Interuniversitaria–Perugia, The University of California–Berkeley, The University of California–Santa Barbara, Worcester Polytechnic Institute, and Yale University.

Although it has been our practice to cover the material from the second part of the book on formal languages after the first part, the chapters on regular and on context-free languages can be read immediately after Chapter 1. The Chomsky–Schützenberger representation theorem for context-free languages is used to develop their relation to pushdown automata in a way that we believe is clarifying. Part 3 is an exposition of the aspects of logic that we think are important for computer science and can

also be read immediately following Chapter 1. Each of the chapters of Part 4 introduces an important theory of computational complexity, concluding with the theory of NP-completeness. Part 5, which is new to the second edition, uses recursion equations to expand upon the notion of computability developed in Part 1, with an emphasis on the techniques of formal semantics, both denotational and operational. Rooted in the early work of Gödel, Herbrand, Kleene, and others, Part 5 introduces ideas from the modern fields of functional programming languages, denotational semantics, and term rewriting systems.

Because many of the chapters are independent of one another, this book can be used in various ways. There is more than enough material for a full-year course at the graduate level on *theory of computation*. We have used the unstarred sections of Chapters 1–6 and Chapter 9 in a successful one-semester junior-level course, *Introduction to Theory of Computation*, at New York University. A course on *finite automata and formal languages* could be based on Chapters 1, 9, and 10. A semester or quarter course on *logic for computer scientists* could be based on selections from Parts 1 and 3. Part 5 could be used for a third semester on the theory of computation or an introduction to *programming language semantics*. Many other arrangements and courses are possible, as should be apparent from the dependency graph, which follows the Acknowledgments. It is our hope, however, that this book will help readers to see theoretical computer science not as a fragmented list of discrete topics, but rather as a unified subject drawing on powerful mathematical methods and on intuitions derived from experience with computing technology to give valuable insights into a vital new area of human knowledge.

Note to the Reader

Many readers will wish to begin with Chapter 2, using the material of Chapter 1 for reference as required. Readers who enjoy skipping around will find the *dependency graph* useful.

Sections marked with an asterisk (*) may be skipped without loss of continuity. The relationship of these sections to later material is given in the dependency graph.

Exercises marked with an asterisk either introduce new material, refer to earlier material in ways not indicated in the dependency graph, or simply are considered more difficult than unmarked exercises.

A reference to Theorem 8.1 is to Theorem 8.1 of the chapter in which the reference is made. When a reference is to a theorem in another chapter, the chapter is specified. The same system is used in referring to numbered formulas and to exercises.

Acknowledgments

It is a pleasure to acknowledge the help we have received. Charlene Herring, Debbie Herring, Barry Jacobs, and Joseph Miller made their student classroom notes available to us. James Cox, Keith Harrow, Steve Henkind, Karen Lemone, Colm O'Dunlaing, and James Robinett provided helpful comments and corrections. Stewart Weiss was kind enough to redraw one of the figures. Thomas Ostrand, Norman Shulman, Louis Salkind, Ron Sigal, Patricia Teller, and Elia Weixelbaum were particularly generous with their time, and devoted many hours to helping us. We are especially grateful to them.

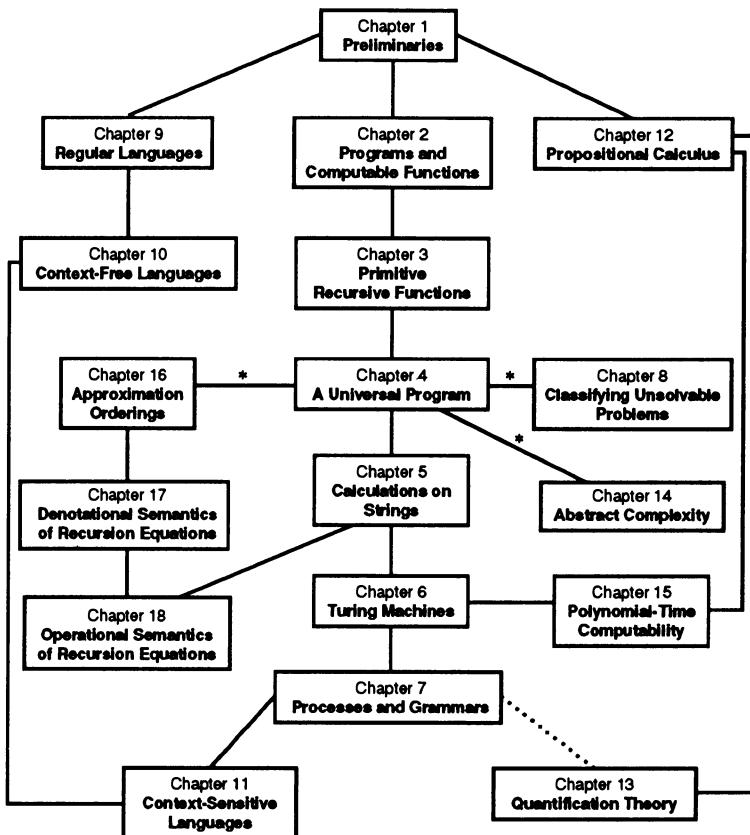
Acknowledgments to Corrected Printing

We have taken this opportunity to correct a number of errors. We are grateful to the readers who have called our attention to errors and who have suggested corrections. The following have been particularly helpful: Alissa Bernholc, Domenico Cantone, John R. Cowles, Herbert Enderton, Phyllis Frankl, Fred Green, Warren Hirsch, J. D. Monk, Steve Rozen, and Stewart Weiss.

Acknowledgments to Second Edition

Yuri Gurevich, Paliath Narendran, Robert Paige, Carl Smith, and particularly Robert McNaughton made numerous suggestions for improving the first edition. Kung Chen, William Hurwood, Dana Latch, Sidd Puri, Benjamin Russell, Jason Smith, Jean Toal, and Niping Wu read a preliminary version of Part 5.

Dependency Graph



A solid line between two chapters indicates the dependence of the unstarred sections of the higher numbered chapter on the unstarred sections of the lower numbered chapter. An asterisk next to a solid line indicates that knowledge of the starred sections of the lower numbered chapter is also assumed. A dotted line shows that knowledge of the unstarred sections of the lower numbered chapter is assumed for the starred sections of the higher numbered chapter.

1

Preliminaries

1. Sets and n -tuples

We shall often be dealing with *sets* of objects of some definite kind. Thinking of a collection of entities as a *set* simply amounts to a decision to regard the whole collection as a single object. We shall use the word *class* as synonymous with *set*. In particular we write N for the set of *natural numbers* $0, 1, 2, 3, \dots$. In this book the word *number* will always mean *natural number* except in contexts where the contrary is explicitly stated.

We write

$$a \in S$$

to mean that a belongs to S or, equivalently, is a member of the set S , and

$$a \notin S$$

to mean that a does not belong to S . It is useful to speak of the *empty set*, written \emptyset , which has no members. The equation $R = S$, where R and S are sets, means that R and S are *identical as sets*, that is, that they have exactly the same members. We write $R \subseteq S$ and speak of R as a *subset* of S to mean that every element of R is also an element of S . Thus, $R = S$ if and only if $R \subseteq S$ and $S \subseteq R$. Note also that for any set R , $\emptyset \subseteq R$ and $R \subseteq R$. We write $R \subset S$ to indicate that $R \subseteq S$ but $R \neq S$. In this case R

is called a *proper subset* of S . If R and S are sets, we write $R \cup S$ for the *union* of R and S , which is the collection of all objects which are members of either R or S or both. $R \cap S$, the *intersection* of R and S , is the set of all objects that belong to both R and S . $R - S$, the set of all objects that belong to R and do not belong to S , is the *difference* between R and S . S may contain objects not in R . Thus $R - S = R - (R \cap S)$. Often we will be working in contexts where all sets being considered are subsets of some fixed set D (sometimes called a *domain* or a *universe*). In such a case we write \bar{S} for $D - S$, and call \bar{S} the *complement* of S . Most frequently we shall be writing \bar{S} for $N - S$. The De Morgan identities

$$\overline{R \cup S} = \bar{R} \cap \bar{S},$$

$$\overline{R \cap S} = \bar{R} \cup \bar{S}$$

are very useful; they are easy to check and any reader not already familiar with them should do so. We write

$$\{a_1, a_2, \dots, a_n\}$$

for the set consisting of the n objects a_1, a_2, \dots, a_n . Sets that can be written in this form as well as the empty set are called *finite*. Sets that are not finite, e.g., N , are called *infinite*. It should be carefully noted that a and $\{a\}$ are not the same thing. In particular, $a \in S$ is true if and only if $\{a\} \subseteq S$. Since two sets are equal if and only if they have the same members, it follows that, for example, $\{a, b, c\} = \{a, c, b\} = \{b, a, c\}$. That is, the order in which we may choose to write the members of a set is irrelevant. Where order is important, we speak instead of an n -tuple or a *list*. We write n -tuples using parentheses rather than curly braces:

$$(a_1, \dots, a_n).$$

Naturally, the elements making up an n -tuple need not be distinct. Thus $(4, 1, 4, 2)$ is a 4-tuple. A 2-tuple is called an *ordered pair*, and a 3-tuple is called an *ordered triple*. Unlike the case for sets of one object, we *do not distinguish between the object a and the 1-tuple (a)* . The crucial property of n -tuples is

$$(a_1, a_2, \dots, a_n) = (b_1, b_2, \dots, b_n)$$

if and only if

$$a_1 = b_1, \quad a_2 = b_2, \quad \dots, \quad \text{and} \quad a_n = b_n.$$

If S_1, S_2, \dots, S_n are given sets, then we write $S_1 \times S_2 \times \dots \times S_n$ for the set of all n -tuples (a_1, a_2, \dots, a_n) such that $a_1 \in S_1, a_2 \in S_2, \dots, a_n \in S_n$.

$S_1 \times S_2 \times \cdots \times S_n$ is sometimes called the *Cartesian product* of S_1, S_2, \dots, S_n . In case $S_1 = S_2 = \cdots = S_n = S$ we write S^n for the Cartesian product $S_1 \times S_2 \times \cdots \times S_n$.

2. Functions

Functions play an important role in virtually every branch of pure and applied mathematics. We may define a function simply as a set f , all of whose members are ordered pairs and that has the special property

$$(a, b) \in f \text{ and } (a, c) \in f \quad \text{implies} \quad b = c.$$

However, intuitively it is more helpful to think of the pairs listed as the rows of a table. For f a function, one writes $f(a) = b$ to mean that $(a, b) \in f$; the definition of function ensures that for each a there can be at most one such b . The set of all a such that $(a, b) \in f$ for some b is called the *domain* of f . The set of all $f(a)$ for a in the domain of f is called the *range* of f .

As an example, let f be the set of ordered pairs (n, n^2) for $n \in N$. Then, for each $n \in N$, $f(n) = n^2$. The domain of f is N . The range of f is the set of perfect squares.

Functions f are often specified by *algorithms* that provide procedures for obtaining $f(a)$ from a . This method of specifying functions is particularly important in computer science. However, as we shall see in Chapter 4, it is quite possible to possess an algorithm that specifies a function without being able to tell which elements belong to its domain. This makes the notion of a so-called *partial function* play a central role in computability theory. A *partial function on a set S* is simply a function whose domain is a subset of S . An example of a partial function on N is given by $g(n) = \sqrt{n}$, where the domain of g is the set of perfect squares. If f is a partial function on S and $a \in S$, then we write $f(a) \downarrow$ and say that $f(a)$ is *defined* to indicate that a is in the domain of f ; if a is not in the domain of f , we write $f(a) \uparrow$ and say that $f(a)$ is *undefined*. If a partial function on S has the domain S , then it is called *total*. Finally, we should mention that the empty set \emptyset is itself a function. Considered as a partial function on some set S , it is *nowhere defined*.

For a partial function f on a Cartesian product $S_1 \times S_2 \times \cdots \times S_n$, we write $f(a_1, \dots, a_n)$ rather than $f((a_1, \dots, a_n))$. A partial function f on a set S^n is called an *n-ary partial function on S* , or a function of n variables on S . We use *unary* and *binary* for 1-ary and 2-ary, respectively. For n -ary partial functions, we often write $f(x_1, \dots, x_n)$ instead of f as a way of showing explicitly that f is n -ary.

Sometimes it is useful to work with particular kinds of functions. A function f is *one-one* if, for all x, y in the domain of f , $f(x) = f(y)$ implies $x = y$. Stated differently, if $x \neq y$ then $f(x) \neq f(y)$. If the range of f is the set S , then we say that f is an *onto* function with respect to S , or simply that f is *onto S*. For example, $f(n) = n^2$ is one-one, and f is onto the set of perfect squares, but it is not onto N .

We will sometimes refer to the idea of *closure*. If S is a set and f is a partial function on S , then S is *closed under f* if the range of f is a subset of S . For example, N is closed under $f(n) = n^2$, but it is not closed under $h(n) = \sqrt{n}$ (where h is a total function on N).

3. Alphabets and Strings

An *alphabet* is simply some finite nonempty set A of objects called *symbols*. An n -tuple of symbols of A is called a *word* or a *string* on A . Instead of writing a word as (a_1, a_2, \dots, a_n) we write simply $a_1 a_2 \cdots a_n$. If $u = a_1 a_2 \cdots a_n$, then we say that n is the length of u and write $|u| = n$. We allow a unique null word, written 0, of length 0. (The reason for using the same symbol for the number zero and the null word will become clear in Chapter 5.) The set of all words on the alphabet A is written A^* . Any subset of A^* is called a *language on A* or a *language with alphabet A*. We do *not* distinguish between a symbol $a \in A$ and the word of length 1 consisting of that symbol. If $u, v \in A^*$, then we write \widehat{uv} for the word obtained by placing the string v after the string u . For example, if $A = \{a, b, c\}$, $u = bab$, and $v = caa$, then

$$\widehat{uv} = babcaa \quad \text{and} \quad \widehat{vu} = caabab.$$

Where no confusion can result, we write uv instead of \widehat{uv} . It is obvious that, for all u ,

$$u0 = 0u = u,$$

and that, for all u, v, w ,

$$u(vw) = (uv)w.$$

Also, if either $uv = uw$ or $vu = wu$, then $v = w$.

If u is a string, and $n \in N$, $n > 0$, we write

$$u^{[n]} = \underbrace{uu \cdots u}_n.$$

We also write $u^{[0]} = 0$. We use the square brackets to avoid confusion with numerical exponentiation.

If $u \in A^*$, we write u^R for u written backward; i.e., if $u = a_1 a_2 \cdots a_n$, for $a_1, \dots, a_n \in A$, then $u^R = a_n \cdots a_2 a_1$. Clearly, $0^R = 0$ and $(uv)^R = v^R u^R$ for $u, v \in A^*$.

4. Predicates

By a *predicate* or a *Boolean-valued function* on a set S we mean a *total* function P on S such that for each $a \in S$, either

$$P(a) = \text{TRUE} \quad \text{or} \quad P(a) = \text{FALSE},$$

where TRUE and FALSE are a pair of distinct objects called *truth values*. We often say $P(a)$ is *true* for $P(a) = \text{TRUE}$, and $P(a)$ is *false* for $P(a) = \text{FALSE}$. For our purposes it is useful to identify the truth values with specific numbers, so we set

$$\text{TRUE} = 1 \quad \text{and} \quad \text{FALSE} = 0.$$

Thus, a predicate is a special kind of function with values in N . Predicates on a set S are usually specified by expressions which become statements, either true or false, when variables in the expression are replaced by symbols designating fixed elements of S . Thus the expression

$$x < 5$$

specifies a predicate on N , namely,

$$P(x) = \begin{cases} 1 & \text{if } x = 0, 1, 2, 3, 4 \\ 0 & \text{otherwise.} \end{cases}$$

Three basic operations on truth values are defined by the tables in Table 4.1. Thus if P and Q are predicates on a set S , there are also the predicates $\sim P$, $P \& Q$, $P \vee Q$. $\sim P$ is true just when P is false; $P \& Q$ is true when both P and Q are true, otherwise it is false; $P \vee Q$ is true when either P or Q or both are true, otherwise it is false. Given a predicate P

Table 4.1

p	$\sim p$	p	q	$p \& q$	$p \vee q$
0	1	1	1	1	1
1	0	0	1	0	1

on a set S , there is a corresponding subset R of S , namely, the set of all elements $a \in S$ for which $P(a) = 1$. We write

$$R = \{a \in S | P(a)\}.$$

Conversely, given a subset R of a given set S , the expression

$$x \in R$$

defines a predicate on S , namely, the predicate defined by

$$P(x) = \begin{cases} 1 & \text{if } x \in R \\ 0 & \text{if } x \notin R. \end{cases}$$

Of course, in this case,

$$R = \{x \in S | P(x)\}.$$

The predicate P is called the *characteristic function* of the set R . The close connection between sets and predicates is such that one can readily translate back and forth between discourse involving one of these notions and discourse involving the other. Thus we have

$$\begin{aligned} \{x \in S | P(x) \& Q(x)\} &= \{x \in S | P(x)\} \cap \{x \in S | Q(x)\}, \\ \{x \in S | P(x) \vee Q(x)\} &= \{x \in S | P(x)\} \cup \{x \in S | Q(x)\}, \\ \{x \in S | \sim P(x)\} &= S - \{x \in S | P(x)\}. \end{aligned}$$

To indicate that two expressions containing variables define the same predicate we place the symbol \Leftrightarrow between them. Thus,

$$x < 5 \Leftrightarrow x = 0 \vee x = 1 \vee x = 2 \vee x = 3 \vee x = 4.$$

The De Morgan identities from Section 1 can be expressed as follows in terms of predicates on a set S :

$$P(x) \& Q(x) \Leftrightarrow \sim(\sim P(x) \vee \sim Q(x)),$$

$$P(x) \vee Q(x) \Leftrightarrow \sim(\sim P(x) \& \sim Q(x)).$$

5. Quantifiers

In this section we will be concerned exclusively with predicates on N^m (or what is the same thing, m -ary predicates on N) for different values of m . Here and later we omit the phrase “on N ” when the meaning is clear.

Thus, let $P(t, x_1, \dots, x_n)$ be an $(n + 1)$ -ary predicate. Consider the predicate $Q(y, x_1, \dots, x_n)$ defined by

$$\begin{aligned} Q(y, x_1, \dots, x_n) &\Leftrightarrow P(0, x_1, \dots, x_n) \vee P(1, x_1, \dots, x_n) \\ &\quad \vee \cdots \vee P(y, x_1, \dots, x_n). \end{aligned}$$

Thus the predicate $Q(y, x_1, \dots, x_n)$ is true just in case there is a value of $t \leq y$ such that $P(t, x_1, \dots, x_n)$ is true. We write this predicate Q as

$$(\exists t)_{\leq y} P(t, x_1, \dots, x_n).$$

The expression “ $(\exists t)_{\leq y}$ ” is called a *bounded existential quantifier*. Similarly, we write $(\forall t)_{\leq y} P(t, x_1, \dots, x_n)$ for the predicate

$$P(0, x_1, \dots, x_n) \& P(1, x_1, \dots, x_n) \& \cdots \& P(y, x_1, \dots, x_n).$$

This predicate is true just in case $P(t, x_1, \dots, x_n)$ is true for *all* $t \leq y$. The expression “ $(\forall t)_{\leq y}$ ” is called a *bounded universal quantifier*. We also write $(\exists t)_{< y} P(t, x_1, \dots, x_n)$ for the predicate that is true just in case $P(t, x_1, \dots, x_n)$ is true for at least one value of $t < y$ and $(\forall t)_{< y} P(t, x_1, \dots, x_n)$ for the predicate that is true just in case $P(t, x_1, \dots, x_n)$ is true for all values of $t < y$.

We write

$$Q(x_1, \dots, x_n) \Leftrightarrow (\exists t) P(t, x_1, \dots, x_n)$$

for the predicate which is true if there exists some $t \in N$ for which $P(t, x_1, \dots, x_n)$ is true. Similarly, $(\forall t) P(t, x_1, \dots, x_n)$ is true if $P(t, x_1, \dots, x_n)$ is true for all $t \in N$.

The following generalized De Morgan identities are sometimes useful:

$$\sim (\exists t)_{\leq y} P(t, x_1, \dots, x_n) \Leftrightarrow (\forall t)_{\leq y} \sim P(t, x_1, \dots, x_n),$$

$$\sim (\exists t) P(t, x_1, \dots, x_n) \Leftrightarrow (\forall t) \sim P(t, x_1, \dots, x_n).$$

The reader may easily verify the following examples:

$$(\exists y)(x + y = 4) \Leftrightarrow x \leq 4,$$

$$(\exists y)(x + y = 4) \Leftrightarrow (\exists y)_{\leq 4}(x + y = 4),$$

$$(\forall y)(xy = 0) \Leftrightarrow x = 0,$$

$$(\exists y)_{\leq z}(x + y = 4) \Leftrightarrow (x + z \geq 4 \& x \leq 4).$$

6. Proof by Contradiction

In this book we will be calling many of the assertions we make *theorems* (or *corollaries* or *lemmas*) and providing *proofs* that they are correct. Why are proofs necessary? The following example should help in answering this question.

Recall that a number is called a *prime* if it has *exactly two distinct divisors*, itself and 1. Thus 2, 17, and 41 are primes, but 0, 1, 4, and 15 are not. Consider the following assertion:

$$n^2 - n + 41 \text{ is prime for all } n \in N.$$

This assertion is in fact *false*. Namely, for $n = 41$ the expression becomes

$$41^2 - 41 + 41 = 41^2,$$

which is certainly not a prime. However, the assertion is true (readers with access to a computer can easily check this!) for all $n \leq 40$. This example shows that inferring a result about all members of an infinite set (such as N) from even a large finite number of instances can be very dangerous. A proof is intended to overcome this obstacle.

A proof begins with some initial statements and uses logical reasoning to infer additional statements. (In Chapters 12 and 13 we shall see how the notion of logical reasoning can be made precise; but in fact, our *use* of logical reasoning will be in an informal intuitive style.) When the initial statements with which a proof begins are already accepted as correct, then any of the additional statements inferred can also be accepted as correct. But proofs often cannot be carried out in this simple-minded pattern. In this and the next section we will discuss more complex proof patterns.

In a *proof by contradiction*, one begins by supposing that the assertion we wish to prove is false. Then we can feel free to use the negation of what we are trying to prove as one of the initial statements in constructing a proof. In a proof by contradiction we look for a pair of statements developed in the course of the proof which *contradict* one another. Since both cannot be true, we have to conclude that our original supposition was wrong and therefore that our desired conclusion is correct.

We give two examples here of proof by contradiction. There will be many in the course of the book. Our first example is quite famous. We recall that every number is either even (i.e., $= 2n$ for some $n \in N$) or odd (i.e., $= 2n + 1$ for some $n \in N$). Moreover, if m is even, $m = 2n$, then $m^2 = 4n^2 = 2 \cdot 2n^2$ is even, while if m is odd, $m = 2n + 1$, then $m^2 = 4n^2 + 4n + 1 = 2(2n^2 + 2n) + 1$ is odd. We wish to prove that the equation

$$2 = (m/n)^2 \tag{6.1}$$

has no solution for $m, n \in N$ (that is, that $\sqrt{2}$ is not a “rational” number). We suppose that our equation has a solution and proceed to derive a contradiction. Given our supposition that (6.1) has a solution, it must have a solution in which m and n are not both even numbers. This is true because if m and n are both even, we can repeatedly “cancel” 2 from numerator and denominator until at least one of them is odd. On the other hand, we shall prove that for every solution of (6.1) m and n must both be even. The contradiction will show that our supposition was false, i.e., that (6.1) has no solution.

It remains to show that in every solution of (6.1), m and n are both even. We can rewrite (6.1) as

$$m^2 = 2n^2,$$

which shows that m^2 is even. As we saw above this implies that m is even, say $m = 2k$. Thus, $m^2 = 4k^2 = 2n^2$, or $n^2 = 2k^2$. Thus, n^2 is even and hence n is even. ■

Note the symbol ■, which means “the proof is now complete.”

Our second example involves strings as discussed in Section 3.

Theorem 6.1. Let $x \in \{a, b\}^*$ such that $xa = ax$. Then $x = a^{[n]}$ for some $n \in N$.

Proof. Suppose that $xa = ax$ but x contains the letter b . Then we can write $x = a^{[n]}bu$, where we have explicitly shown the first (i.e., leftmost) occurrence of b in x . Then

$$a^{[n]}bua = aa^{[n]}bu = a^{[n+1]}bu.$$

Thus,

$$bua = abu.$$

But this is impossible, since the same string cannot have its first symbol be both b and a . This contradiction proves the theorem. ■

Exercises

1. Prove that the equation $(p/q)^2 = 3$ has no solution for $p, q \in N$.
2. Prove that if $x \in \{a, b\}^*$ and $abx = xab$, then $x = (ab)^{[n]}$ for some $n \in N$.

7. Mathematical Induction

Mathematical induction furnishes an important technique for proving statements of the form $(\forall n)P(n)$, where P is a predicate on N . One

proceeds by proving a pair of auxiliary statements, namely,

$$P(0)$$

and

$$(\forall n)(\text{If } P(n) \text{ then } P(n + 1)). \quad (7.1)$$

Once we have succeeded in proving these auxiliary statements we can regard $(\forall n)P(n)$ as also proved. The justification for this is as follows.

From the second auxiliary statement we can infer each of the infinite set of statements:

$$\begin{aligned} &\text{If } P(0) \text{ then } P(1), \\ &\text{If } P(1) \text{ then } P(2), \\ &\text{If } P(2) \text{ then } P(3), \dots . \end{aligned}$$

Since we have proved $P(0)$, we can infer $P(1)$. Having now proven $P(1)$ we can get $P(2)$, etc. Thus, we see that $P(n)$ is true for all n and hence $(\forall n)P(n)$ is true.

Why is this helpful? Because sometimes it is much easier to prove (7.1) than to prove $(\forall n)P(n)$ in some other way. In proving this second auxiliary proposition one typically considers some fixed but arbitrary value k of n and shows that if we assume $P(k)$ we can prove $P(k + 1)$. $P(k)$ is then called the *induction hypothesis*. This methodology enables us to use $P(k)$ as one of the initial statements in the proof we are constructing.

There are some paradoxical things about proofs by mathematical induction. One is that considered superficially, it seems like an example of circular reasoning. One seems to be assuming $P(k)$ for an arbitrary k , which is exactly what one is supposed to be engaged in proving. Of course, one is not really assuming $(\forall n)P(n)$. One is assuming $P(k)$ for some *particular* k in order to show that $P(k + 1)$ follows.

It is also paradoxical that in using induction (we shall often omit the word *mathematical*), it is sometimes easier to prove statements by first making them “stronger.” We can put this schematically as follows. We wish to prove $(\forall n)P(n)$. Instead we decide to prove the *stronger* assertion $(\forall n)(P(n) \& Q(n))$ (which of course implies the original statement). Proving the stronger statement by induction requires that we prove

$$P(0) \& Q(0)$$

and

$$(\forall n)[\text{If } P(n) \& Q(n) \text{ then } P(n + 1) \& Q(n + 1)].$$

In proving this second auxiliary statement, we may take $P(k) \& Q(k)$ as our induction hypothesis. Thus, although strengthening the statement to

be proved gives us more to prove, it also gives us a stronger induction hypothesis and, therefore, more to work with. The technique of deliberately strengthening what is to be proven for the purpose of making proofs by induction easier is called *induction loading*.

It is time for an example of a proof by induction. The following is useful in doing one of the exercises in Chapter 6.

Theorem 7.1. For all $n \in N$ we have $\sum_{i=0}^n (2i + 1) = (n + 1)^2$.

Proof. For $n = 0$, our theorem states simply that $1 = 1^2$, which is true.

Suppose the result known for $n = k$. That is, our induction hypothesis is

$$\sum_{i=0}^k (2i + 1) = (k + 1)^2.$$

Then

$$\begin{aligned}\sum_{i=0}^{k+1} (2i + 1) &= \sum_{i=0}^k (2i + 1) + 2(k + 1) + 1 \\ &= (k + 1)^2 + 2(k + 1) + 1 \\ &= (k + 2)^2.\end{aligned}$$

But this is the desired result for $n = k + 1$. ■

Another form of mathematical induction that is often very useful is called *course-of-values induction* or sometimes *complete induction*. In the case of course-of-values induction we prove the single auxiliary statement

$$(\forall n)[\text{If } (\forall m)_{m < n} P(m) \text{ then } P(n)], \quad (7.2)$$

and then conclude that $(\forall n)P(n)$ is true. A potentially confusing aspect of course-of-values induction is the apparent lack of an initial statement $P(0)$. But in fact there is no such lack. The case $n = 0$ of (7.2) is

$$\text{If } (\forall m)_{m < 0} P(m) \text{ then } P(0).$$

But the “induction hypothesis” $(\forall m)_{m < 0} P(m)$ is entirely vacuous because there is no $m \in N$ such that $m < 0$. So in proving (7.2) for $n = 0$ we really are just proving $P(0)$. In practice it is sometimes possible to give a single proof of (7.2) that works for all n including $n = 0$. But often the case $n = 0$ has to be handled separately.

To see why course-of-values induction works, consider that, in the light of what we have said about the $n = 0$ case, (7.2) leads to the following

infinite set of statements:

$$\begin{aligned} & P(0), \\ & \text{If } P(0) \text{ then } P(1), \\ & \text{If } P(0) \& P(1) \text{ then } P(2), \\ & \text{If } P(0) \& P(1) \& P(2) \text{ then } P(3), \\ & \vdots \end{aligned}$$

Here is an example of a theorem proved by course-of-values induction.

Theorem 7.2. There is no string $x \in \{a, b\}^*$ such that $ax = xb$.

Proof. Consider the following predicate: *If* $x \in \{a, b\}^*$ and $|x| = n$, *then* $ax \neq xb$. We will show that this is true for all $n \in N$. So we assume it true for all $m < k$ for some given k and show that it follows for k . This proof will be by contradiction. Thus, suppose that $|x| = k$ and $ax = xb$. The equation implies that a is the first and b the last symbol in x . So, we can write $x = aub$. Then

$$aaub = aubb,$$

i.e.,

$$au = ub.$$

But $|u| < |x|$. Hence by the induction hypothesis $au \neq ub$. This contradiction proves the theorem. ■

Proofs by course-of-values induction can always be rewritten so as to involve reference to the principle that if some predicate is true for some element of N , then there must be a least element of N for which it is true. Here is the proof of Theorem 7.2 given in this style.

Proof. Suppose there is a string $x \in \{a, b\}^*$ such that $ax = xb$. Then there must be a string satisfying this equation of minimum length. Let x be such a string. Then $ax = xb$, but, if $|u| < |x|$, then $au \neq ub$. However, $ax = xb$ implies that $x = aub$, so that $au = ub$ and $|u| < |x|$. This contradiction proves the theorem. ■

Exercises

1. Prove by mathematical induction that $\sum_{i=1}^n i = n(n + 1)/2$.
2. Here is a “proof” by mathematical induction that if $x, y \in N$, then $x = y$. What is wrong?

Let

$$\max(x, y) = \begin{cases} x & \text{if } x \geq y \\ y & \text{otherwise} \end{cases}$$

for $x, y \in N$. Consider the predicate

$$(\forall x)(\forall y)[\text{If } \max(x, y) = n, \text{ then } x = y].$$

For $n = 0$, this is clearly true. Assume the result for $n = k$, and let $\max(x, y) = k + 1$. Let $x_1 = x - 1, y_1 = y - 1$. Then $\max(x_1, y_1) = k$. By the induction hypothesis, $x_1 = y_1$ and therefore $x = x_1 + 1 = y_1 + 1 = y$.

3. Here is another incorrect proof that purports to use mathematical induction to prove that all flowers have the same color! What is wrong?

Consider the following predicate: If S is a set of flowers containing exactly n elements, then all the flowers in S have the same color. The predicate is clearly true if $n = 1$. We suppose it true for $n = k$ and prove the result for $n = k + 1$. Thus, let S be a set of $k + 1$ flowers. If we remove one flower from S we get a set of k flowers. Therefore, by the induction hypothesis they all have the same color. Now return the flower removed from S and remove another. Again by our induction hypothesis the remaining flowers all have the same color. But now both of the flowers removed have been shown to have the same color as the rest. Thus, all the flowers in S have the same color.

4. Show that there are no strings $x, y \in \{a, b\}^*$ such that $xay = ybx$.
5. Give a “one-line” proof of Theorem 7.2 that does not use mathematical induction.

Part 1

Computability

2

Programs and Computable Functions

1. A Programming Language

Our development of computability theory will be based on a specific programming language \mathcal{S} . We will use certain letters as variables whose values are *numbers*. (In this book the word *number* will always mean nonnegative integer, unless the contrary is specifically stated.) In particular, the letters

$$X_1 \ X_2 \ X_3 \ \dots$$

will be called the *input variables* of \mathcal{S} , the letter Y will be called the *output variable* of \mathcal{S} , and the letters

$$Z_1 \ Z_2 \ Z_3 \ \dots$$

will be called the *local variables* of \mathcal{S} . The subscript 1 is often omitted; i.e., X stands for X_1 and Z for Z_1 . Unlike the programming languages in actual use, there is no upper limit on the values these variables can assume. Thus from the outset, \mathcal{S} must be regarded as a purely theoretical entity. Nevertheless, readers having programming experience will find working with \mathcal{S} very easy.

In \mathcal{S} we will be able to write “instructions” of various sorts; a “program” of \mathcal{S} will then consist of a *list* (i.e., a finite sequence) of

Table 1.1

Instruction	Interpretation
$V \leftarrow V + 1$	Increase by 1 the value of the variable V .
$V \leftarrow V - 1$	If the value of V is 0, leave it unchanged; otherwise decrease by 1 the value of V .
IF $V \neq 0$ GOTO L	If the value of V is nonzero, perform the instruction with label L next; otherwise proceed to the next instruction in the list.

instructions. For example, for each variable V there will be an instruction:

$$V \leftarrow V + 1$$

A simple example of a program of \mathcal{S} is

$$\begin{aligned} X &\leftarrow X + 1 \\ X &\leftarrow X + 1 \end{aligned}$$

“Execution” of this program has the effect of increasing the value of X by 2. In addition to variables, we will need “labels.” In \mathcal{S} these are

$$A_1 \ B_1 \ C_1 \ D_1 \ E_1 \ A_2 \ B_2 \ C_2 \ D_2 \ E_2 \ A_3 \ \dots$$

Once again the subscript 1 can be omitted. We give in Table 1.1 a complete list of our instructions. In this list V stands for any variable and L stands for any label.

These instructions will be called the *increment*, *decrement*, and *conditional branch* instructions, respectively.

We will use the special convention that *the output variable Y and the local variables Z_i initially have the value 0*. We will sometimes indicate the value of a variable by writing it in lowercase italics. Thus x_5 is the value of X_5 .

Instructions may or may not have labels. When an instruction is labeled, the label is written to its left in square brackets. For example,

$$[B] \quad Z \leftarrow Z - 1$$

In order to base computability theory on the language \mathcal{S} , we will require formal definitions. But before we supply these, it is instructive to work informally with programs of \mathcal{S} .

2. Some Examples of Programs

(a) Our first example is the program

$$\begin{aligned} [A] \quad &X \leftarrow X - 1 \\ &Y \leftarrow Y + 1 \\ &\text{IF } X \neq 0 \text{ GOTO } A \end{aligned}$$

If the initial value x of X is not 0, the effect of this program is to copy x into Y and to decrement the value of X down to 0. (By our conventions the initial value of Y is 0.) If $x = 0$, then the program halts with Y having the value 1. We will say that this program *computes* the function

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ x & \text{otherwise.} \end{cases}$$

This program halts when it executes the third instruction of the program with X having the value 0. In this case the condition $X \neq 0$ is not fulfilled and therefore the branch is not taken. When an attempt is made to move on to the nonexistent fourth instruction, the program halts. A program will also halt if an instruction labeled L is to be executed, but there is no instruction in the program with that label. In this case, we usually will use the letter E (for “exit”) as the label which labels no instruction.

(b) Although the preceding program is a perfectly well-defined program of our language \mathcal{S} , we may think of it as having arisen in an attempt to write a program that copies the value of X into Y , and therefore containing a “bug” because it does not handle 0 correctly. The following slightly more complicated example remedies this situation.

[A]	IF $X \neq 0$ GOTO B $Z \leftarrow Z + 1$ IF $Z \neq 0$ GOTO E
[B]	$X \leftarrow X - 1$ $Y \leftarrow Y + 1$ $Z \leftarrow Z + 1$ IF $Z \neq 0$ GOTO A

As we can easily convince ourselves, this program does copy the value of X into Y for all initial values of X . Thus, we say that it computes the function $f(x) = x$. At first glance Z ’s role in the computation may not be obvious. It is used simply to allow us to code an *unconditional branch*. That is, the program segment

$$\begin{aligned} Z &\leftarrow Z + 1 \\ \text{IF } Z &\neq 0 \text{ GOTO } L \end{aligned} \tag{2.1}$$

has the effect (ignoring the effect on the value of Z) of an instruction

GOTO L

such as is available in most programming languages. To see that this is true we note that the first instruction of the segment guarantees that Z has a nonzero value. Thus the condition $Z \neq 0$ is always true and hence the next instruction performed will be the instruction labeled L . Now *GOTO L* is

not an instruction in our language \mathcal{S} , but since we will frequently have use for such an instruction, we can use it as an abbreviation for the program segment (2.1). Such an abbreviating pseudoinstruction will be called a *macro* and the program or program segment which it abbreviates will be called its *macro expansion*.

The use of these terms is obviously motivated by similarities with the notion of a macro instruction occurring in many programming languages. At this point we will not discuss how to ensure that the variables local to the macro definition are distinct from the variables used in the main program. Instead, we will manually replace any such duplicate variable uses with unused variables. This will be illustrated in the “expanded” multiplication program in (e). In Section 5 this matter will be dealt with in a formal manner.

(c) Note that although the program of (b) does copy the value of X into Y , in the process the value of X is “destroyed” and the program terminates with X having the value 0. Of course, typically, programmers want to be able to copy the value of one variable into another without the original being “zeroed out.” This is accomplished in the next program. (Note that we use our macro instruction GOTO L several times to shorten the program. Of course, if challenged, we could produce a legal program of \mathcal{S} by replacing each GOTO L by a macro expansion. These macro expansions would have to use a local variable other than Z so as not to interfere with the value of Z in the main program.)

[A]	If $X \neq 0$ GOTO B
	GOTO C
[B]	$X \leftarrow X - 1$
	$Y \leftarrow Y + 1$
	$Z \leftarrow Z + 1$
	GOTO A
[C]	IF $Z \neq 0$ GOTO D
	GOTO E
[D]	$Z \leftarrow Z - 1$
	$X \leftarrow X + 1$
	GOTO C

In the first loop, this program copies the value of X into both Y and Z , while in the second loop, the value of X is restored. When the program terminates, both X and Y contain X ’s original value and $z = 0$.

We wish to use this program to justify the introduction of a macro which we will write

$$V \leftarrow V'$$

the execution of which will replace the contents of the variable V by the contents of the variable V' while leaving the contents of V' unaltered. Now, this program (c) functions correctly as a copying program only under our assumption that the variables Y and Z are initialized to the value 0. Thus, we can use the program as the basis of a macro expansion of $V \leftarrow V'$ only if we can arrange matters so as to be sure that the corresponding variables have the value 0 whenever the macro expansion is entered. To solve this problem we introduce the macro

$$V \leftarrow 0$$

which will have the effect of setting the contents of V equal to 0. The corresponding macro expansion is simply

$$\begin{aligned} [L] \quad & V \leftarrow V - 1 \\ & \text{IF } V \neq 0 \text{ GOTO } L \end{aligned}$$

where, of course, the label L is to be chosen to be different from any of the labels in the main program. We can now write the macro expansion of $V \leftarrow V'$ by letting the macro $V \leftarrow 0$ precede the program which results when X is replaced by V' and Y is replaced by V in program (c). The result is as follows:

$$\begin{aligned} & V \leftarrow 0 \\ [A] \quad & \text{IF } V' \neq 0 \text{ GOTO } B \\ & \text{GOTO } C \\ [B] \quad & V' \leftarrow V' - 1 \\ & V \leftarrow V + 1 \\ & Z \leftarrow Z + 1 \\ & \text{GOTO } A \\ [C] \quad & \text{IF } Z \neq 0 \text{ GOTO } D \\ & \text{GOTO } E \\ [D] \quad & Z \leftarrow Z - 1 \\ & V' \leftarrow V' + 1 \\ & \text{GOTO } C \end{aligned}$$

With respect to this macro expansion the following should be noted:

1. It is unnecessary (although of course it would be harmless) to include a $Z \leftarrow 0$ macro at the beginning of the expansion because, as has already been remarked, program (c) terminates with $z = 0$.
2. When inserting the expansion in an actual program, the variable Z will have to be replaced by a local variable which does not occur in the main program.

3. Likewise the labels A, B, C, D will have to be replaced by labels which do not occur in the main program.
4. Finally, the label E in the macro expansion must be replaced by a label L such that the instruction which follows the macro in the main program (if there is one) begins $[L]$.

- (d) A program with two inputs that computes the function

$$f(x_1, x_2) = x_1 + x_2$$

is as follows:

	$Y \leftarrow X_1$
	$Z \leftarrow X_2$
$[B]$	IF $Z \neq 0$ GOTO A
	GOTO E
$[A]$	$Z \leftarrow Z - 1$
	$Y \leftarrow Y + 1$
	GOTO B

Again, if challenged we would supply macro expansions for “ $Y \leftarrow X_1$ ” and “ $Z \leftarrow X_2$ ” as well as for the two unconditional branches. Note that Z is used to preserve the value of X_2 .

- (e) We now present a program that multiplies, i.e. that computes $f(x_1, x_2) = x_1 \cdot x_2$. Since multiplication can be regarded as repeated addition, we are led to the “program”

	$Z_2 \leftarrow X_2$
$[B]$	IF $Z_2 \neq 0$ GOTO A
	GOTO E
$[A]$	$Z_2 \leftarrow Z_2 - 1$
	$Z_1 \leftarrow X_1 + Y$
	$Y \leftarrow Z_1$
	GOTO B

Of course, the “instruction” $Z_1 \leftarrow X_1 + Y$ is not permitted in the language \mathcal{S} . What we have in mind is that since we already have an addition program, we can replace the macro $Z_1 \leftarrow X_1 + Y$ by a program for computing it, which we will call its macro expansion. At first glance, one might wonder why the pair of instructions

$$Z_1 \leftarrow X_1 + Y$$

$$Y \leftarrow Z_1$$

was used in this program rather than the single instruction

$$Y \leftarrow X_1 + Y$$

since we simply want to replace the current value of Y by the sum of its value and X_1 . The sum program in (d) computes $Y = X_1 + X_2$. If we were to use that as a template, we would have to replace X_2 in the program by Y . Now if we tried to use Y also as the variable being assigned, the macro expansion would be as follows:

	$Y \leftarrow X_1$
	$Z \leftarrow Y$
[B]	IF $Z \neq 0$ GOTO A
	GOTO E
[A]	Z $\leftarrow Z - 1$
	$Y \leftarrow Y + 1$
	GOTO B

What does this program actually compute? It should not be difficult to see that instead of computing $x_1 + y$ as desired, this program computes $2x_1$. Since X_1 is to be added over and over again, it is important that X_1 not be destroyed by the addition program. Here is the multiplication program, showing the macro expansion of $Z_1 \leftarrow X_1 + Y$:

[B]	$Z_2 \leftarrow X_2$
	IF $Z_2 \neq 0$ GOTO A
	GOTO E
[A]	$Z_2 \leftarrow Z_2 - 1$
	$Z_1 \leftarrow X_1$
	$Z_3 \leftarrow Y$
[B ₂]	IF $Z_3 \neq 0$ GOTO A ₂
	GOTO E ₂
[A ₂]	$Z_3 \leftarrow Z_3 - 1$
	$Z_1 \leftarrow Z_1 + 1$
	GOTO B ₂
[E ₂]	$Y \leftarrow Z_1$
	GOTO B

Macro Expansion of
 $Z_1 \leftarrow X_1 + Y$

Note the following:

1. The local variable Z_1 in the addition program in (d) must be replaced by another local variable (we have used Z_3) because Z_1 (the other name for Z) is also used as a local variable in the multiplication program.

2. The labels A, B, E are used in the multiplication program and hence cannot be used in the macro expansion. We have used A_2, B_2, E_2 instead.
3. The instruction GOTO E_2 terminates the addition. Hence, it is necessary that the instruction immediately following the macro expansion be labeled E_2 .

In the future we will often omit such details in connection with macro expansions. All that is important is that our infinite supply of variables and labels guarantees that the needed changes can always be made.

- (f) For our final example, we take the program

	$Y \leftarrow X_1$
	$Z \leftarrow X_2$
[C]	IF $Z \neq 0$ GOTO A
	GOTO E
[A]	IF $Y \neq 0$ GOTO B
	GOTO A
[B]	$Y \leftarrow Y - 1$
	$Z \leftarrow Z - 1$
	GOTO C

If we begin with $X_1 = 5, X_2 = 2$, the program first sets $Y = 5$ and $Z = 2$. Successively the program sets $Y = 4, Z = 1$ and $Y = 3, Z = 0$. Thus, the computation terminates with $Y = 3 = 5 - 2$. Clearly, if we begin with $X_1 = m, X_2 = n$, where $m \geq n$, the program will terminate with $Y = m - n$.

What happens if we begin with a value of X_1 less than the value of X_2 , e.g., $X_1 = 2, X_2 = 5$? The program sets $Y = 2$ and $Z = 5$ and successively sets $Y = 1, Z = 4$ and $Y = 0, Z = 3$. At this point the computation enters the “loop”:

	[A] IF $Y \neq 0$ GOTO B
	GOTO A

Since $y = 0$, there is no way out of this loop and the computation will continue “forever.” Thus, if we begin with $X_1 = m, X_2 = n$, where $m < n$, the computation will never terminate. In this case (and in similar cases) we will say that the program computes the *partial function*

$$g(x_1, x_2) = \begin{cases} x_1 - x_2 & \text{if } x_1 \geq x_2 \\ \uparrow & \text{if } x_1 < x_2. \end{cases}$$

(Partial functions are discussed in Chapter 1, Section 2.)

Exercises

1. Write a program in \mathcal{S} (using macros freely) that computes the function $f(x) = 3x$.
2. Write a program in \mathcal{S} that solves Exercise 1 using no macros.
3. Let $f(x) = 1$ if x is even; $f(x) = 0$ if x is odd. Write a program in \mathcal{S} that computes f .
4. Let $f(x) = 1$ if x is even; $f(x)$ undefined if x is odd. Write a program in \mathcal{S} that computes f .
5. Let $f(x_1, x_2) = 1$ if $x_1 = x_2$; $f(x_1, x_2) = 0$ if $x_1 \neq x_2$. Without using macros, write a program in \mathcal{S} that computes f .
6. Let $f(x)$ be the greatest number n such that $n^2 \leq x$. Write a program in \mathcal{S} that computes f .
7. Let $\text{gcd}(x_1, x_2)$ be the greatest common divisor of x_1 and x_2 . Write a program in \mathcal{S} that computes gcd .

3. Syntax

We are now ready to be mercilessly precise about the language \mathcal{S} . Some of the description recapitulates the preceding discussion.

The symbols

$$X_1 \ X_2 \ X_3 \ \dots$$

are called *input variables*,

$$Z_1 \ Z_2 \ Z_3 \ \dots$$

are called *local variables*, and Y is called the *output variable* of \mathcal{S} . The symbols

$$A_1 \ B_1 \ C_1 \ D_1 \ E_1 \ A_2 \ B_2 \ \dots$$

are called *labels* of \mathcal{S} . (As already indicated, in practice the subscript 1 is often omitted.) A *statement* is one of the following:

```
V ← V + 1
V ← V - 1
V ← V
IF V ≠ 0 GOTO L
```

where V may be any variable and L may be any label.

Note that we have included among the statements of \mathcal{S} the “dummy” commands $V \leftarrow V$. Since execution of these commands leaves all values unchanged, they have no effect on what a program computes. They are included for reasons that will not be made clear until much later. But their inclusion is certainly quite harmless.

Next, an *instruction* is either a statement (in which case it is also called an *unlabeled instruction*) or $[L]$ followed by a statement (in which case the instruction is said to have L as its label or to be labeled L). A *program* is a list (i.e., a finite sequence) of instructions. The length of this list is called the *length* of the program. It is useful to include the *empty program* of length 0, which of course contains no instructions.

As we have seen informally, in the course of a computation, the variables of a program assume different numerical values. This suggests the following definition:

A *state of a program* \mathcal{P} is a list of equations of the form $V = m$, where V is a variable and m is a number, including an equation for each variable that occurs in \mathcal{P} and including no two equations with the same variable. As an example, let \mathcal{P} be the program of (b) from Section 2, which contains the variables $X Y Z$. The list

$$X = 4, \quad Y = 3, \quad Z = 3$$

is thus a state of \mathcal{P} . (The definition of *state* does not require that the state can actually be “attained” from some initial state.) The list

$$X_1 = 4, \quad X_2 = 5, \quad Y = 4, \quad Z = 4$$

is also a state of \mathcal{P} . (Recall that X is another name for X_1 and note that the definition permits inclusion of equations involving variables not actually occurring in \mathcal{P} .) The list

$$X = 3, \quad Z = 3$$

is *not* a state of \mathcal{P} since no equation in Y occurs. Likewise, the list

$$X = 3, \quad X = 4, \quad Y = 2, \quad Z = 2$$

is *not* a state of \mathcal{P} : there are two equations in X .

Let σ be a state of \mathcal{P} and let V be a variable that occurs in σ . The *value of V at σ* is then the (unique) number q such that the equation $V = q$ is one of the equations making up σ . For example, the value of X at the state

$$X = 4, \quad Y = 3, \quad Z = 3$$

is 4.

Suppose we have a program \mathcal{P} and a state σ of \mathcal{P} . In order to say what happens “next,” we also need to know which instruction of \mathcal{P} is about to be executed. We therefore define a *snapshot* or *instantaneous description* of a program \mathcal{P} of length n to be a pair (i, σ) where $1 \leq i \leq n + 1$, and σ is a state of \mathcal{P} . (Intuitively the number i indicates that it is the i th instruction which is about to be executed; $i = n + 1$ corresponds to a “stop” instruction.)

If $s = (i, \sigma)$ is a snapshot of \mathcal{P} and V is a variable of \mathcal{P} , then the *value of V at s* just means the value of V at σ .

A snapshot (i, σ) of a program \mathcal{P} of length n is called *terminal* if $i = n + 1$. If (i, σ) is a nonterminal snapshot of \mathcal{P} , we define the *successor* of (i, σ) to be the snapshot (j, τ) defined as follows:

Case 1. The i th instruction of \mathcal{P} is $V \leftarrow V + 1$ and σ contains the equation $V = m$. Then $j = i + 1$ and τ is obtained from σ by replacing the equation $V = m$ by $V = m + 1$ (i.e., the value of V at τ is $m + 1$).

Case 2. The i th instruction of \mathcal{P} is $V \leftarrow V - 1$ and σ contains the equation $V = m$. Then $j = i + 1$ and τ is obtained from σ by replacing the equation $V = m$ by $V = m - 1$ if $m \neq 0$; if $m = 0$, $\tau = \sigma$.

Case 3. The i th instruction of \mathcal{P} is $V \leftarrow V$. Then $\tau = \sigma$ and $j = i + 1$.

Case 4. The i th instruction of \mathcal{P} is IF $V \neq 0$ GOTO L . Then $\tau = \sigma$, and there are two subcases:

Case 4a. σ contains the equation $V = 0$. Then $j = i + 1$.

Case 4b. σ contains the equation $V = m$ where $m \neq 0$. Then, if there is an instruction of \mathcal{P} labeled L , j is the *least number* such that the j th instruction of \mathcal{P} is labeled L . Otherwise, $j = n + 1$.

For an example, we return to the program of (b), Section 2. Let σ be the state

$$X = 4, \quad Y = 0, \quad Z = 0$$

and let us compute the successor of the snapshots (i, σ) for various values of i .

For $i = 1$, the successor is $(4, \sigma)$ where σ is as above. For $i = 2$, the successor is $(3, \tau)$, where τ consists of the equations

$$X = 4, \quad Y = 0, \quad Z = 1.$$

For $i = 7$, the successor is $(8, \sigma)$. This is a terminal snapshot.

A *computation* of a program \mathcal{P} is defined to be a sequence (i.e., a list) s_1, s_2, \dots, s_k of snapshots of \mathcal{P} such that s_{i+1} is the successor of s_i for $i = 1, 2, \dots, k - 1$ and s_k is terminal.

Note that we have not forbidden a program to contain more than one instruction having the same label. However, our definition of successor of a snapshot, in effect, interprets a branch instruction as always referring to the *first* statement in the program having the label in question. Thus, for example, the program

[A]	$X \leftarrow X - 1$
	IF $X \neq 0$ GOTO A
[A]	$X \leftarrow X + 1$

is equivalent to the program

[A]	$X \leftarrow X - 1$
	IF $X \neq 0$ GOTO A
	$X \leftarrow X + 1$

Exercises

1. Let \mathcal{P} be the program of (b), Section 2. Write out a computation of \mathcal{P} beginning with the snapshot $(1, \sigma)$, where σ consists of the equations $X = 2, Y = 0, Z = 0$.
2. Give a program \mathcal{P} such that for every computation s_1, \dots, s_k of \mathcal{P} , $k = 5$.
3. Give a program \mathcal{P} such that for any $n \geq 0$ and every computation $s_1 = (1, \sigma), s_2, \dots, s_k$ of \mathcal{P} that has the equation $X = n$ in σ , $k = 2n + 1$.

4. Computable Functions

We have been speaking of the function computed by a program \mathcal{P} . It is now time to make this notion precise.

One would expect a program that computes a function of m variables to contain the input variables X_1, X_2, \dots, X_m , and the output variable Y , and to have all other variables (if any) in the program be local. Although this has been and will continue to be our practice, it is convenient not to make it a formal requirement. According to the definitions we are going to present, any program \mathcal{P} of the language \mathcal{S} can be used to compute a function of one variable, a function of two variables, and, in general, for each $m \geq 1$, a function of m variables.

Thus, let \mathcal{P} be any program in the language \mathcal{S} and let r_1, \dots, r_m be m given numbers. We form the state σ of \mathcal{P} which consists of the equations

$$X_1 = r_1, \quad X_2 = r_2, \quad \dots, \quad X_m = r_m, \quad Y = 0$$

together with the equations $V = 0$ for each variable V in \mathcal{P} other than X_1, \dots, X_m, Y . We will call this the *initial state*, and the snapshot $(1, \sigma)$, the *initial snapshot*.

- Case 1.* There is a computation s_1, s_2, \dots, s_k of \mathcal{P} beginning with the initial snapshot. Then we write $\psi_{\mathcal{P}}^{(m)}(r_1, r_2, \dots, r_m)$ for the value of the variable Y at the (terminal) snapshot s_k .
- Case 2.* There is no such computation; i.e., there is an infinite sequence s_1, s_2, s_3, \dots beginning with the initial snapshot where each s_{i+1} is the successor of s_i . In this case $\psi_{\mathcal{P}}^{(m)}(r_1, \dots, r_m)$ is undefined.

Let us reexamine the examples in Section 2 from the point of view of this definition. We begin with the program of (b). For this program \mathcal{P} , we have

$$\psi_{\mathcal{P}}^{(1)}(x) = x$$

for all x . For this one example, we give a detailed treatment. The following list of snapshots is a computation of \mathcal{P} :

$$\begin{aligned} &(1, \{X = r, Y = 0, Z = 0\}), \\ &(4, \{X = r, Y = 0, Z = 0\}), \\ &(5, \{X = r - 1, Y = 0, Z = 0\}), \\ &(6, \{X = r - 1, Y = 1, Z = 0\}), \\ &(7, \{X = r - 1, Y = 1, Z = 1\}), \\ &(1, \{X = r - 1, Y = 1, Z = 1\}), \\ &\vdots \\ &(1, \{X = 0, Y = r, Z = r\}), \\ &(2, \{X = 0, Y = r, Z = r\}), \\ &(3, \{X = 0, Y = r, Z = r + 1\}), \\ &(8, \{X = 0, Y = r, Z = r + 1\}). \end{aligned}$$

We have included a copy of \mathcal{P} showing line numbers:

[A]	IF $X \neq 0$ GOTO B	(1)
	$Z \leftarrow Z + 1$	(2)
	IF $Z \neq 0$ GOTO E	(3)
[B]	$X \leftarrow X - 1$	(4)
	$Y \leftarrow Y + 1$	(5)
	$Z \leftarrow Z + 1$	(6)
	IF $Z \neq 0$ GOTO A	(7)

For other examples of Section 2 we have

- (a) $\psi^{(1)}(r) = \begin{cases} 1 & \text{if } r = 0 \\ r & \text{otherwise,} \end{cases}$
- (b), (c) $\psi^{(1)}(r) = r,$
- (d) $\psi^{(2)}(r_1, r_2) = r_1 + r_2,$
- (e) $\psi^{(2)}(r_1, r_2) = r_1 \cdot r_2,$
- (f) $\psi^{(2)}(r_1, r_2) = \begin{cases} r_1 - r_2 & \text{if } r_1 \geq r_2 \\ \uparrow & \text{if } r_1 < r_2. \end{cases}$

Of course in several cases the programs written in Section 2 are abbreviations, and we are assuming that the appropriate macro expansions have been provided.

As indicated, we are permitting each program to be used with any number of inputs. If the program has n input variables, but only $m < n$ are specified, then according to the definition, the remaining input variables are assigned the value 0 and the computation proceeds. If on the other hand, m values are specified where $m > n$ the extra input values are ignored. For example, referring again to the examples from Section 2, we have

- (c) $\psi_{\mathcal{P}}^{(2)}(r_1, r_2) = r_1,$
- (d) $\psi_{\mathcal{P}}^{(1)}(r_1) = r_1 + 0 = r_1,$
- $\psi_{\mathcal{P}}^{(3)}(r_1, r_2, r_3) = r_1 + r_2.$

For any program \mathcal{P} and any positive integer m , the function $\psi_{\mathcal{P}}^{(m)}(x_1, \dots, x_m)$ is said to be *computed* by \mathcal{P} . A given partial function g (of one or more variables) is said to be *partially computable* if it is computed by some program. That is, g is partially computable if there is a program \mathcal{P} such that

$$g(r_1, \dots, r_m) = \psi_{\mathcal{P}}^{(m)}(r_1, \dots, r_m)$$

for all r_1, \dots, r_m . Here this equation must be understood to mean not only that both sides have the same value when they are defined, but also that when either side of the equation is undefined, the other is also.

As explained in Chapter 1, a given function g of m variables is called *total* if $g(r_1, \dots, r_m)$ is defined for *all* r_1, \dots, r_m . A function is said to be *computable* if it is both partially computable and total.

Partially computable functions are also called *partial recursive*, and computable functions, i.e., functions that are both total and partial recursive, are called *recursive*. The reason for this terminology is largely historical and will be discussed later.

Our examples from Section 2 give us a short list of partially computable functions, namely: x , $x + y$, $x \cdot y$, and $x - y$. Of these, all except the last one are total and hence computable.

Computability theory (also called recursion theory) studies the class of partially computable functions. In order to justify the name, we need some evidence that for every function which one can claim to be “computable” on intuitive grounds, there really is a program of the language \mathcal{S} which computes it. Such evidence will be developed as we go along.

We close this section with one final example of a program of \mathcal{S} :

[A] $X \leftarrow X + 1$
 IF $X \neq 0$ GOTO A

For this program \mathcal{P} , $\psi_{\mathcal{P}}^{(1)}(x)$ is undefined for all x . So, the nowhere defined function (see Chapter 1, Section 2) must be included in the class of partially computable functions.

Exercises

1. Let \mathcal{P} be the program

[A] IF $X \neq 0$ GOTO A
 X $\leftarrow X + 1$
 IF $X \neq 0$ GOTO A
 [A] Y $\leftarrow Y + 1$

What is $\psi_{\mathcal{P}}^{(1)}(x)$?

2. The same as Exercise 1 for the program

[B] IF $X \neq 0$ GOTO A
 Z $\leftarrow Z + 1$
 IF $Z \neq 0$ GOTO B
 [A] X $\leftarrow X$

3. The same as Exercise 1 for the empty program.

4. Let \mathcal{P} be the program

[A] Y $\leftarrow X_1$
 IF $X_2 = 0$ GOTO E
 Y $\leftarrow Y + 1$
 Y $\leftarrow Y + 1$
 $X_2 \leftarrow X_2 - 1$
 GOTO A

What is $\psi_{\mathcal{P}}^{(1)}(r_1)$? $\psi_{\mathcal{P}}^{(2)}(r_1, r_2)$? $\psi_{\mathcal{P}}^{(3)}(r_1, r_2, r_3)$?

5. Show that for every partially computable function $f(x_1, \dots, x_n)$, there is a number $m \geq 0$ such that f is computed by infinitely many programs of length m .

6. (a) For every number $k \geq 0$, let f_k be the constant function $f_k(x) = k$. Show that for every k , f_k is computable.
- (b) Let us call an \mathcal{S} program a *straightline program* if it contains no (labeled or unlabeled) instruction of the form IF $V \neq 0$ GOTO L . Show by induction on the length of programs that if the length of a straightline program \mathcal{P} is k , then $\psi_{\mathcal{P}}^{(1)}(x) \leq k$ for all x .
- (c) Show that, if \mathcal{P} is a straightline program that computes f_k , then the length of \mathcal{P} is at least k .
- (d) Show that no straightline \mathcal{S} program computes the function $f(x) = x + 1$. Conclude that the class of functions computable by straightline \mathcal{S} programs is contained in but is not equal to the class of computable functions.
7. Let us call an \mathcal{S} program \mathcal{P} *forward-branching* if the following condition holds for each occurrence in \mathcal{P} of a (labeled or unlabeled) instruction of the form IF $V \neq 0$ GOTO L . If IF $V \neq 0$ GOTO L is the i th instruction of \mathcal{P} , then either L does not appear as the label of an instruction in \mathcal{P} , or else, if j is the least number such that L is the label of the j th instruction in \mathcal{P} , then $i < j$. Show that a function is computed by some forward-branching program if and only if it is computed by some straightline program (see Exercise 6).
8. Let us call a unary function $f(x)$ *partially n -computable* if it is computed by some \mathcal{S} program \mathcal{P} such that \mathcal{P} has no more than n instructions, every variable in \mathcal{P} is among X, Y, Z_1, \dots, Z_n , and every label in \mathcal{P} is among A_1, \dots, A_n, E .
- (a) Show that if a unary function is computed by a program with no more than n instructions, then it is partially n -computable.
- (b) Show that for every $n \geq 0$, there are only finitely many distinct partially n -computable unary functions.
- (c) Show that for every $n \geq 0$, there are only finitely many distinct unary functions computed by \mathcal{S} programs of length no greater than n .
- (d) Conclude that for every $n \geq 0$, there is a partially computable unary function which is not computed by any \mathcal{S} program of length less than n .

5. More about Macros

In Section 2 we gave some examples of computable functions (i.e., $x + y$, $x \cdot y$) giving rise to corresponding macros. Now we consider this process in general.

Let $f(x_1, \dots, x_n)$ be some partially computable function computed by the program \mathcal{P} . We shall assume that the variables that occur in \mathcal{P} are all included in the list $Y, X_1, \dots, X_n, Z_1, \dots, Z_k$ and that the labels that occur in \mathcal{P} are all included in the list E, A_1, \dots, A_l . We also assume that for each instruction of \mathcal{P} of the form

IF $V \neq 0$ GOTO A_i

there is in \mathcal{P} an instruction labeled A_i . (In other words, E is the only “exit” label.) It is obvious that, if \mathcal{P} does not originally meet these conditions, it will after minor changes in notation. We write

$$\mathcal{P} = \mathcal{P}(Y, X_1, \dots, X_n, Z_1, \dots, Z_k; E, A_1, \dots, A_l)$$

in order that we can represent programs obtained from \mathcal{P} by replacing the variables and labels by others. In particular, we will write

$$\begin{aligned}\mathcal{Q}_m &= \mathcal{P}(Z_m, Z_{m+1}, \dots, Z_{m+n}, Z_{m+n+1}, \dots, Z_{m+n+k}; \\ &\quad E_m, A_{m+1}, \dots, A_{m+l})\end{aligned}$$

for each given value of m . Now we want to be able to use macros like

$$W \leftarrow f(V_1, \dots, V_n)$$

in our programs, where V_1, \dots, V_n, W can be any variables whatever. (In particular, W might be one of V_1, \dots, V_n .) We will take such a macro to be an abbreviation of the following expansion:

$$\begin{aligned}&Z_m \leftarrow 0 \\ &Z_{m+1} \leftarrow V_1 \\ &Z_{m+2} \leftarrow V_2 \\ &\vdots \\ &Z_{m+n} \leftarrow V_n \\ &Z_{m+n+1} \leftarrow 0 \\ &Z_{m+n+2} \leftarrow 0 \\ &\vdots \\ &Z_{m+n+k} \leftarrow 0 \\ &\mathcal{Q}_m \\ [E_m] \quad &W \leftarrow Z_m\end{aligned}$$

Here it is understood that the number m is chosen so large that none of the variables or labels used in \mathcal{Q}_m occur in the main program of which the expansion is a part. Notice that the expansion sets the variables corresponding to the output and local variables of \mathcal{P} equal to 0 and those corresponding to X_1, \dots, X_n equal to the values of V_1, \dots, V_n , respectively. Setting the variables equal to 0 is necessary (even though they are

all local variables automatically initialized to 0) because the expansion may be part of a loop in the main program; in this case, at the second and subsequent times through the loop the local variables will have whatever values they acquired the previous time around, and so will need to be reset. Note that when \mathcal{Q}_m terminates, the value of Z_m is $f(V_1, \dots, V_n)$, so that W finally does get the value $f(V_1, \dots, V_n)$.

If $f(V_1, \dots, V_n)$ is undefined, the program \mathcal{Q}_m will never terminate. Thus if f is not total, and the macro

$$W \leftarrow f(V_1, \dots, V_n)$$

is encountered in a program where V_1, \dots, V_n have values for which f is not defined, the main program will never terminate.

Here is an example:

$$Z \leftarrow X_1 - X_2$$

$$Y \leftarrow Z + X_3$$

This program computes the function $f(x_1, x_2, x_3)$, where

$$f(x_1, x_2, x_3) = \begin{cases} (x_1 - x_2) + x_3 & \text{if } x_1 \geq x_2 \\ \uparrow & \text{if } x_1 < x_2. \end{cases}$$

In particular, $f(2, 5, 6)$ is undefined, although $(2 - 5) + 6 = 3$ is positive. The computation never gets past the attempt to compute $2 - 5$.

So far we have augmented our language \mathcal{S} to permit the use of macros which allow assignment statements of the form

$$W \leftarrow f(V_1, \dots, V_n),$$

where f is any partially computable function. Nonetheless there is available only one highly restrictive conditional branch statement, namely,

IF $V \neq 0$ GOTO L

We will now see how to augment our language to include macros of the form

IF $P(V_1, \dots, V_n)$ GOTO L

where $P(x_1, \dots, x_n)$ is a computable predicate. Here we are making use of the convention, introduced in Chapter 1, that

TRUE = 1, FALSE = 0.

Hence predicates are just total functions whose values are always either 0 or 1. And therefore, it makes perfect sense to say that some given *predicate* is or is not computable.

Let $P(x_1, \dots, x_n)$ be any computable predicate. Then the appropriate macro expansion of

IF $P(V_1, \dots, V_n)$ GOTO L

is simply

$Z \leftarrow P(V_1, \dots, V_n)$
IF $Z \neq 0$ GOTO L

Note that P is a computable function and hence we have already shown how to expand the first instruction. The second instruction, being one of the basic instructions in the language \mathcal{S} , needs no further expansion.

A simple example of this general kind of conditional branch statement which we will use frequently is

IF $V = 0$ GOTO L

To see that this is legitimate we need only check that the predicate $P(x)$, defined by $P(x) = \text{TRUE}$ if $x = 0$ and $P(x) = \text{FALSE}$ otherwise, is computable. Since $\text{TRUE} = 1$ and $\text{FALSE} = 0$, the following program does the job:

IF $X \neq 0$ GOTO E
 $Y \leftarrow Y + 1$

The use of macros has the effect of enabling us to write much shorter programs than would be possible restricting ourselves to instructions of the original language \mathcal{S} . The original “assignment” statements $V \leftarrow V + 1$, $V \leftarrow V - 1$ are now augmented by general assignment statements of the form $W \leftarrow f(V_1, \dots, V_n)$ for any partially computable function f . Also, the original conditional branch statements IF $V \neq 0$ GOTO L are now augmented by general conditional branch statements of the form IF $P(V_1, \dots, V_n)$ GOTO L for any computable predicate P . The fact that any function which can be computed using these general instructions could already have been computed by a program of our original language \mathcal{S} (since the general instructions are merely abbreviations of programs of \mathcal{S}) is powerful evidence of the generality of our notion of computability.

Our next task will be to develop techniques that will make it easy to see that various particular functions are computable.

Exercises

1. (a) Use the process described in this section to expand the program in example (d) of Section 2.
 (b) What is the length of the \mathcal{S} program expanded from example (e) by this process?
 (c) Replace the instructions

$$\begin{aligned} Z_1 &\leftarrow X_1 + Y \\ Y &\leftarrow Z_1 \end{aligned}$$

in example (e) of Section 2 with the instruction $Y \leftarrow X_1 + Y$, and expand the result by the process described in this section. If \mathcal{P} is the resulting \mathcal{S} program, what is $\psi_{\mathcal{P}}^{(2)}(r_1, r_2)$?

3. Let $f(x), g(x)$ be computable functions and let $h(x) = f(g(x))$. Show that h is computable.
4. Show by constructing a program that the predicate $x_1 \leq x_2$ is computable.
5. Let $P(x)$ be a computable predicate. Show that the function f defined by

$$f(x_1, x_2) = \begin{cases} x_1 + x_2 & \text{if } P(x_1 + x_2) \\ \uparrow & \text{otherwise} \end{cases}$$

is partially computable.

6. Let $P(x)$ be a computable predicate. Show that

$$EX_P(r) = \begin{cases} 1 & \text{if there are at least } r \text{ numbers } n \text{ such that } P(n) = 1 \\ \uparrow & \text{otherwise} \end{cases}$$

is partially computable.

7. Let π be a computable permutation (i.e., one-one, onto function) of N , and let π^{-1} be the inverse of π , i.e.,

$$\pi^{-1}(y) = x \quad \text{if and only if} \quad \pi(x) = y.$$

Show that π^{-1} is computable.

8. Let $f(x)$ be a partially computable but not total function, let M be a finite set of numbers such that $f(m)\uparrow$ for all $m \in M$, and let $g(x)$ be

an arbitrary partially computable function. Show that

$$h(x) = \begin{cases} g(x) & \text{if } x \in M \\ f(x) & \text{otherwise} \end{cases}$$

is partially computable.

9. Let \mathcal{S}'^+ be a programming language that extends \mathcal{S}' by permitting instructions of the form $V \leftarrow k$, for any $k \geq 0$. These instructions have the obvious effect of setting the value of V to k . Show that a function is partially computable by some \mathcal{S}'^+ program if and only if it is partially computable.
10. Let \mathcal{S}' be a programming language defined like \mathcal{S} except that its (labeled and unlabeled) instructions are of the three types

$$\begin{aligned} V &\leftarrow V' \\ V &\leftarrow V + 1 \\ \text{If } V &\neq V' \text{ GOTO } L \end{aligned}$$

These instructions are given the obvious meaning. Show that a function is partially computable in \mathcal{S}' if and only if it is partially computable.

3

Primitive Recursive Functions

1. Composition

We want to combine computable functions in such a way that the output of one becomes an input to another. In the simplest case we combine functions f and g to obtain the function

$$h(x) = f(g(x)).$$

More generally, for functions of several variables:

Definition. Let f be a function of k variables and let g_1, \dots, g_k be functions of n variables. Let

$$h(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n)).$$

Then h is said to be obtained from f and g_1, \dots, g_k by *composition*.

Of course, the functions f, g_1, \dots, g_k need not be total. $h(x_1, \dots, x_n)$ will be defined when all of $z_1 = g_1(x_1, \dots, x_n), \dots, z_k = g_k(x_1, \dots, x_n)$ are defined and also $f(z_1, \dots, z_k)$ is defined.

Using macros it is very easy to prove

Theorem 1.1. If h is obtained from the (partially) computable functions f, g_1, \dots, g_k by composition, then h is (partially) computable.

The word *partially* is placed in parentheses in order to assert the correctness of the statement with the word included or omitted in both places.

Proof. The following program obviously computes h :

$$\begin{aligned} Z_1 &\leftarrow g_1(X_1, \dots, X_n) \\ &\vdots \\ Z_k &\leftarrow g_k(X_1, \dots, X_n) \\ Y &\leftarrow f(Z_1, \dots, Z_k) \end{aligned}$$

If f, g_1, \dots, g_k are not only partially computable but are also total, then so is h . ■

By Section 4 of Chapter 2, we know that x , $x + y$, $x \cdot y$, and $x - y$ are partially computable. So by Theorem 1.1 we see that $2x = x + x$ and $4x^2 = (2x) \cdot (2x)$ are computable. So are $4x^2 + 2x$ and $4x^2 - 2x$. Note that $4x^2 - 2x$ is total, although it is obtained from the nontotal function $x - y$ by composition with $4x^2$ and $2x$.

2. Recursion

Suppose k is some fixed number and

$$\begin{aligned} h(0) &= k, \\ h(t+1) &= g(t, h(t)), \end{aligned} \tag{2.1}$$

where g is some given *total* function of two variables. Then h is said to be obtained from g by *primitive recursion*, or simply *recursion*.¹

Theorem 2.1. Let h be obtained from g as in (2.1), and let g be computable. Then h is also computable.

Proof. We first note that the constant function $f(x) = k$ is computable; in fact, it is computed by the program

$$\left. \begin{array}{l} Y \leftarrow Y + 1 \\ Y \leftarrow Y + 1 \\ \vdots \\ Y \leftarrow Y + 1 \end{array} \right\} \text{ } k \text{ lines}$$

¹ Primitive recursion, characterized by Equations (2.1) and (2.2), is just one specialized form of recursion, but it is the only one we will be concerned with in this chapter, so we will refer to it simply as *recursion*. We will consider more general forms of recursion in Part 5.

Hence we have available the macro $Y \leftarrow k$. The following is a program that computes $h(x)$:

```

 $Y \leftarrow k$ 
[A]    IF  $X = 0$  GOTO E
 $Y \leftarrow g(Z, Y)$ 
 $Z \leftarrow Z + 1$ 
 $X \leftarrow X - 1$ 
GOTO A

```

To see that this program does what it is supposed to do, note that, if Y has the value $h(z)$ before executing the instruction labeled A , then it has the value $g(z, h(z)) = h(z + 1)$ after executing the instruction $Y \leftarrow g(Z, Y)$. Since Y is initialized to $k = h(0)$, Y successively takes on the values $h(0), h(1), \dots, h(x)$ and then terminates. ■

A slightly more complicated kind of recursion is involved when we have

$$\begin{aligned} h(x_1, \dots, x_n, 0) &= f(x_1, \dots, x_n), \\ h(x_1, \dots, x_n, t+1) &= g(t, h(x_1, \dots, x_n, t), x_1, \dots, x_n). \end{aligned} \tag{2.2}$$

Here the function h of $n + 1$ variables is said to be obtained by *primitive recursion*, or simply *recursion*, from the total functions f (of n variables) and g (of $n + 2$ variables). The recursion (2.2) is just like (2.1) except that parameters x_1, \dots, x_n are involved. Again we have

Theorem 2.2. Let h be obtained from f and g as in (2.2) and let f, g be computable. Then h is also computable.

Proof. The proof is almost the same as for Theorem 2.1. The following program computes $h(x_1, \dots, x_n, x_{n+1})$:

```

 $Y \leftarrow f(X_1, \dots, X_n)$ 
[A]    IF  $X_{n+1} = 0$  GOTO E
 $Y \leftarrow g(Z, Y, X_1, \dots, X_n)$ 
 $Z \leftarrow Z + 1$ 
 $X_{n+1} \leftarrow X_{n+1} - 1$ 
GOTO A

```

3. PRC Classes

So far we have considered the operations of composition and recursion. Now we need some functions on which to get started. These will be

$$s(x) = x + 1,$$

$$n(x) = 0,$$

and the *projection functions*

$$u_i^n(x_1, \dots, x_n) = x_i, \quad 1 \leq i \leq n.$$

[For example, $u_3^4(x_1, x_2, x_3, x_4) = x_3$.] The functions s , n , and u_i^n are called the *initial functions*.

Definition. A class of total functions \mathcal{C} is called a *PRC² class* if

1. the initial functions belong to \mathcal{C} ,
2. a function obtained from functions belonging to \mathcal{C} by either composition or recursion also belongs to \mathcal{C} .

Then we have

Theorem 3.1. The class of computable functions is a PRC class.

Proof. By Theorems 1.1, 2.1, and 2.2, we need only verify that the initial functions are computable.

Now this is obvious; $s(x) = x + 1$ is computed by

$$Y \leftarrow X + 1$$

$n(x)$ is computed by the empty program, and $u_i^n(x_1, \dots, x_n)$ is computed by the program

$$Y \leftarrow X_i$$

■

Definition. A function is called *primitive recursive* if it can be obtained from the initial functions by a finite number of applications of composition and recursion.

It is obvious from this definition that

² This is an abbreviation for “primitive recursively closed.”

Corollary 3.2. The class of primitive recursive functions is a PRC class.

Actually we can say more:

Theorem 3.3. A function is primitive recursive if and only if it belongs to every PRC class.

Proof. If a function belongs to every PRC class, then, in particular, by Corollary 3.2, it belongs to the class of primitive recursive functions.

Conversely let a function f be a primitive recursive function and let \mathcal{C} be some PRC class. We want to show that f belongs to \mathcal{C} . Since f is a primitive recursive function, there is a list f_1, f_2, \dots, f_n of functions such that $f_n = f$ and each f_i in the list is either an initial function or can be obtained from preceding functions in the list by composition or recursion. Now the initial functions certainly belong to the PRC class \mathcal{C} . Moreover the result of applying composition or recursion to functions in \mathcal{C} is again a function belonging to \mathcal{C} . Hence each function in the list f_1, \dots, f_n belongs to \mathcal{C} . Since $f_n = f$, f belongs to \mathcal{C} . ■

Corollary 3.4. Every primitive recursive function is computable.

Proof. By the theorem just proved, every primitive recursive function belongs to the PRC class of computable functions. ■

In Chapter 4 we shall show how to obtain a computable function that is not primitive recursive. Hence it will follow that the set of primitive recursive functions is a proper subset of the set of computable functions.

Exercises

- Let \mathcal{C} be a PRC class, and let g_1, g_2, g_3, g_4 belong to \mathcal{C} . Show that if

$$h_1(x, y, z) = g_1(z, y, x),$$

$$h_2(x) = g_2(x, x, x), \text{ and}$$

$$h_3(w, x, y, z) = h_1(g_3(w, y), z, g_4(2, g_4(y, z))),$$

then h_2, h_3 also belong to \mathcal{C} .

- Show that the class of all total functions is a PRC class.
- Let $n > 0$ be some given number, and let \mathcal{C} be a class of total functions of no more than n variables. Show that \mathcal{C} is not a PRC class.

4. Let \mathcal{C} be a PRC class, let h belong to \mathcal{C} , and let

$$\begin{aligned} f(x) &= h(g(x)) \quad \text{and} \\ g(x) &= h(f(x)). \end{aligned}$$

Show that f belongs to \mathcal{C} if and only if g belongs to \mathcal{C} .

5. Prove Corollary 3.4 directly from Theorems 1.1, 2.1, 2.2, and the proof of Theorem 3.1.

4. Some Primitive Recursive Functions

We proceed to make a short list of primitive recursive functions. Being primitive recursive, they are also computable.

1. $x + y$

To see that this is primitive recursive, we have to show how to obtain this function from the initial functions using only the operations of composition and recursion.

If we write $f(x, y) = x + y$, we have the recursion equations

$$\begin{aligned} f(x, 0) &= x, \\ f(x, y + 1) &= f(x, y) + 1. \end{aligned}$$

We can rewrite these equations as

$$\begin{aligned} f(x, 0) &= u_1^1(x), \\ f(x, y + 1) &= g(y, f(x, y), x), \end{aligned}$$

where $g(x_1, x_2, x_3) = s(u_2^3(x_1, x_2, x_3))$. The functions $u_1^1(x)$, $u_2^3(x_1, x_2, x_3)$, and $s(x)$ are primitive recursive functions; in fact they are initial functions. Also, $g(x_1, x_2, x_3)$ is a primitive recursive function, since it is obtained by composition of primitive recursive functions. Thus, the preceding is a valid application of the operation of recursion to primitive recursive functions. Hence $f(x, y) = x + y$ is primitive recursive.

Of course we already knew that $x + y$ was a computable function. So we have only obtained the additional information that it is in fact primitive recursive.

2. $x \cdot y$

The recursion equations for $h(x, y) = x \cdot y$ are

$$\begin{aligned} h(x, 0) &= 0, \\ h(x, y + 1) &= h(x, y) + x. \end{aligned}$$

This can be rewritten

$$\begin{aligned} h(x, 0) &= n(x) \\ h(x, y + 1) &= g(y, h(x, y), x). \end{aligned}$$

Here, $n(x)$ is the zero function,

$$g(x_1, x_2, x_3) = f(u_2^3(x_1, x_2, x_3), u_3^3(x_1, x_2, x_3)),$$

$f(x_1, x_2)$ is $x_1 + x_2$, and $u_2^3(x_1, x_2, x_3), u_3^3(x_1, x_2, x_3)$ are projection functions. Notice that the functions $n(x), u_2^3(x_1, x_2, x_3)$, and $u_3^3(x_1, x_2, x_3)$ are all primitive recursive functions, since they are all initial functions. We have just shown that $f(x_1, x_2) = x_1 + x_2$ is primitive recursive, so $g(x_1, x_2, x_3)$ is a primitive recursive function since it is obtained from primitive recursive functions by composition. Finally, we conclude that

$$h(x, y) = x \cdot y$$

is primitive recursive.

3. $x!$

The recursion equations are

$$\begin{aligned} 0! &= 1, \\ (x + 1)! &= x! \cdot s(x). \end{aligned}$$

More precisely, $x! = h(x)$, where

$$\begin{aligned} h(0) &= 1, \\ h(t + 1) &= g(t, h(t)), \end{aligned}$$

and

$$g(x_1, x_2) = s(x_1) \cdot x_2.$$

Finally, g is primitive recursive because

$$g(x_1, x_2) = s(u_1^2(x_1, x_2)) \cdot u_2^2(x_1, x_2)$$

and multiplication is already known to be primitive recursive.

In the examples that follow, we leave it to the reader to check that the recursion equations can be put in the precise form called for by the definition of the operation of recursion.

4. x^y

The recursion equations are

$$\begin{aligned}x^0 &= 1, \\x^{y+1} &= x^y \cdot x.\end{aligned}$$

Note that these equations assign the value 1 to the “indeterminate” 0^0 .

5. $p(x)$

The *predecessor function* $p(x)$ is defined as follows:

$$p(x) = \begin{cases} x - 1 & \text{if } x \neq 0 \\ 0 & \text{if } x = 0. \end{cases}$$

It corresponds to the instruction in our programming language $X \leftarrow X - 1$.

The recursion equations for $p(x)$ are simply

$$\begin{aligned}p(0) &= 0, \\p(t + 1) &= t.\end{aligned}$$

Hence, $p(x)$ is primitive recursive.

6. $x \dot{-} y$

The function $x \dot{-} y$ is defined as follows:

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y. \end{cases}$$

This function should not be confused with the function $x - y$, which is undefined if $x < y$. In particular, $x \dot{-} y$ is total, while $x - y$ is not.

We show that $x \dot{-} y$ is primitive recursive by displaying the recursion equations:

$$\begin{aligned}x \dot{-} 0 &= x, \\x \dot{-} (t + 1) &= p(x \dot{-} t).\end{aligned}$$

7. $|x - y|$

The function $|x - y|$ is defined as the absolute value of the difference between x and y . It can be expressed simply as

$$|x - y| = (x \dot{-} y) + (y \dot{-} x)$$

and thus is primitive recursive.

8. $\alpha(x)$

The function $\alpha(x)$ is defined as

$$\alpha(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0. \end{cases}$$

$\alpha(x)$ is primitive recursive since

$$\alpha(x) = 1 \dot{-} x.$$

Or we can simply write the recursion equations:

$$\begin{aligned}\alpha(0) &= 1, \\ \alpha(t + 1) &= 0.\end{aligned}$$

Exercises

1. Give a detailed argument that x^y , $p(x)$, and $x \dot{-} y$ are primitive recursive.
2. Show that for each k , the function $f(x) = k$ is primitive recursive.
3. Prove that if $f(x)$ and $g(x)$ are primitive recursive functions, so is $f(x) + g(x)$.
4. Without using $x + y$ as a macro, apply the constructions in the proofs of Theorems 1.1, 2.2, and 3.1 to give an \mathcal{S} program that computes $x \cdot y$.
5. For any unary function $f(x)$, the n th iteration of f , written f^n , is

$$f^n(x) = f(\cdots f(x) \cdots),$$

where f is composed with itself n times on the right side of the equation. (Note that $f^0(x) = x$.) Let $\iota_f(n, x) = f^n(x)$. Show that if f is primitive recursive, then ι_f is also primitive recursive.

- 6.* (a) Let $E(x) = 0$ if x is even, $E(x) = 1$ if x is odd. Show that $E(x)$ is primitive recursive.
 (b) Let $H(x) = x/2$ if x is even, $(x - 1)/2$ if x is odd. Show that $H(x)$ is primitive recursive.
- 7.* Let $f(0) = 0$, $f(1) = 1$, $f(2) = 2^2$, $f(3) = 3^{3^3} = 3^{27}$, etc. In general, $f(n)$ is written as a stack n high, of n 's as exponents. Show that f is primitive recursive.

- 8.* Let k be some fixed number, let f be a function such that $f(x + 1) < x + 1$ for all x , and let

$$\begin{aligned} h(0) &= k \\ h(t + 1) &= g(h(f(t + 1))). \end{aligned}$$

Show that if f and g belong to some PRC class \mathcal{C} , then so does h . [Hint: Define $f'(x) = \min_{t \leq x} f'(x) = 0$. See Exercise 5 for the definition of $f'(x)$.]

- 9.* Let $g(x)$ be a primitive recursive function and let $f(0, x) = g(x)$, $f(n + 1, x) = f(n, f(n, x))$. Prove that $f(n, x)$ is primitive recursive.
- 10.* Let COMP be the class of functions obtained from the initial functions by a finite sequence of compositions.

- (a) Show that for every function $f(x_1, \dots, x_n)$ in COMP, either $f(x_1, \dots, x_n) = k$ for some constant k , or $f(x_1, \dots, x_n) = x_i + k$ for some $1 \leq i \leq n$ and some constant k .
- (b) An n -ary function f is *monotone* if for all n -tuples (x_1, \dots, x_n) , (y_1, \dots, y_n) such that $x_i \leq y_i$, $1 \leq i \leq n$, $f(x_1, \dots, x_n) \leq f(y_1, \dots, y_n)$. Show that every function in COMP is monotone.
- (c) Show that COMP is a proper subset of the class of primitive recursive functions.
- (d) Show that the class of functions computed by straightline \mathcal{S} programs is a proper subset of COMP. [See Exercise 4.6 in Chapter 2 for the definition of straightline programs.]

- 11.* Let \mathcal{P}_1 be the class of all functions obtained from the initial functions by any finite number of compositions and no more than one recursion (in any order).

- (a) Let $f(x_1, \dots, x_n)$ belong to COMP. [See Exercise 10 for the definition of COMP.] Show that there is a $k > 0$ such that $f(x_1, \dots, x_n) \leq \max\{x_1, \dots, x_n\} + k$.
- (b) Let

$$\begin{aligned} h(0) &= c \\ h(t + 1) &= g(t, h(t)), \end{aligned}$$

where c is some given number and g belongs to COMP. Show that there is a $k > 0$ such that $h(t) \leq tk + c$.

- (c) Let

$$\begin{aligned} h(x_1, \dots, x_n, 0) &= f(x_1, \dots, x_n) \\ h(x_1, \dots, x_n, t + 1) &= g(t, h(x_1, \dots, x_n, t), x_1, \dots, x_n), \end{aligned}$$

where f, g belong to COMP. Show that there are $k, l > 0$ such that $h(x_1, \dots, x_n, t) \leq tk + \max\{x_1, \dots, x_n\} + l$.

- (d) Let $f(x_1, \dots, x_n)$ belong to \mathcal{P}_1 . Show that there are $k, l > 0$ such that $f(x_1, \dots, x_n) \leq \max\{x_1, \dots, x_n\} \cdot k + l$.
- (e) Show that \mathcal{P}_1 is a proper subset of the class of primitive recursive functions.

5. Primitive Recursive Predicates

We recall from Chapter 1, Section 4, that predicates or Boolean-valued functions are simply total functions whose values are 0 or 1. (We have identified 1 with TRUE and 0 with FALSE.) Thus we can speak without further ado of primitive recursive predicates.

We continue our list of primitive recursive functions, including some that are predicates.

9. $x = y$

The predicate $x = y$ is defined as 1 if the values of x and y are the same and 0 otherwise. Thus we wish to show that the function

$$d(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

is primitive recursive. This follows immediately from the equation

$$d(x, y) = \alpha(|x - y|).$$

10. $x \leq y$

This predicate is simply the primitive recursive function $\alpha(x \dot{-} y)$.

Theorem 5.1. Let \mathcal{C} be a PRC class. If P, Q are predicates that belong to \mathcal{C} , then so are $\sim P$, $P \vee Q$, and $P \& Q$.³

Proof. Since $\sim P = \alpha(P)$, it follows that $\sim P$ belongs to \mathcal{C} . (α was defined in Section 4, item 8.)

³ See Chapter 1, Section 4.

Also, we have

$$P \ \& \ Q = P \cdot Q,$$

so that $P \ \& \ Q$ belongs to \mathcal{C} .

Finally, the De Morgan law

$$P \vee Q \Leftrightarrow \sim(\sim P \ \& \ \sim Q)$$

shows, using what we have already done, that $P \vee Q$ belongs to \mathcal{C} . ■

A result like Theorem 5.1 which refers to PRC classes can be applied to the two classes we have shown to be PRC. That is, taking \mathcal{C} to be the class of all primitive recursive functions, we have

Corollary 5.2. If P, Q are primitive recursive predicates, then so are $\sim P$, $P \vee Q$, and $P \ \& \ Q$.

Similarly taking \mathcal{C} to be the class of all computable functions, we have

Corollary 5.3. If P, Q are computable predicates, then so are $\sim P$, $P \vee Q$, and $P \ \& \ Q$.

As a simple example we have

$$II. \quad x < y$$

We can write

$$x < y \Leftrightarrow x \leq y \ \& \ \sim(x = y),$$

or more simply

$$x < y \Leftrightarrow \sim(y \leq x).$$

Theorem 5.4 (Definition by Cases). Let \mathcal{C} be a PRC class. Let the functions g, h and the predicate P belong to \mathcal{C} . Let

$$f(x_1, \dots, x_n) = \begin{cases} g(x_1, \dots, x_n) & \text{if } P(x_1, \dots, x_n) \\ h(x_1, \dots, x_n) & \text{otherwise.} \end{cases}$$

Then f belongs to \mathcal{C} .

This will be recognized as a version of the familiar “if...then..., else...” statement.

Proof. The result is obvious because

$$f(x_1, \dots, x_n)$$

$$= g(x_1, \dots, x_n) \cdot P(x_1, \dots, x_n) + h(x_1, \dots, x_n) \cdot \alpha(P(x_1, \dots, x_n)).$$

■

Corollary 5.5. Let \mathcal{C} be a PRC class, let n -ary functions g_1, \dots, g_m, h and predicates P_1, \dots, P_m belong to \mathcal{C} , and let

$$P_i(x_1, \dots, x_n) \& P_j(x_1, \dots, x_n) = 0$$

for all $1 \leq i < j \leq m$ and all x_1, \dots, x_n . If

$$f(x_1, \dots, x_n) = \begin{cases} g_1(x_1, \dots, x_n) & \text{if } P_1(x_1, \dots, x_n) \\ \vdots & \vdots \\ g_m(x_1, \dots, x_n) & \text{if } P_m(x_1, \dots, x_n) \\ h(x_1, \dots, x_n) & \text{otherwise,} \end{cases}$$

then f also belongs to \mathcal{C} .

Proof. We argue by induction on m . The case for $m = 1$ is given by Theorem 5.4, so let

$$f(x_1, \dots, x_n) = \begin{cases} g_1(x_1, \dots, x_n) & \text{if } P_1(x_1, \dots, x_n) \\ \vdots & \vdots \\ g_{m+1}(x_1, \dots, x_n) & \text{if } P_{m+1}(x_1, \dots, x_n) \\ h(x_1, \dots, x_n) & \text{otherwise,} \end{cases}$$

and let

$$h'(x_1, \dots, x_n) = \begin{cases} g_{m+1}(x_1, \dots, x_n) & \text{if } P_{m+1}(x_1, \dots, x_n) \\ h(x_1, \dots, x_n) & \text{otherwise.} \end{cases}$$

Then

$$f(x_1, \dots, x_n) = \begin{cases} g_1(x_1, \dots, x_n) & \text{if } P_1(x_1, \dots, x_n) \\ \vdots & \vdots \\ g_m(x_1, \dots, x_n) & \text{if } P_m(x_1, \dots, x_n) \\ h'(x_1, \dots, x_n) & \text{otherwise,} \end{cases}$$

and h' belongs to \mathcal{C} by Theorem 5.4, so f belongs to \mathcal{C} by the induction hypothesis. ■

Exercise

- Let us call a predicate *trivial* if it is always TRUE or always FALSE. Show that no nontrivial predicates belong to COMP (see Exercise 4.10 for the definition of COMP.)

6. Iterated Operations and Bounded Quantifiers

Theorem 6.1. Let \mathcal{C} be a PRC class. If $f(t, x_1, \dots, x_n)$ belongs to \mathcal{C} , then so do the functions

$$g(y, x_1, \dots, x_n) = \sum_{t=0}^y f(t, x_1, \dots, x_n)$$

and

$$h(y, x_1, \dots, x_n) = \prod_{t=0}^y f(t, x_1, \dots, x_n).$$

A common error is to attempt to prove this by using mathematical induction on y . A little reflection reveals that such an argument by induction shows that

$$g(0, x_1, \dots, x_n), g(1, x_1, \dots, x_n), \dots$$

all belong to \mathcal{C} , but not that the function $g(y, x_1, \dots, x_n)$, one of whose arguments is y , belongs to \mathcal{C} .

We proceed with the correct proof.

Proof. We note the recursion equations

$$g(0, x_1, \dots, x_n) = f(0, x_1, \dots, x_n),$$

$$g(t + 1, x_1, \dots, x_n) = g(t, x_1, \dots, x_n) + f(t + 1, x_1, \dots, x_n),$$

and recall that since $+$ is primitive recursive, it belongs to \mathcal{C} .

Similarly,

$$h(0, x_1, \dots, x_n) = f(0, x_1, \dots, x_n),$$

$$h(t + 1, x_1, \dots, x_n) = h(t, x_1, \dots, x_n) \cdot f(t + 1, x_1, \dots, x_n). \quad \blacksquare$$

Sometimes we will want to begin the summation (or product) at 1 instead of 0. That is, we will want to consider

$$g(y, x_1, \dots, x_n) = \sum_{t=1}^y f(t, x_1, \dots, x_n)$$

or

$$h(y, x_1, \dots, x_n) = \prod_{t=1}^y f(t, x_1, \dots, x_n).$$

Then the initial recursion equations can be taken to be

$$g(0, x_1, \dots, x_n) = 0,$$

$$h(0, x_1, \dots, x_n) = 1,$$

with the equations for $g(t + 1, x_1, \dots, x_n)$ and $h(t + 1, x_1, \dots, x_n)$ as in the preceding proof. Note that we are implicitly defining a vacuous sum to be 0 and a vacuous product to be 1. With this understanding we have proved

Corollary 6.2. If $f(t, x_1, \dots, x_n)$ belongs to the PRC class \mathcal{C} , then so do the functions

$$g(y, x_1, \dots, x_n) = \sum_{t=1}^y f(t, x_1, \dots, x_n)$$

and

$$h(y, x_1, \dots, x_n) = \prod_{t=1}^y f(t, x_1, \dots, x_n).$$

We have

Theorem 6.3. If the predicate $P(t, x_1, \dots, x_n)$ belongs to some PRC class \mathcal{C} , then so do the predicates⁴

$$(\forall t)_{\leq y} P(t, x_1, \dots, x_n) \quad \text{and} \quad (\exists t)_{\leq y} P(t, x_1, \dots, x_n).$$

Proof. We need only observe that

$$(\forall t)_{\leq y} P(t, x_1, \dots, x_n) \Leftrightarrow \left[\prod_{t=0}^y P(t, x_1, \dots, x_n) \right] = 1$$

and

$$(\exists t)_{\leq y} P(t, x_1, \dots, x_n) \Leftrightarrow \left[\sum_{t=0}^y P(t, x_1, \dots, x_n) \right] \neq 0. \quad \blacksquare$$

Actually for the universal quantifier it would even have been correct to write the equation

$$(\forall t)_{\leq y} P(t, x_1, \dots, x_n) = \prod_{t=0}^y P(t, x_1, \dots, x_n).$$

⁴ See Chapter 1, Section 5.

Sometimes in applying Theorem 6.3 we want to use the quantifier

$$(\forall t)_{< y} \quad \text{or} \quad (\exists t)_{< y}.$$

That the theorem is still valid is clear from the relations

$$(\exists t)_{< y} P(t, x_1, \dots, x_n) \Leftrightarrow (\exists t)_{\leq y} [t \neq y \ \& \ P(t, x_1, \dots, x_n)],$$

$$(\forall t)_{< y} P(t, x_1, \dots, x_n) \Leftrightarrow (\forall t)_{\leq y} [t = y \ \vee \ P(t, x_1, \dots, x_n)].$$

We continue our list of examples.

12. $y|x$

This is the predicate “ y is a divisor of x .” For example,

$$3|12 \quad \text{is true}$$

while

$$3|13 \quad \text{is false.}$$

The predicate is primitive recursive since

$$y|x \Leftrightarrow (\exists t)_{\leq x} (y \cdot t = x).$$

13. Prime(x)

The predicate “ x is a prime” is primitive recursive since

$$\text{Prime}(x) \Leftrightarrow x > 1 \ \& \ (\forall t)_{\leq x} \{t = 1 \ \vee \ t = x \ \vee \ \sim (t|x)\}.$$

(A number is a *prime* if it is greater than 1 and it has no divisors other than 1 and itself.)

Exercises

- Let $f(x) = 2x$ if x is a perfect square; $f(x) = 2x + 1$ otherwise. Show that f is primitive recursive.
- Let $\sigma(x)$ be the sum of the divisors of x if $x \neq 0$; $\sigma(0) = 0$ [e.g., $\sigma(6) = 1 + 2 + 3 + 6 = 12$]. Show that $\sigma(x)$ is primitive recursive.
- Let $\pi(x)$ be the number of primes that are $\leq x$. Show that $\pi(x)$ is primitive recursive.
- Let SQSM(x) be true if x is the sum of two perfect squares; false otherwise. Show that SQSM(x) is primitive recursive.

5. Let \mathcal{C} be a PRC class, let $P(t, x_1, \dots, x_n)$ be a predicate in \mathcal{C} , and let

$$g(y, z, x_1, \dots, x_n) = (\forall t)_{y \leq t \leq z} P(t, x_1, \dots, x_n) \text{ and}$$

$$h(y, z, x_1, \dots, x_n) = (\exists t)_{y \leq t \leq z} P(t, x_1, \dots, x_n),$$

(where $(\forall t)_{y \leq t \leq z} P(t, x_1, \dots, x_n)$ and $(\exists t)_{y \leq t \leq z} P(t, x_1, \dots, x_n)$ mean that $P(t, x_1, \dots, x_n)$ is true for all t (respectively, for some t) from y to z). Show that g, h also belong to \mathcal{C} .

6. Let $\text{RP}(x, y)$ be true if x and y are relatively prime (i.e., their greatest common divisor is 1). Show that $\text{RP}(x, y)$ is primitive recursive.
7. Give a sequence of compositions and recursions that shows explicitly that $\text{Prime}(x)$ is primitive recursive.

7. Minimalization

Let $P(t, x_1, \dots, x_n)$ belong to some given PRC class \mathcal{C} . Then by Theorem 6.1, the function

$$g(y, x_1, \dots, x_n) = \sum_{u=0}^y \prod_{t=0}^u \alpha(P(t, x_1, \dots, x_n))$$

also belongs to \mathcal{C} . (Recall that the primitive recursive function α was defined in Section 4.) Let us analyze this function g . Suppose for definiteness that for some value of $t_0 \leq y$,

$$P(t, x_1, \dots, x_n) = 0 \quad \text{for } t < t_0,$$

but

$$P(t_0, x_1, \dots, x_n) = 1,$$

i.e., that t_0 is the least value of $t \leq y$ for which $P(t, x_1, \dots, x_n)$ is true. Then

$$\prod_{t=0}^u \alpha(P(t, x_1, \dots, x_n)) = \begin{cases} 1 & \text{if } u < t_0 \\ 0 & \text{if } u \geq t_0. \end{cases}$$

Hence,

$$g(y, x_1, \dots, x_n) = \sum_{u < t_0} 1 = t_0,$$

so that $g(y, x_1, \dots, x_n)$ is the least value of t for which $P(t, x_1, \dots, x_n)$ is true. Now, we define

$$\min_{t \leq y} P(t, x_1, \dots, x_n) = \begin{cases} g(y, x_1, \dots, x_n) & \text{if } (\exists t)_{\leq y} P(t, x_1, \dots, x_n) \\ 0 & \text{otherwise.} \end{cases}$$

Thus, $\min_{t \leq y} P(t, x_1, \dots, x_n)$ is the least value of $t \leq y$ for which $P(t, x_1, \dots, x_n)$ is true, if such exists; otherwise it assumes the (default) value 0. Using Theorems 5.4 and 6.3, we have

Theorem 7.1. If $P(t, x_1, \dots, x_n)$ belongs to some PRC class \mathcal{C} and $f(y, x_1, \dots, x_n) = \min_{t \leq y} P(t, x_1, \dots, x_n)$, then f also belongs to \mathcal{C} .

The operation “ $\min_{t \leq y}$ ” is called *bounded minimization*.

Continuing our list:

14. $\lfloor x/y \rfloor$

$\lfloor x/y \rfloor$ is the “integer part” of the quotient x/y . For example, $\lfloor 7/2 \rfloor = 3$ and $\lfloor 2/3 \rfloor = 0$. The equation

$$\lfloor x/y \rfloor = \min_{t \leq x} [(t + 1) \cdot y > x]$$

shows that $\lfloor x/y \rfloor$ is primitive recursive. Note that according to this equation, we are taking $\lfloor x/0 \rfloor = 0$.

15. $R(x, y)$

$R(x, y)$ is the *remainder* when x is divided by y . Since

$$\frac{x}{y} = \lfloor x/y \rfloor + \frac{R(x, y)}{y},$$

we can write

$$R(x, y) = x \dot{-} (y \cdot \lfloor x/y \rfloor),$$

so that $R(x, y)$ is primitive recursive. [Note that $R(x, 0) = x$.]

16. p_n

Here, for $n > 0$, p_n is the n th prime number (in order of size). So that p_n be a total function, we set $p_0 = 0$. Thus, $p_0 = 0$, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, etc.

Consider the recursion equations

$$p_0 = 0,$$

$$p_{n+1} = \min_{t \leq p_n! + 1} [\text{Prime}(t) \& t > p_n].$$

To see that these equations are correct we must verify the inequality

$$p_{n+1} \leq (p_n)! + 1. \quad (7.1)$$

To do so note that for $0 < i \leq n$ we have

$$\frac{(p_n)! + 1}{p_i} = K + \frac{1}{p_i},$$

where K is an integer. Hence $(p_n)! + 1$ is not divisible by any of the primes p_1, p_2, \dots, p_n . So, either $(p_n)! + 1$ is itself a prime or it is divisible by a prime $> p_n$. In either case there is a prime q such that $p_n < q \leq (p_n)! + 1$, which gives the inequality (7.1). (This argument is just Euclid's proof that there are infinitely many primes.)

Before we can confidently assert that p_n is a primitive recursive function, we need to justify the interleaving of the recursion equations with bounded minimalization. To do so, we first define the primitive recursive function

$$h(y, z) = \min_{t \leq z} [\text{Prime}(t) \& t > y].$$

Then we set

$$k(x) = h(x, x! + 1),$$

another primitive recursive function. Finally, our recursion equations reduce to

$$p_0 = 0,$$

$$p_{n+1} = k(p_n),$$

so that we can conclude finally that p_n is a primitive recursive function.

It is worth noting that by using our various theorems (and appropriate macro expansions) we could now obtain explicitly a program of \mathcal{S} which actually computes p_n . Of course the program obtained in this way would be extremely inefficient.

Now we want to discuss minimalization when there is no bound. We write

$$\min_y P(x_1, \dots, x_n, y)$$

for the least value of y for which the predicate P is true *if there is one. If there is no value of y for which $P(x_1, \dots, x_n, y)$ is true, then $\min_y P(x_1, \dots, x_n, y)$ is undefined.* (Note carefully the difference with bounded minimalization.) Thus unbounded minimalization of a predicate can easily produce a function which is not total. For example,

$$x - y = \min_z [y + z = x]$$

is undefined for $x < y$. Now, as we shall see later, there are primitive recursive predicates $P(x, y)$ such that $\min_y P(x, y)$ is a total function which is *not* primitive recursive. However, we can prove

Theorem 7.2. If $P(x_1, \dots, x_n, y)$ is a computable predicate and if

$$g(x_1, \dots, x_n) = \min_y P(x_1, \dots, x_n, y),$$

then g is a partially computable function.

Proof. The following program obviously computes g :

```
[A]      IF  $P(X_1, \dots, X_n, Y)$  GOTO E
          Y ← Y + 1
          GOTO A
```

■

Exercises

- Let $h(x)$ be the integer n such that $n \leq \sqrt{2}x < n + 1$. Show that $h(x)$ is primitive recursive.

- Do the same when $h(x)$ is the integer n such that

$$n \leq (1 + \sqrt{2})x < n + 1.$$

- p is called a *larger twin prime* if p and $p - 2$ are both primes. (5, 7, 13, 19 are larger twin primes.) Let $T(0) = 0$, $T(n) =$ the n th larger twin prime. It is widely believed, but has not been proved, that there are infinitely many larger twin primes. Assuming that this is true prove that $T(n)$ is computable.
- Let $u(n)$ be the n th number in order of size which is the sum of two squares. Show that $u(n)$ is primitive recursive.
- Let $R(x, t)$ be a primitive recursive predicate. Let

$$g(x, y) = \max_{t \leq y} R(x, t),$$

i.e., $g(x, y)$ is the largest value of $t \leq y$ for which $R(x, t)$ is true; if there is none, $g(x, y) = 0$. Prove that $g(x, y)$ is primitive recursive.

- Let $\gcd(x, y)$ be the greatest common divisor of x and y . Show that $\gcd(x, y)$ is primitive recursive.
- Let $\text{lcm}(x, y)$ be the least common multiple of x and y . Show that $\text{lcm}(x, y)$ is primitive recursive.

8. Give a computable predicate $P(x_1, \dots, x_n, y)$ such that the function $\min_y P(x_1, \dots, x_n, y)$ is not computable.
- 9.* A function is *elementary* if it can be obtained from the functions s , n , u_j^i , $+$, $\dot{-}$ by a finite sequence of applications of composition, bounded summation, and bounded product. (By application of bounded summation we mean obtaining the function $\sum_{t=0}^y f(t, x_1, \dots, x_n)$ from $f(t, x_1, \dots, x_n)$, and similarly for bounded product.)
- Show that every elementary function is primitive recursive.
 - Show that $x \cdot y$, x^y , and $x!$ are elementary.
 - Show that if $n + 1$ -ary predicates P and Q are elementary, then so are $\sim P$, $P \vee Q$, $P \& Q$, $(\forall t)_{\leq y} P(t, x_1, \dots, x_n)$, $(\exists t)_{\leq y} P(t, x_1, \dots, x_n)$, and $\min_{t \leq y} P(t, x_1, \dots, x_n)$.
 - Show that $\text{Prime}(x)$ is elementary.
 - Let the binary function $\exp_y(x)$ be defined

$$\exp_0(x) = x$$

$$\exp_{y+1}(x) = 2^{\exp_y(x)}.$$

Show that for every elementary function $f(x_1, \dots, x_n)$, there is a constant k such that $f(x_1, \dots, x_n) \leq \exp_k(\max\{x_1, \dots, x_n\})$. [Hint: Show that for every n there is an $m \geq n$ such that $x \cdot \exp_n(x) \leq \exp_m(x)$ for all x .]

- Show that $\exp_y(x)$ is not elementary. Conclude that the class of elementary functions is a proper subset of the class of primitive recursive functions.

8. Pairing Functions and Gödel Numbers

In this section we shall study two convenient coding devices which use primitive recursive functions. The first is for coding pairs of numbers by single numbers, and the second is for coding lists of numbers.

We define the primitive recursive function

$$\langle x, y \rangle = 2^x(2y + 1) \dot{-} 1.$$

Note that $2^x(2y + 1) \neq 0$ so

$$\langle x, y \rangle + 1 = 2^x(2y + 1).$$

If z is any given number, there is a *unique* solution x, y to the equation

$$\langle x, y \rangle = z, \tag{8.1}$$

namely, x is the largest number such that $2^x \mid (z + 1)$, and y is then the solution of the equation

$$2y + 1 = (z + 1)/2^x;$$

this last equation has a (unique) solution because $(z + 1)/2^x$ must be odd. (The twos have been “divided out.”) Equation (8.1) thus defines functions

$$x = l(z), \quad y = r(z).$$

Since Eq. (8.1) implies that $x, y < z + 1$ we have

$$l(z) \leq z, \quad r(z) \leq z.$$

Hence we can write

$$l(z) = \min_{x \leq z} [(\exists y)_{\leq z} (z = \langle x, y \rangle)],$$

$$r(z) = \min_{y \leq z} [(\exists x)_{\leq z} (z = \langle x, y \rangle)],$$

so that $l(z), r(z)$ are primitive recursive functions.

The definition of $l(z), r(z)$ can be expressed by the statement

$$\langle x, y \rangle = z \Leftrightarrow x = l(z) \& y = r(z).$$

We summarize the properties of the functions $\langle x, y \rangle$, $l(z)$, and $r(z)$ in

Theorem 8.1 (Pairing Function Theorem). The functions $\langle x, y \rangle$, $l(z)$, and $r(z)$ have the following properties:

1. they are primitive recursive;
2. $l(\langle x, y \rangle) = x, r(\langle x, y \rangle) = y$;
3. $\langle l(z), r(z) \rangle = z$;
4. $l(z), r(z) \leq z$.

We next obtain primitive recursive functions that encode and decode arbitrary finite sequences of numbers. The method we use, first employed by Gödel, depends on the prime power decomposition of integers.

We define the *Gödel number* of the sequence (a_1, \dots, a_n) to be the number

$$[a_1, \dots, a_n] = \prod_{i=1}^n p_i^{a_i}.$$

Thus, the Gödel number of the sequence $(3, 1, 5, 4, 6)$ is

$$[3, 1, 5, 4, 6] = 2^3 \cdot 3^1 \cdot 5^5 \cdot 7^4 \cdot 11^6.$$

For each fixed n , the function $[a_1, \dots, a_n]$ is clearly primitive recursive.

Gödel numbering satisfies the following uniqueness property:

Theorem 8.2. If $[a_1, \dots, a_n] = [b_1, \dots, b_n]$, then

$$a_i = b_i, \quad i = 1, \dots, n.$$

This result is an immediate consequence of the uniqueness of the factorization of integers into primes, sometimes referred to as the *unique factorization theorem* or the *fundamental theorem of arithmetic*. (For a proof, see any elementary number theory textbook.)

However, note that

$$[a_1, \dots, a_n] = [a_1, \dots, a_n, 0] \quad (8.2)$$

because $p_{n+1}^0 = 1$. This same result obviously holds for any finite number of zeros adjoined to the right end of a sequence. In particular, since

$$1 = 2^0 = 2^0 3^0 = 2^0 3^0 5^0 = \dots,$$

it is natural to regard 1 as the Gödel number of the “empty” sequence of length 0, and it is useful to do so.

If one adjoins 0 to the left end of a sequence, the Gödel number of the new sequence will not be the same as the Gödel number of the original sequence. For example,

$$[2, 3] = 2^2 \cdot 3^3 = 108,$$

and

$$[2, 3, 0] = 2^2 \cdot 3^3 \cdot 5^0 = 108,$$

but

$$[0, 2, 3] = 2^0 \cdot 3^2 \cdot 5^3 = 1125.$$

We will now define a primitive recursive function $(x)_i$ so that if

$$x = [a_1, \dots, a_n],$$

then $(x)_i = a_i$. We set

$$(x)_i = \min_{t \leq x} (\sim p_i^{t+1} \mid x).$$

Note that $(x)_0 = 0$, and $(0)_i = 0$ for all i .

We shall also use the primitive recursive function

$$\text{Lt}(x) = \min_{i \leq x} ((x)_i \neq 0 \ \& \ (\forall j)_{\leq x} (j \leq i \vee (x)_j = 0)).$$

(Lt stands for “length.”) Thus, if $x = 20 = 2^2 \cdot 5^1 = [2, 0, 1]$, then $(x)_3 = 1$, but $(x)_4 = (x)_5 = \dots = (x)_{20} = 0$. So, $\text{Lt}(20) = 3$. Also, $\text{Lt}(0) = \text{Lt}(1) = 0$.

If $x > 1$, and $\text{Lt}(x) = n$, then p_n divides x but no prime greater than p_n divides x . Note that $\text{Lt}([a_1, \dots, a_n]) = n$ if and only if $a_n \neq 0$.

We summarize the key properties of these primitive recursive functions.

Theorem 8.3 (Sequence Number Theorem).

- a. $([a_1, \dots, a_n])_i = \begin{cases} a_i & \text{if } 1 \leq i \leq n \\ 0 & \text{otherwise.} \end{cases}$
- b. $[(x)_1, \dots, (x)_n] = x \quad \text{if } n \geq \text{Lt}(x).$

Our main application of these coding techniques is given in the next chapter. The following exercises indicate that they can also be used to show that PRC classes are closed under various interesting and useful forms of recursion.

Exercises

1. Let $f(x_1, \dots, x_n)$ be a function of n variables, and let $f'(x)$ be a unary function defined so that $f'([x_1, \dots, x_n]) = f(x_1, \dots, x_n)$ for all x_1, \dots, x_n . Show that f' is partially computable if and only if f is partially computable.
2. Define $\text{Sort}([x_1, \dots, x_n]) = [y_1, \dots, y_n]$, where y_1, \dots, y_n is a permutation of x_1, \dots, x_n such that $y_1 \leq y_2 \leq \dots \leq y_n$. Show that $\text{Sort}(x)$ is primitive recursive.
3. Let $F(0) = 0$, $F(1) = 1$, $F(n + 2) = F(n + 1) + F(n)$. [$F(n)$ is the n th so-called Fibonacci number.] Prove that $F(n)$ is primitive recursive.
4. (Simultaneous Recursion) Let

$$h_1(x, 0) = f_1(x),$$

$$h_2(x, 0) = f_2(x),$$

$$h_1(x, t + 1) = g_1(x, h_1(x, t), h_2(x, t)),$$

$$h_2(x, t + 1) = g_2(x, h_1(x, t), h_2(x, t)).$$

Prove that if f_1, f_2, g_1, g_2 all belong to some PRC class \mathcal{C} , then h_1, h_2 do also.

5.* (Course-of-Values Recursion)

(a) For $f(n)$ any function, we write

$$\tilde{f}(0) = 1, \tilde{f}(n) = [f(0), f(1), \dots, f(n - 1)] \text{ if } n \neq 0.$$

Let

$$f(n) = g(n, \tilde{f}(n))$$

for all n . Show that if g is primitive recursive so is f .

(b) Let

$$\begin{aligned} f(0) &= 1, & f(1) &= 4, & f(2) &= 6, \\ f(x+3) &= f(x) + f(x+1)^2 + f(x+2)^3. \end{aligned}$$

Show that $f(x)$ is primitive recursive.

(c) Let

$$\begin{aligned} h(0) &= 3 \\ h(x+1) &= \sum_{t=0}^x h(t). \end{aligned}$$

Show that h is primitive recursive.

6.* (Unnested Double Recursion) Let

$$\begin{aligned} f(0, y) &= g_1(y) \\ f(x+1, 0) &= g_2(x) \\ f(x+1, y+1) &= h(x, y, f(x, y+1), f(x+1, y)). \end{aligned}$$

Show that if g_1 , g_2 , and h all belong to some PRC class \mathcal{C} , then f also belongs to \mathcal{C} .

4

A Universal Program

1. Coding Programs by Numbers

We are going to associate with each program \mathcal{P} of the language \mathcal{S} a number, which we write $\#(\mathcal{P})$, in such a way that the program can be retrieved from its number. To begin with we arrange the variables in order as follows:

$$Y \ X_1 \ Z_1 \ X_2 \ Z_2 \ X_3 \ Z_3 \dots$$

Next we do the same for the labels:

$$A_1 \ B_1 \ C_1 \ D_1 \ E_1 \ A_2 \ B_2 \ C_2 \ D_2 \ E_2 \ A_3 \dots$$

We write $\#(V)$, $\#(L)$ for the position of a given variable or label in the appropriate ordering. Thus $\#(X_2) = 4$, $\#(Z_1) = \#(Z) = 3$, $\#(E) = 5$, $\#(B_2) = 7$.

Now let I be an instruction (labeled or unlabeled) of the language \mathcal{S} . Then we write

$$\#(I) = \langle a, \langle b, c \rangle \rangle$$

where

1. if I is unlabeled, then $a = 0$; if I is labeled L , then $a = \#(L)$;
2. if the variable V is mentioned in I , then $c = \#(V) - 1$;

3. if the statement in I is

$$V \leftarrow V \quad \text{or} \quad V \leftarrow V + 1 \quad \text{or} \quad V \leftarrow V - 1,$$

then $b = 0$ or 1 or 2 , respectively;

4. if the statement in I is

$$\text{IF } V \neq 0 \text{ GOTO } L'$$

then $b = \#(L') + 2$.

Some examples:

The number of the unlabeled instruction $X \leftarrow X + 1$ is

$$\langle 0, \langle 1, 1 \rangle \rangle = \langle 0, 5 \rangle = 10,$$

whereas the number of the instruction

$$[A] \quad X \leftarrow X + 1$$

is

$$\langle 1, \langle 1, 1 \rangle \rangle = \langle 1, 5 \rangle = 21.$$

Note that for any given number q there is a unique instruction I with $\#(I) = q$. We first calculate $l(q)$. If $l(q) = 0$, I is unlabeled; otherwise I has the $l(q)$ th label in our list. To find the variable mentioned in I , we compute $i = r(r(q)) + 1$ and locate the i th variable V in our list. Then, the statement in I will be

$$\begin{aligned} V &\leftarrow V && \text{if } l(r(q)) = 0, \\ V &\leftarrow V + 1 && \text{if } l(r(q)) = 1, \\ V &\leftarrow V - 1 && \text{if } l(r(q)) = 2, \\ \text{IF } V &\neq 0 \text{ GOTO } L && \text{if } j = l(r(q)) - 2 > 0 \end{aligned}$$

and L is the j th label in our list.

Finally, let a program \mathcal{P} consist of the instructions I_1, I_2, \dots, I_k . Then we set

$$\#(\mathcal{P}) = [\#(I_1), \#(I_2), \dots, \#(I_k)] - 1. \quad (1.1)$$

Since Gödel numbers tend to be very large, the number of even rather simple programs usually will be quite enormous. We content ourselves with a simple example:

$$\begin{aligned} [A] \quad X &\leftarrow X + 1 \\ \text{IF } X &\neq 0 \text{ GOTO } A \end{aligned}$$

The reader will recognize this as the example given in Chapter 2 of a program that computes the nowhere defined function. Calling these instructions I_1 and I_2 , respectively, we have seen that $\#(I_1) = 21$. Since I_2 is unlabeled,

$$\#(I_2) = \langle 0, \langle 3, 1 \rangle \rangle = \langle 0, 23 \rangle = 46.$$

Thus, finally, the number of this short program is

$$2^{21} \cdot 3^{46} - 1.$$

Note that the number of the unlabeled instruction $Y \leftarrow Y$ is

$$\langle 0, \langle 0, 0 \rangle \rangle = \langle 0, 0 \rangle = 0.$$

Thus, by the ambiguity in Gödel numbers [recall Eq. (8.2), Chapter 3], the number of a program will be unchanged if an unlabeled $Y \leftarrow Y$ is tacked onto its end. Of course this is a harmless ambiguity; the longer program computes exactly what the shorter one does. However, we remove even this ambiguity by adding to our official definition of program of \mathcal{S} the harmless stipulation that *the final instruction in a program is not permitted to be the unlabeled statement $Y \leftarrow Y$* .

With this last stipulation each number determines a unique program. As an example, let us determine the program whose number is 199. We have

$$199 + 1 = 200 = 2^3 \cdot 3^0 \cdot 5^2 = [3, 0, 2].$$

Thus, if $\#(\mathcal{P}) = 199$, \mathcal{P} consists of 3 instructions, the second of which is the unlabeled statement $Y \leftarrow Y$. We have

$$3 = \langle 2, 0 \rangle = \langle 2, \langle 0, 0 \rangle \rangle$$

and

$$2 = \langle 0, 1 \rangle = \langle 0, \langle 1, 0 \rangle \rangle.$$

Thus, the program is

$$\begin{aligned}[B] & Y \leftarrow Y \\ & Y \leftarrow Y \\ & Y \leftarrow Y + 1\end{aligned}$$

a not very interesting program that computes the function $y = 1$.

Note also that the empty program has the number $1 - 1 = 0$.

Exercises

1. Compute $\#(\mathcal{P})$ for \mathcal{P} the programs of Exercises 4.1, 4.2, Chapter 2.
2. Find \mathcal{P} such that $\#(\mathcal{P}) = 575$.

2. The Halting Problem

In this section we want to discuss a predicate $\text{HALT}(x, y)$, which we now define. For given y , let \mathcal{P} be the program such that $\#(\mathcal{P}) = y$. Then $\text{HALT}(x, y)$ is true if $\psi_{\mathcal{P}}^{(1)}(x)$ is defined and false if $\psi_{\mathcal{P}}^{(1)}(x)$ is undefined. To put it succinctly:

$$\text{HALT}(x, y) \Leftrightarrow \text{program number } y \text{ eventually halts on input } x.$$

We now prove the remarkable:

Theorem 2.1. $\text{HALT}(x, y)$ is not a computable predicate.

Proof. Suppose that $\text{HALT}(x, y)$ were computable. Then we could construct the program \mathcal{P} :

[A] IF $\text{HALT}(X, X)$ GOTO A

(Of course \mathcal{P} is to be the macro expansion of this program.) It is quite clear that \mathcal{P} has been constructed so that

$$\psi_{\mathcal{P}}^{(1)}(x) = \begin{cases} \text{undefined} & \text{if } \text{HALT}(x, x) \\ 0 & \text{if } \sim \text{HALT}(x, x). \end{cases}$$

Let $\#(\mathcal{P}) = y_0$. Then using the definition of the HALT predicate,

$$\text{HALT}(x, y_0) \Leftrightarrow \sim \text{HALT}(x, x).$$

Since this equivalence is true for all x , we can set $x = y_0$:

$$\text{HALT}(y_0, y_0) \Leftrightarrow \sim \text{HALT}(y_0, y_0).$$

But this is a contradiction. ■

To begin with, this theorem provides us with an example of a function that is not computable by any program in the language \mathcal{S} . But we would like to go further; we would like to conclude the following:

There is no algorithm that, given a program of \mathcal{S} and an input to that program, can determine whether or not the given program will eventually halt on the given input.

In this form the result is called the *unsolvability of the halting problem*. We reason as follows: if there were such an algorithm, we could use it to check the truth or falsity of $\text{HALT}(x, y)$ for given x, y by first obtaining program \mathcal{Q} with $\#(\mathcal{Q}) = y$ and then checking whether \mathcal{Q} eventually halts on input x . But we have reason to believe that *any algorithm for computing on*

numbers can be carried out by a program of \mathcal{S} . Hence this would contradict the fact that $\text{HALT}(x, y)$ is not computable.

The last italicized assertion is a form of what has come to be called *Church's thesis*. We have already accumulated some evidence for it, and we will see more later. But, since the word *algorithm* has no general definition separated from a particular language, Church's thesis cannot be proved as a mathematical theorem.

In fact, we will use Church's thesis freely in asserting the nonexistence of algorithms whenever we have shown that some problem cannot be solved by a program of \mathcal{S} .

In the light of Church's thesis, Theorem 2.1 tells us that there really is no algorithm for testing a given program and input to determine whether it will ever halt. Anyone who finds it surprising that no algorithm exists for such a "simple" problem should be made to realize that it is easy to construct relatively short programs (of \mathcal{S}) such that nobody is in a position to tell whether they will ever halt. For example, consider the assertion from number theory that every even number ≥ 4 is the sum of two prime numbers. This assertion, known as *Goldbach's conjecture*, is clearly true for small even numbers: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, etc. It is easy to write a program \mathcal{P} of \mathcal{S} that will search for a counterexample to Goldbach's conjecture, that is, an even number $n \geq 4$ that is not the sum of two primes. Note that the test that a given even number n is a counterexample only requires checking the primitive recursive predicate

$$\sim (\exists x)_{\leq n} (\exists y)_{\leq n} [\text{Prime}(x) \& \text{Prime}(y) \& x + y = n].$$

The statement that \mathcal{P} never halts is equivalent to Goldbach's conjecture. Since the conjecture is still open after 250 years, nobody knows whether this program \mathcal{P} will eventually halt.

Exercises

1. Show that $\text{HALT}(x, x)$ is not computable.
2. Let $\overline{\text{HALT}}(x, y)$ be defined

$$\overline{\text{HALT}}(x, y) \Leftrightarrow \text{program number } y \text{ never halts on input } x.$$

Show that $\overline{\text{HALT}}(x, y)$ is not computable.

3. Let $\text{HALT}^1(x)$ be defined $\text{HALT}^1(x) \Leftrightarrow \text{HALT}(l(x), r(x))$. Show that $\text{HALT}^1(x)$ is not computable.

4. Prove or disprove: If $f(x_1, \dots, x_n)$ is a total function such that for some constant k , $f(x_1, \dots, x_n) \leq k$ for all x_1, \dots, x_n , then f is computable.
5. Suppose we claim that \mathcal{P} is a program that computes $\text{HALT}(x, x)$. Give a counterexample that shows the claim to be false. That is, give an input x for which \mathcal{P} gives the wrong answer.
6. Let

$$f(x) = \begin{cases} x & \text{if Goldbach's conjecture is true} \\ 0 & \text{otherwise.} \end{cases}$$

Show that $f(x)$ is primitive recursive.

3. Universality

The negative character of the results in the previous section might lead one to believe that it is not possible to compute in a useful way with numbers of programs. But, as we shall soon see, this belief is not justified.

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \dots, x_n, y) = \psi_{\mathcal{P}}^{(n)}(x_1, \dots, x_n), \quad \text{where } \#(\mathcal{P}) = y.$$

One of the key tools in computability theory is

Theorem 3.1 (Universality Theorem). For each $n > 0$, the function $\Phi^{(n)}(x_1, \dots, x_n, y)$ is partially computable.

We shall prove this theorem by showing how to construct, for each $n > 0$, a program \mathcal{U}_n which computes $\Phi^{(n)}$. That is, we shall have for each $n > 0$,

$$\psi_{\mathcal{U}_n}^{(n+1)}(x_1, \dots, x_n, x_{n+1}) = \Phi^{(n)}(x_1, \dots, x_n, x_{n+1}).$$

The programs \mathcal{U}_n are called *universal*. For example, \mathcal{U}_1 can be used to compute *any* partially computable function of one variable, namely, if $f(x)$ is computed by a program \mathcal{P} and $y = \#(\mathcal{P})$, then $f(x) = \Phi^{(1)}(x, y) = \psi_{\mathcal{U}_1}^{(2)}(x, y)$. The program \mathcal{U}_n will work very much like an interpreter. It must keep track of the current snapshot in a computation and by “decoding” the number of the program being interpreted, decide what to do next and then do it.

In writing the programs \mathcal{U}_n we shall freely use macros corresponding to functions that we know to be primitive recursive using the methods of Chapter 3. We shall also freely ignore the rules concerning which letters may be used to represent variables or labels of \mathcal{S} .

In considering the state of a computation we can assume that all variables which are not given values have the value 0. With this understanding, we can code the state in which the i th variable in our list has the value a_i and all variables after the m th have the value 0, by the Gödel number $[a_1, \dots, a_m]$. For example, the state

$$Y = 0, \quad X_1 = 2, \quad X_2 = 1$$

is coded by the number

$$[0, 2, 0, 1] = 3^2 \cdot 7 = 63.$$

Notice in particular that the input variables are those whose position in our list is an *even* number.

Now in the universal programs, we shall allocate storage as follows:

K will be the number such that the K th instruction is about to be executed;

S will store the current state coded in the manner just explained.

We proceed to give the program \mathcal{U}_n for computing

$$Y = \Phi^{(n)}(X_1, \dots, X_n, X_{n+1}).$$

We begin by exhibiting \mathcal{U}_n in sections, explaining what each part does. Finally, we shall put the pieces together. We begin:

$$\begin{aligned} Z &\leftarrow X_{n+1} + 1 \\ S &\leftarrow \prod_{i=1}^n (p_{2i})^{X_i} \\ K &\leftarrow 1 \end{aligned}$$

If $X_{n+1} = \#(\mathcal{P})$, where \mathcal{P} consists of the instructions I_1, \dots, I_m , then Z gets the value $[\#(I_1), \dots, \#(I_m)]$ [see Eq. (1.1)]. S is initialized as $[0, X_1, 0, X_2, \dots, 0, X_n]$, which gives the first n input variables their appropriate values and gives all other variables the value 0. K , the instruction counter, is given the initial value 1 (so that the computation can begin with the first instruction). Next,

$$[C] \text{ IF } K = \text{Lt}(Z) + 1 \vee K = 0 \text{ GOTO } F$$

If the computation has ended, GOTO F , where the proper value will be output. (The significance of $K = 0$ will be explained later.) Otherwise, the current instruction must be decoded and executed:

$$\begin{aligned} U &\leftarrow r((Z)_K) \\ P &\leftarrow p_{r(U)+1} \end{aligned}$$

$(Z)_K = \langle a, \langle b, c \rangle \rangle$ is the number of the K th instruction. Thus, $U = \langle b, c \rangle$ is the code for the *statement* about to be executed. The variable mentioned in the K th instruction is the $(c + 1)$ th, i.e., the $(r(U) + 1)$ th, in our list. Thus, its current value is stored as the exponent to which P divides S :

```

IF  $l(U) = 0$  GOTO  $N$ 
IF  $l(U) = 1$  GOTO  $A$ 
IF  $\sim (P | S)$  GOTO  $N$ 
IF  $l(U) = 2$  GOTO  $M$ 
```

If $l(U) = 0$, the instruction is a dummy $V \leftarrow V$ and the computation need do nothing to S . If $l(U) = 1$, the instruction is of the form $V \leftarrow V + 1$, so that 1 has to be added to the exponent on P in the prime power factorization of S . The computation executes a GOTO A (for *Add*). If $l(U) \neq 0, 1$, then the current instruction is either of the form $V \leftarrow V - 1$ or IF $V \neq 0$ GOTO L . In either case, if P is not a divisor of S , i.e., if the current value of V is 0, the computation need do *nothing* to S . If $P | S$ and $l(U) = 2$, then the computation executes a GOTO M (for *Minus*), so that 1 can be subtracted from the exponent to which P divides S . To continue,

```

 $K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$ 
GOTO  $C$ 
```

If $l(U) > 2$ and $P | S$, the current instruction is of the form IF $V \neq 0$ GOTO L where V has a nonzero value and L is the label whose position in our list is $l(U) - 2$. Accordingly the next instruction should be the first with this label. That is, K should get as its value the least i for which $l((Z)_i) = l(U) - 2$. If there is no instruction with the appropriate label, K gets the value 0, which will lead to termination the next time through the main loop. In either case the GOTO C causes a “jump” to the beginning of the loop for the next instruction (if any) to be processed. Continuing,

```

 $[M] \quad S \leftarrow \lfloor S/P \rfloor$ 
GOTO  $N$ 
 $[A] \quad S \leftarrow S \cdot P$ 
 $[N] \quad K \leftarrow K + 1$ 
GOTO  $C$ 
```

1 is subtracted or added to the value of the variable mentioned in the current instruction by dividing or multiplying S by P , respectively. The

```

 $Z \leftarrow X_{n+1} + 1$ 
 $S \leftarrow \prod_{i=1}^n (p_{2i})^{X_i}$ 
 $K \leftarrow 1$ 
[C] IF  $K = \text{Lt}(Z) + 1 \vee K = 0$  GOTO F
 $U \leftarrow r((Z)_K)$ 
 $P \leftarrow p_{r(U)+1}$ 
IF  $l(U) = 0$  GOTO N
IF  $l(U) = 1$  GOTO A
IF  $\sim(P \mid S)$  GOTO N
IF  $l(U) = 2$  GOTO M
 $K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$ 
GOTO C
[M]  $S \leftarrow \lfloor S/P \rfloor$ 
GOTO N
[A]  $S \leftarrow S \cdot P$ 
[N]  $K \leftarrow K + 1$ 
GOTO C
[F]  $Y \leftarrow (S)_1$ 

```

Figure 3.1. Program \mathcal{U}_n , which computes $Y = \Phi^{(n)}(X_1, \dots, X_n, X_{n+1})$.

instruction counter is increased by 1 and the computation returns to process the next instruction. To conclude the program,

[F] $Y \leftarrow (S)_1$

On termination, the value of Y for the program being simulated is stored as the exponent on $p_1 (= 2)$ in S . We have now completed our description of \mathcal{U}_n and we put the pieces together in Fig. 3.1.

For each $n > 0$, the sequence

$$\Phi^{(n)}(x_1, \dots, x_n, 0), \Phi^{(n)}(x_1, \dots, x_n, 1), \dots$$

enumerates all partially computable functions of n variables. When we want to emphasize this aspect of the situation we write

$$\Phi_y^{(n)}(x_1, \dots, x_n) = \Phi^{(n)}(x_1, \dots, x_n, y).$$

It is often convenient to omit the superscript when $n = 1$, writing

$$\Phi_y(x) = \Phi(x, y) = \Phi^{(1)}(x, y).$$

A simple modification of the programs \mathcal{U}_n would enable us to prove that the predicates

$$\begin{aligned} \text{STP}^{(n)}(x_1, \dots, x_n, y, t) &\Leftrightarrow \text{Program number } y \text{ halts after } t \text{ or fewer} \\ &\quad \text{steps on inputs } x_1, \dots, x_n \\ &\Leftrightarrow \text{There is a computation of program } y \text{ of} \\ &\quad \text{length } \leq t + 1, \text{ beginning with inputs} \\ &\quad x_1, \dots, x_n \end{aligned}$$

are computable. We simply need to add a counter to determine when we have simulated t steps. However, we can prove a stronger result.

Theorem 3.2 (Step-Counter Theorem). For each $n > 0$, the predicate $\text{STP}^{(n)}(x_1, \dots, x_n, y, t)$ is primitive recursive.

Proof. The idea is to provide numeric versions of the notions of snapshot and successor snapshot and to show that the necessary functions are primitive recursive. We use the same representation of program states that we used in defining the universal programs, and if z represents state σ , then $\langle i, z \rangle$ represents the snapshot (i, σ) .

We begin with some functions for extracting the components of the i th instruction of program number y :

$$\text{LABEL}(i, y) = l((y + 1)_i)$$

$$\text{VAR}(i, y) = r(r((y + 1)_i)) + 1$$

$$\text{INSTR}(i, y) = l(r((y + 1)_i))$$

$$\text{LABEL}'(i, y) = l(r((y + 1)_i)) \div 2$$

Next we define some predicates that indicate, for program y and the snapshot represented by x , which kind of action is to be performed next.

$$\text{SKIP}(x, y) \Leftrightarrow [\text{INSTR}(l(x), y) = 0 \ \& \ l(x) \leq \text{Lt}(y + 1)]$$

$$\vee [\text{INSTR}(l(x), y) \geq 2 \ \& \ \neg(p_{\text{VAR}(l(x), y)} \mid r(x))]$$

$$\text{INCR}(x, y) \Leftrightarrow \text{INSTR}(l(x), y) = 1$$

$$\text{DECR}(x, y) \Leftrightarrow \text{INSTR}(l(x), y) = 2 \ \& \ p_{\text{VAR}(l(x), y)} \mid r(x)$$

$$\text{BRANCH}(x, y) \Leftrightarrow \text{INSTR}(l(x), y) > 2 \ \& \ p_{\text{VAR}(l(x), y)} \mid r(x)$$

$$\quad \& (\exists i)_{\leq \text{Lt}(y + 1)} \text{LABEL}(i, y) = \text{LABEL}'(l(x), y)$$

Now we can define $\text{SUCC}(x, y)$, which, for program number y , gives the representative of the successor to the snapshot represented by x .

$$\text{SUCC}(x, y) = \begin{cases} \langle l(x) + 1, r(x) \rangle & \text{if } \text{SKIP}(x, y) \\ \langle l(x) + 1, r(x) \cdot p_{\text{VAR}(l(x), y)} \rangle & \text{if } \text{INCR}(x, y) \\ \langle l(x) + 1, \lfloor r(x)/p_{\text{VAR}(l(x), y)} \rfloor \rangle & \text{if } \text{DECR}(x, y) \\ \langle \min_{i \leq \text{Lt}(y+1)} [\text{LABEL}(i, y) = \text{LABEL}'(l(x), y)], r(x) \rangle & \text{if } \text{BRANCH}(x, y) \\ \langle \text{Lt}(y+1) + 1, r(x) \rangle & \text{otherwise.} \end{cases}$$

We also need

$$\text{INIT}^{(n)}(x_1, \dots, x_n) = \langle 1, \prod_{i=1}^n (p_{2i})^{x_i} \rangle,$$

which gives the representation of the initial snapshot for inputs x_1, \dots, x_n , and

$$\text{TERM}(x, y) \Leftrightarrow l(x) > \text{Lt}(y+1),$$

which tests whether x represents a terminal snapshot for program y .

Putting these together we can define a primitive recursive function that gives the numbers of the successive snapshots produced by a given program.

$$\text{SNAP}^{(n)}(x_1, \dots, x_n, y, 0) = \text{INIT}^{(n)}(x_1, \dots, x_n)$$

$$\text{SNAP}^{(n)}(x_1, \dots, x_n, y, i+1) = \text{SUCC}(\text{SNAP}^{(n)}(x_1, \dots, x_n, y, i), y)$$

Thus,

$$\text{STP}^{(n)}(x_1, \dots, x_n, y, t) \Leftrightarrow \text{TERM}(\text{SNAP}^{(n)}(x_1, \dots, x_n, y, t), y),$$

and it is clear that $\text{STP}^{(n)}(x_1, \dots, x_n, y, t)$ is primitive recursive. ■

By using the technique of the above proof, we can obtain the following important result.

Theorem 3.3 (Normal Form Theorem). Let $f(x_1, \dots, x_n)$ be a partially computable function. Then there is a primitive recursive predicate $R(x_1, \dots, x_n, y)$ such that

$$f(x_1, \dots, x_n) = l \left(\min_z R(x_1, \dots, x_n, z) \right).$$

Proof. Let y_0 be the number of a program that computes $f(x_1, \dots, x_n)$. We shall prove the following equation, which clearly implies the desired result:

$$f(x_1, \dots, x_n) = l\left(\min_z R(x_1, \dots, x_n, z)\right) \quad (3.1)$$

where $R(x_1, \dots, x_n, z)$ is the predicate

$$\begin{aligned} & \text{STP}^{(n)}(x_1, \dots, x_n, y_0, r(z)) \\ & \quad \& (r(\text{SNAP}^{(n)}(x_1, \dots, x_n, y_0, r(z))))_1 \\ & = l(z). \end{aligned}$$

First consider the case when the righthand side of this equation is defined. Then, in particular, there exists a number z such that

$$\begin{aligned} & \text{STP}^{(n)}(x_1, \dots, x_n, y_0, r(z)) \\ & \quad \text{and } (r(\text{SNAP}^{(n)}(x_1, \dots, x_n, y_0, r(z))))_1 \\ & = l(z). \end{aligned}$$

For any such z , the computation by the program with number y_0 has reached a terminal snapshot in $r(z)$ or fewer steps and $l(z)$ is the value held in the output variable Y , i.e., $l(z) = f(x_1, \dots, x_n)$.

If, on the other hand, the right side is undefined, it must be the case that $\text{STP}^{(n)}(x_1, \dots, x_n, y_0, t)$ is false for all values of t , i.e., $f(x_1, \dots, x_n) \uparrow$. ■

The normal form theorem leads to another characterization of the class of partially computable functions.

Theorem 3.4. A function is partially computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and minimalization.

Proof. That every function which can be so obtained is partially computable is an immediate consequence of Theorems 1.1, 2.1, 2.2, 3.1, and 7.2 in Chapter 3. Note that a partially computable predicate is necessarily computable, so Theorem 7.2 covers all applications of minimalization to a predicate obtained as described in the theorem.

Conversely, we can use the normal form theorem to write any given partially computable function in the form

$$l\left(\min_y R(x_1, \dots, x_n, y)\right),$$

where R is a primitive recursive predicate and so is obtained from the initial functions by a finite number of applications of composition and

recursion. Finally, our given function is obtained from R by one use of minimalization and then by composition with the primitive recursive function l . ■

When $\min_y R(x_1, \dots, x_n, y)$ is a total function [that is, when for each x_1, \dots, x_n there is at least one y for which $R(x_1, \dots, x_n, y)$ is true], we say that we are applying the operation of *proper minimization* to R . Now, if

$$l\left(\min_y R(x_1, \dots, x_n, y)\right)$$

is total, then $\min_y R(x_1, \dots, x_n, y)$ must be total. Hence we have

Theorem 3.5. A function is computable if and only if it can be obtained from the initial functions by a finite number of applications of composition, recursion, and *proper* minimization.

Exercises

1. Show that for each u , there are infinitely many different numbers v such that for all x , $\Phi_u(x) = \Phi_v(x)$.
2. (a) Let

$$H_1(x) = \begin{cases} 1 & \text{if } \Phi(x, x) \downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

Show that $H_1(x)$ is partially computable.

- (b) Let $A = \{a_1, \dots, a_n\}$ be a finite set such that $\Phi(a_i, a_i) \uparrow$ for $1 \leq i \leq n$, and let

$$H_2(x) = \begin{cases} 1 & \text{if } \Phi(x, x) \downarrow \\ 0 & \text{if } x \in A \\ \uparrow & \text{otherwise.} \end{cases}$$

Show that $H_2(x)$ is partially computable.

- (c) Give an infinite set B such that $\Phi(b, b) \uparrow$ for all $b \in B$ and such that

$$H_3(x) = \begin{cases} 1 & \text{if } \Phi(x, x) \downarrow \\ 0 & \text{if } x \in B \\ \uparrow & \text{otherwise} \end{cases}$$

is partially computable.

- (d) Give an infinite set C such that $\Phi(c, c) \uparrow$ for all $c \in C$ and such that

$$H_4(x) = \begin{cases} 1 & \text{if } \Phi(x, x) \downarrow \\ 0 & \text{if } x \in C \\ \uparrow & \text{otherwise} \end{cases}$$

is not partially computable.

3. Give a program \mathcal{P} such that $H_{\mathcal{P}}(x_1, x_2)$, defined

$$H_{\mathcal{P}}(x_1, x_2) \Leftrightarrow \text{program } \mathcal{P} \text{ eventually halts on inputs } x_1, x_2$$

is not computable.

4. Let $f(x_1, \dots, x_n)$ be computed by program \mathcal{P} , and suppose that for some primitive recursive function $g(x_1, \dots, x_n)$,

$$\text{STP}^{(n)}(x_1, \dots, x_n, \#(\mathcal{P}), g(x_1, \dots, x_n))$$

is true for all x_1, \dots, x_n . Show that $f(x_1, \dots, x_n)$ is primitive recursive.

- 5.* Give a primitive recursive function $\text{counter}(x)$ such that if Φ_n is a computable predicate, then

$$\Phi_n(\text{counter}(n)) \Leftrightarrow \sim \text{HALT}(\text{counter}(n), \text{counter}(n)).$$

That is, $\text{counter}(n)$ is a counterexample to the possibility that Φ_n computes $\text{HALT}(x, x)$. [Compare this exercise with Exercise 2.5.]

- 6.* Give an upper bound on the length of the shortest \mathcal{S} program that computes the function $\Phi_y(x)$.

4. Recursively Enumerable Sets

The close relation between predicates and sets, as described in Chapter 1, lets us use the language of sets in talking about solvable and unsolvable problems. For example, the predicate $\text{HALT}(x, y)$ is the characteristic function of the set $\{(x, y) \in N^2 \mid \text{HALT}(x, y)\}$. To say that a set B , where $B \subseteq N^m$, belongs to some class of *functions* means that the characteristic function $P(x_1, \dots, x_m)$ of B belongs to the class in question. Thus, in particular, to say that the set B is computable or recursive is just to say that $P(x_1, \dots, x_m)$ is a computable function. Likewise, B is a primitive recursive set if $P(x_1, \dots, x_m)$ is a primitive recursive predicate.

We have, for example,

Theorem 4.1. Let the sets B, C belong to some PRC class \mathcal{C} . Then so do the sets $B \cup C, B \cap C, \bar{B}$.

Proof. This is an immediate consequence of Theorem 5.1, Chapter 3. ■

As long as the Gödel numbering functions $[x_1, \dots, x_n]$ and $(x)_i$ are available, we can restrict our attention to subsets of N . We have, for example,

Theorem 4.2. Let \mathcal{C} be a PRC class, and let B be a subset of N^m , $m \geq 1$. Then B belongs to \mathcal{C} if and only if

$$B' = \{[x_1, \dots, x_m] \in N \mid (x_1, \dots, x_m) \in B\}$$

belongs to \mathcal{C} .

Proof. If $P_B(x_1, \dots, x_m)$ is the characteristic function of B , then

$$P_{B'}(x) \Leftrightarrow P_B((x)_1, \dots, (x)_m) \text{ & } \text{Lt}(x) = m$$

is the characteristic function of B' , and $P_{B'}$ clearly belongs to \mathcal{C} if P_B belongs to \mathcal{C} . On the other hand, if $P_{B'}(x)$ is the characteristic function of B' , then

$$P_B(x_1, \dots, x_m) \Leftrightarrow P_{B'}([x_1, \dots, x_m])$$

is the characteristic function of B , and P_B clearly belongs to \mathcal{C} if $P_{B'}$ belongs to \mathcal{C} . ■

It immediately follows, for example, that $\{[x, y] \in N \mid \text{HALT}(x, y)\}$ is not a computable set.

Definition. The set $B \subseteq N$ is called *recursively enumerable* if there is a partially computable function $g(x)$ such that

$$B = \{x \in N \mid g(x) \downarrow\}. \quad (4.1)$$

The term *recursively enumerable* is usually abbreviated *r.e.* A set is recursively enumerable just when it is the domain of a partially computable function. If \mathcal{P} is a program that computes the function g in (4.1), then B is simply the set of all inputs to \mathcal{P} for which \mathcal{P} eventually halts. If we think of \mathcal{P} as providing an algorithm for testing for membership in B , we see that for numbers that do belong to B , the algorithm will provide a

“yes” answer; but for numbers that do not, the algorithm will never terminate. If we invoke Church’s thesis, r.e. sets B may be thought of intuitively as sets for which there exist algorithms related to B as in the previous sentence, but without stipulating that the algorithms be expressed by programs of the language \mathcal{S} . Such algorithms, sometimes called *semi-decision procedures*, provide a kind of “approximation” to solving the problem of testing membership in B .

We have

Theorem 4.3. If B is a recursive set, then B is r.e.

Proof. Consider the program \mathcal{P} :

$$[A] \text{ IF } \sim(X \in B) \text{ GOTO } A$$

Since B is recursive, the predicate $x \in B$ is computable and \mathcal{P} can be expanded to a program of \mathcal{S} . Let \mathcal{P} compute the function $h(x)$. Then, clearly,

$$B = \{x \in N \mid h(x) \downarrow\}. \quad \blacksquare$$

If B and \bar{B} are both r.e., we have a pair of algorithms that will terminate in case a given input is or is not in B , respectively. We can think of combining these two algorithms to obtain a single algorithm that will always terminate and that will tell us whether a given input belongs to B . This combined algorithm might work by “running” the two separate algorithms for longer and longer times until one of them terminates. This method of combining algorithms is called *dovetailing*, and the step-counter theorem enables us to use it in a rigorous manner.

Theorem 4.4. The set B is recursive if and only if B and \bar{B} are both r.e.

Proof. If B is recursive, then by Theorem 4.1 so is \bar{B} , and hence by Theorem 4.3, they are both r.e.

Conversely, if B and \bar{B} are both r.e., we may write

$$B = \{x \in N \mid g(x) \downarrow\},$$

$$\bar{B} = \{x \in N \mid h(x) \downarrow\},$$

where g and h are both partially computable. Let g be computed by program \mathcal{P} and h be computed by program \mathcal{Q} , and let $p = \#(\mathcal{P})$, $q = \#(\mathcal{Q})$. Then the program that follows computes B . (That is, the program computes the characteristic function of B .)

```

[A]      IF STP(1)(X, p, T) GOTO C
          IF STP(1)(X, q, T) GOTO E
          T ← T + 1
          GOTO A
[C]      Y ← 1

```

■

Theorem 4.5. If B and C are r.e. sets so are $B \cup C$ and $B \cap C$.

Proof. Let

$$\begin{aligned} B &= \{x \in N \mid g(x) \downarrow\}, \\ C &= \{x \in N \mid h(x) \downarrow\}, \end{aligned}$$

where g and h are both partially computable. Let $f(x)$ be the function computed by the program

$$\begin{aligned} Y &\leftarrow g(X) \\ Y &\leftarrow h(X) \end{aligned}$$

Then $f(x)$ is defined if and only if $g(x)$ and $h(x)$ are both defined. Hence

$$B \cap C = \{x \in N \mid f(x) \downarrow\},$$

so that $B \cap C$ is also r.e.

To obtain the result for $B \cup C$ we must use dovetailing again. Let g and h be computed by programs \mathcal{P} and \mathcal{Q} , respectively, and let $\#(\mathcal{P}) = p$, $\#(\mathcal{Q}) = q$. Let $k(x)$ be the function computed by the program

```

[A]      IF STP(1)(X, p, T) GOTO E
          IF STP(1)(X, q, T) GOTO E
          T ← T + 1
          GOTO A'

```

Then $k(x)$ is defined just in case either $g(x)$ or $h(x)$ is defined. That is,

$$B \cup C = \{x \in N \mid k(x) \downarrow\}. \quad ■$$

Definition. We write

$$W_n = \{x \in N \mid \Phi(x, n) \downarrow\}.$$

Then we have

Theorem 4.6 (Enumeration Theorem). A set B is r.e. if and only if there is an n for which $B = W_n$.

Proof. This is an immediate consequence of the definition of $\Phi(x, n)$. ■

The theorem gets its name from the fact that the sequence

$$W_0, W_1, W_2, \dots$$

is an enumeration of all r.e. sets.

We define

$$K = \{n \in N \mid n \in W_n\}.$$

Now,

$$n \in W_n \Leftrightarrow \Phi(n, n) \downarrow \Leftrightarrow \text{HALT}(n, n).$$

Thus, K is the set of all numbers n such that program number n eventually halts on input n . We have

Theorem 4.7. K is r.e. but not recursive.

Proof. Since $K = \{n \in N \mid \Phi(n, n) \downarrow\}$ and (by the universality theorem—Theorem 3.1), $\Phi(n, n)$ is certainly partially computable, K is clearly r.e. If \bar{K} were also r.e., by the enumeration theorem we would have

$$\bar{K} = W_i$$

for some i . Then

$$i \in K \Leftrightarrow i \in W_i \Leftrightarrow i \in \bar{K},$$

which is a contradiction. ■

Actually the *proof* of Theorem 2.1 already shows not only that $\text{HALT}(x, z)$ is not computable, but also that $\text{HALT}(x, x)$ is not computable, i.e., that K is not a recursive set. (This was Exercise 2.1.)

We conclude this section with some alternative ways of characterizing r.e. sets.

Theorem 4.8. Let B be an r.e. set. Then there is a primitive recursive predicate $R(x, t)$ such that $B = \{x \in N \mid (\exists t)R(x, t)\}$.

Proof. Let $B = W_n$. Then $B = \{x \in N \mid (\exists t)\text{STP}^{(1)}(x, n, t)\}$, and $\text{STP}^{(1)}$ is primitive recursive by Theorem 3.2. ■

Theorem 4.9. Let S be a nonempty r.e. set. Then there is a primitive recursive function $f(u)$ such that $S = \{f(n) \mid n \in N\} = \{f(0), f(1), f(2), \dots\}$. That is, S is the range of f .

Proof. By Theorem 4.8

$$S = \{x \mid (\exists t)R(x, t)\},$$

where R is a primitive recursive predicate. Let x_0 be some fixed member of S (for example, the smallest). Let

$$f(u) = \begin{cases} l(u) & \text{if } R(l(u), r(u)) \\ x_0 & \text{otherwise.} \end{cases}$$

Then by Theorem 5.4 in Chapter 3, f is primitive recursive. Each value $f(u)$ is in S , since x_0 is automatically in S , while if $R(l(u), r(u))$ is true, then certainly $(\exists t)R(l(u), t)$ is true, which implies that $f(u) = l(u) \in S$. Conversely, if $x \in S$, then $R(x, t_0)$ is true for some t_0 . Then

$$f(\langle x, t_0 \rangle) = l(\langle x, t_0 \rangle) = x,$$

so that $x = f(u)$ for $u = \langle x, t_0 \rangle$. ■

Theorem 4.10. Let $f(x)$ be a partially computable function and let $S = \{f(x) \mid f(x) \downarrow\}$. (That is, S is the *range* of f .) Then S is r.e.

Proof. Let

$$g(x) = \begin{cases} 0 & \text{if } x \in S \\ \uparrow & \text{otherwise.} \end{cases}$$

Since

$$S = \{x \mid g(x) \downarrow\},$$

it suffices to show that $g(x)$ is partially computable. Let \mathcal{P} be a program that computes f and let $\#(\mathcal{P}) = p$. Then the following program computes $g(x)$:

```
[A] IF ~STP(1)(Z, p, T) GOTO B
      V ← f(Z)
      IF V = X GOTO E
[B] Z ← Z + 1
      IF Z ≤ T GOTO A
      T ← T + 1
      Z ← 0
      GOTO A
```

Note that in this program the macro expansion of $V \leftarrow f(Z)$ will be entered only when the step-counter test has already guaranteed that f is defined. ■

Combining Theorems 4.9 and 4.10, we have

Theorem 4.11. Suppose that $S \neq \emptyset$. Then the following statements are all equivalent:

1. S is r.e.;
2. S is the range of a primitive recursive function;
3. S is the range of a recursive function;
4. S is the range of a partial recursive function.

Proof. By Theorem 4.9, (1) implies (2). Obviously, (2) implies (3), and (3) implies (4). By Theorem 4.10, (4) implies (1). Hence all four statements are equivalent. ■

Theorem 4.11 provides the motivation for the term *recursively enumerable*. In fact, such a set (if it is nonempty) is enumerated by a recursive function.

Exercises

1. Let B be a subset of N^m , $m > 1$. We say that B is r.e. if $B = \{(x_1, \dots, x_m) \in N^m \mid g(x_1, \dots, x_m) \downarrow\}$ for some partially computable function $g(x_1, \dots, x_m)$. Let

$$B' = \{[x_1, \dots, x_m] \in N \mid (x_1, \dots, x_m) \in B\}.$$

Show that B' is r.e. if and only if B is r.e.

2. Let $K_0 = \{\langle x, y \rangle \mid x \in W_y\}$. Show that K_0 is r.e.
3. Let f be an n -ary partial function. The graph of f , denoted $\text{gr}(f)$, is the set $\{[x_1, \dots, x_n, f(x_1, \dots, x_n)] \mid f(x_1, \dots, x_n) \downarrow\}$.
 - (a) Let \mathcal{C} be a PRC class. Prove that if f belongs to \mathcal{C} then $\text{gr}(f)$ belongs to \mathcal{C} .
 - (b) Prove that if $\text{gr}(f)$ is recursive then f is partially computable.
 - (c) Prove that the recursiveness of $\text{gr}(f)$ does not necessarily imply that f is computable.
4. Let $B = \{f(n) \mid n \in N\}$, where f is a strictly increasing computable function [i.e., $f(n+1) > f(n)$ for all n]. Prove that B is recursive.
5. Show that every infinite r.e. set has an infinite recursive subset.
6. Prove that an infinite set A is r.e. if and only if $A = \{f(n) \mid n \in N\}$ for some one-one computable function $f(x)$.

7. Let A, B be sets. Prove or disprove:
 - (a) If $A \cup B$ is r.e., then A and B are both r.e.
 - (b) If $A \subseteq B$ and B is r.e., then A is r.e.
8. Show that there is no computable function $f(x)$ such that $f(x) = \Phi(x, x) + 1$ whenever $\Phi(x, x) \downarrow$.
9. (a) Let $g(x), h(x)$ be partially computable functions. Show there is a partially computable function $f(x)$ such that $f(x) \downarrow$ for precisely those values of x for which either $g(x) \downarrow$ or $h(x) \downarrow$ (or both) and such that when $f(x) \downarrow$, either $f(x) = g(x)$ or $f(x) = h(x)$.

 (b) Can f be found fulfilling all the requirements of (a) but such that in addition $f(x) = g(x)$ whenever $g(x) \downarrow$? Proof?
10. (a) Let $A = \{y \mid (\exists t)P(t, y)\}$, where P is a computable predicate. Show that A is r.e.

 (b) Let $B = \{y \mid (\exists t_1) \dots (\exists t_n)Q(t_1, \dots, t_n, y)\}$, where Q is a computable predicate. Show that B is r.e.
11. Give a computable predicate $R(x, y)$ such that $\{y \mid (\forall t)R(t, y)\}$ is not r.e.

5. The Parameter Theorem

The parameter theorem (which has also been called the *iteration theorem* and the *s-m-n theorem*) is an important technical result that relates the various functions $\Phi^{(n)}(x_1, x_2, \dots, x_n, y)$ for different values of n .

Theorem 5.1 (Parameter Theorem). For each $n, m > 0$, there is a primitive recursive function $S_m^n(u_1, u_2, \dots, u_n, y)$ such that

$$\Phi^{(m+n)}(x_1, \dots, x_m, u_1, \dots, u_n, y) = \Phi^{(m)}(x_1, \dots, x_m, S_m^n(u_1, \dots, u_n, y)). \quad (5.1)$$

Suppose that values for variables u_1, \dots, u_n are fixed and we have in mind some particular value of y . Then the left side of (5.1) is a partially computable function of the m arguments x_1, \dots, x_m . Letting q be the number of a program that computes this function of m variables, we have

$$\Phi^{(m+n)}(x_1, \dots, x_m, u_1, \dots, u_n, y) = \Phi^{(m)}(x_1, \dots, x_m, q).$$

The parameter theorem tells us that not only does there exist such a number q , but that it can be obtained from u_1, \dots, u_n, y in a computable (in fact, primitive recursive) way.

Proof. The proof is by mathematical induction on n .

For $n = 1$, we need to show that there is a primitive recursive function $S_m^1(u, y)$ such that

$$\Phi^{(m+1)}(x_1, \dots, x_m, u, y) = \Phi^{(m)}(x_1, \dots, x_m, S_m^1(u, y)).$$

Here $S_m^1(u, y)$ must be the number of a program which, given m inputs x_1, \dots, x_m , computes the same value as program number y does when given the $m + 1$ inputs x_1, \dots, x_m, u . Let \mathcal{P} be the program such that $\#(\mathcal{P}) = y$. Then $S_m^1(u, y)$ can be taken to be the number of a program which first gives the variable X_{m+1} the value u and then proceeds to carry out \mathcal{P} . X_{m+1} will be given the value u by the program

$$\left. \begin{array}{l} X_{m+1} \leftarrow X_{m+1} + 1 \\ \vdots \\ X_{m+1} \leftarrow X_{m+1} + 1 \end{array} \right\} u$$

The number of the unlabeled instruction

$$X_{m+1} \leftarrow X_{m+1} + 1$$

is

$$\langle 0, \langle 1, 2m + 1 \rangle \rangle = 16m + 10.$$

So we may take

$$S_m^1(u, y) = \left[\left(\prod_{i=1}^u p_i \right)^{16m+10} \prod_{j=1}^{\text{Lt}(y+1)} p_{u+j}^{(y+1)_j} \right] \dot{-} 1,$$

a primitive recursive function. Here the numbers of the instructions of \mathcal{P} which appear as exponents in the prime power factorization of $y + 1$ have been shifted to the primes $p_{u+1}, p_{u+2}, \dots, p_{u+\text{Lt}(y+1)}$.

To complete the proof, suppose the result known for $n = k$. Then we have

$$\begin{aligned} \Phi^{(m+k+1)}(x_1, \dots, x_m, u_1, \dots, u_k, u_{k+1}, y) \\ = \Phi^{(m+k)}(x_1, \dots, x_m, u_1, \dots, u_k, S_{m+k}^1(u_{k+1}, y)) \\ = \Phi^{(m)}(x_1, \dots, x_m, S_m^k(u_1, \dots, u_k, S_{m+k}^1(u_{k+1}, y))), \end{aligned}$$

using first the result for $n = 1$ and then the induction hypothesis. But now, if we define

$$S_m^{k+1}(u_1, \dots, u_k, u_{k+1}, y) = S_m^k(u_1, \dots, u_k, S_{m+k}^1(u_{k+1}, y)),$$

we have the desired result. ■

We next give a sample application of the parameter theorem. It is desired to find a computable function $g(u, v)$ such that

$$\Phi_u(\Phi_v(x)) = \Phi_{g(u, v)}(x).$$

We have by the meaning of the notation that

$$\Phi_u(\Phi_v(x)) = \Phi(\Phi(x, v), u)$$

is a partially computable function of x, u, v . Hence, we have

$$\Phi_u(\Phi_v(x)) = \Phi^{(3)}(x, u, v, z_0)$$

for some number z_0 . By the parameter theorem,

$$\Phi^{(3)}(x, u, v, z_0) = \Phi(x, S_1^2(u, v, z_0)) = \Phi_{S_1^2(u, v, z_0)}(x).$$

Exercises

- Given a partially computable function $f(x, y)$, find a primitive recursive function $g(u, v)$ such that

$$\Phi_{g(u, v)}(x) = f(\Phi_u(x), \Phi_v(x)).$$

- Show that there is a primitive recursive function $g(u, v, w)$ such that

$$\Phi^{(3)}(u, v, w, z) = \Phi_{g(u, v, w)}(z).$$

- Let us call a partially computable function $g(x)$ *extendable* if there is a computable function $f(x)$ such that $f(x) = g(x)$ for all x for which $g(x) \downarrow$. Show that there is no algorithm for determining of a given z whether or not $\Phi_z(x)$ is extendable. [Hint: Exercise 8 of Section 4 shows that $\Phi(x, x) + 1$ is not extendable. Find an extendable function $k(x)$ such that the function

$$h(x, t) = \begin{cases} \Phi(x, x) + 1 & \text{if } \Phi(t, t) \downarrow \\ k(x) & \text{otherwise} \end{cases}$$

is partially computable.]

- 4.* A *programming system* is an enumeration $S = \{\phi_i^{(n)} \mid i \in N, n > 0\}$ of the partially computable functions. That is, for each partially computable function $f(x_1, \dots, x_n)$ there is an i such that f is $\phi_i^{(n)}$.
- (a) A programming system S is *universal* if for each $n > 0$, the function $\Psi^{(n)}$, defined

$$\Psi^{(n)}(x_1, \dots, x_n, i) = \phi_i^{(n)}(x_1, \dots, x_n),$$

is partially computable. That is, S is universal if a version of the universality theorem holds for S . Obviously,

$$\{\Phi_i^{(n)} \mid i \in N, n > 0\}$$

is a universal programming system. Prove that a programming system S is universal if and only if for each $n > 0$ there is a computable function f_n such that $\phi_i^{(n)} = \Phi_{f_n(i)}^{(n)}$ for all i .

- (b) A universal programming system S is *acceptable* if for each $n, m > 0$ there is a computable function $s_m^n(u_1, \dots, u_n, y)$ such that

$$\begin{aligned} \Psi^{(m+n)}(x_1, \dots, x_m, u_1, \dots, u_n, y) \\ = \Psi^{(m)}(x_1, \dots, x_m, s_m^n(u_1, \dots, u_n, y)). \end{aligned}$$

That is, S is acceptable if a version of the parameter theorem holds for S . Again, $\{\Phi_i^{(n)} \mid i \in N, n > 0\}$ is obviously an acceptable programming system. Prove that S is acceptable if and only if for each $n > 0$ there is a computable function g_n such that $\Phi_i^{(n)} = \phi_{g_n(i)}^{(n)}$ for all i .

6. Diagonalization and Reducibility

So far we have seen very few examples of nonrecursive sets. We now discuss two general techniques for proving that given sets are not recursive or even that they are not r.e. The first method, *diagonalization*, turns on the demonstration of two assertions of the following sort:

1. A certain set A can be enumerated in a suitable fashion.
2. It is possible, with the help of the enumeration, to define an object b that is different from every object in the enumeration, i.e., $b \notin A$.

We sometimes say that b is defined by *diagonalizing over A* . In some diagonalization arguments the goal is simply to find some $b \notin A$. We will give an example of such an argument later in the chapter. The arguments we will consider in this section have an additional twist: the definition of b is such that b must belong to A , contradicting the assertion that we began

with an enumeration of *all* of the elements in A . The end of the argument, then, is to draw some conclusion from this contradiction.

For example, the proof given for Theorem 2.1 is a diagonalization argument that the predicate $\text{HALT}(x, y)$, or equivalently, the set

$$\{(x, y) \in N^2 \mid \text{HALT}(x, y)\},$$

is not computable. The set A in this case is the class of unary partially computable functions, and assertion 1 follows from the fact that \mathcal{S} programs can be coded as numbers. For each n , let \mathcal{P}_n be the program with number n . Then all unary partially computable functions occur among $\psi_{\mathcal{P}_0}^{(1)}, \psi_{\mathcal{P}_1}^{(1)}, \dots$. We began by assuming that $\text{HALT}(x, y)$ is computable, and we wrote a program \mathcal{P} that computes $\psi_{\mathcal{P}}^{(1)}$. The heart of the proof consisted of showing that $\psi_{\mathcal{P}}^{(1)}$ does not appear among $\psi_{\mathcal{P}_0}^{(1)}, \psi_{\mathcal{P}_1}^{(1)}, \dots$. In particular, we wrote \mathcal{P} so that for every x , $\psi_{\mathcal{P}}^{(1)}(x) \downarrow$ if and only if $\psi_{\mathcal{P}_x}^{(1)}(x) \uparrow$, i.e.,

$$\text{HALT}(x, \#(\mathcal{P})) \Leftrightarrow \sim \text{HALT}(x, x),$$

so $\psi_{\mathcal{P}}^{(1)}$ differs from each function $\psi_{\mathcal{P}_0}^{(1)}, \psi_{\mathcal{P}_1}^{(1)}, \dots$ on at least one input value. That is, n is a counterexample to the possibility that $\psi_{\mathcal{P}}^{(1)}$ is $\psi_{\mathcal{P}_n}^{(1)}$, since $\psi_{\mathcal{P}}^{(1)}(n) \downarrow$ if and only if $\psi_{\mathcal{P}_n}^{(1)}(n) \uparrow$. Now we have the unary partially computable function $\psi_{\mathcal{P}}^{(1)}$ that is not among $\psi_{\mathcal{P}_0}^{(1)}, \psi_{\mathcal{P}_1}^{(1)}, \dots$, so assertion 2 is satisfied, giving us a contradiction. In the proof of Theorem 2.1 the contradiction was expressed a bit differently: Because $\psi_{\mathcal{P}}^{(1)}$ is partially computable, it *must* appear among $\psi_{\mathcal{P}_0}^{(1)}, \psi_{\mathcal{P}_1}^{(1)}, \dots$, and, in particular, it must be $\psi_{\mathcal{P}_{\#(\mathcal{P})}}^{(1)}$, since $\mathcal{P}_{\#(\mathcal{P})}$ is \mathcal{P} by definition, but we have the counterexample $\psi_{\mathcal{P}}^{(1)}(\#(\mathcal{P})) \downarrow$ if and only if $\psi_{\mathcal{P}_{\#(\mathcal{P})}}^{(1)}(\#(\mathcal{P})) \uparrow$, i.e.,

$$\text{HALT}(\#(\mathcal{P}), \#(\mathcal{P})) \Leftrightarrow \sim \text{HALT}(\#(\mathcal{P}), \#(\mathcal{P})).$$

Since we know assertion 1 to be true, and since assertion 2 depended on the assumption that $\text{HALT}(x, y)$ is computable, $\text{HALT}(x, y)$ cannot be computable.

To present the situation more graphically, we can represent the values of each function $\psi_{\mathcal{P}_0}^{(1)}, \psi_{\mathcal{P}_1}^{(1)}, \dots$ by the infinite array

$\boxed{\psi_{\mathcal{P}_0}^{(1)}(0)}$	$\psi_{\mathcal{P}_0}^{(1)}(1)$	$\psi_{\mathcal{P}_0}^{(1)}(2)$	\dots
$\psi_{\mathcal{P}_1}^{(1)}(0)$	$\boxed{\psi_{\mathcal{P}_1}^{(1)}(1)}$	$\psi_{\mathcal{P}_1}^{(1)}(2)$	\dots
$\psi_{\mathcal{P}_2}^{(1)}(0)$	$\psi_{\mathcal{P}_2}^{(1)}(1)$	$\boxed{\psi_{\mathcal{P}_2}^{(1)}(2)}$	\dots
\vdots	\vdots	\vdots	

Each row represents one function. It is along the *diagonal* of this array that we have arranged to find the counterexamples, which explains the origin of the term *diagonalization*.

We can use a similar argument to give an example of a non-r.e. set. Let TOT be the set of all numbers p such that p is the number of a program that computes a total function $f(x)$ of one variable. That is,

$$\text{TOT} = \{z \in N \mid (\forall x)\Phi(x, z) \downarrow\}.$$

Since

$$\Phi(x, z) \downarrow \Leftrightarrow x \in W_z,$$

TOT is simply the set of numbers z such that W_z is the set of all nonnegative integers.

We have

Theorem 6.1. TOT is not r.e.

Proof. Suppose that TOT were r.e. Since $\text{TOT} \neq \emptyset$, by Theorem 4.9 there is a computable function $g(x)$ such that $\text{TOT} = \{g(0), g(1), g(2), \dots\}$. Let

$$h(x) = \Phi(x, g(x)) + 1.$$

Since each value $g(x)$ is the number of a program that computes a total function, $\Phi(u, g(x)) \downarrow$ for all x, u and hence, in particular, $h(x) \downarrow$ for all x . Thus h is itself a computable function. Let h be computed by program \mathcal{P} , and let $p = \#(\mathcal{P})$. Then $p \in \text{TOT}$, so that $p = g(i)$ for some i . Then

$$\begin{aligned} h(i) &= \Phi(i, g(i)) + 1 && \text{by definition of } h \\ &= \Phi(i, p) + 1 && \text{since } p = g(i) \\ &= h(i) + 1 && \text{since } h \text{ is computed by } \mathcal{P}, \end{aligned}$$

which is a contradiction. ■

Note that in the proof of Theorem 6.1, the set A is TOT itself, and this time assertion 1 was taken as an assumption, while assertion 2 is shown to be true. Theorem 6.1 helps to explain why we base the study of computability on partial functions rather than total functions. By Church's thesis, Theorem 6.1 implies that there is no algorithm to determine if an \mathcal{S} program computes a total function.

Once some set such as K has been shown to be nonrecursive, we can use that set to give other examples of nonrecursive sets by way of the *reducibility* method.

Definition. Let A, B be sets. A is *many-one reducible to* B , written $A \leq_m B$, if there is a computable function f such that

$$A = \{x \in N \mid f(x) \in B\}.$$

That is, $x \in A$ if and only if $f(x) \in B$. (The word *many-one* simply refers to the fact that we do not require f to be one-one.)

If $A \leq_m B$, then in a sense testing membership in A is “no harder than” testing membership in B . In particular, to test $x \in A$, we can compute $f(x)$ and then test $f(x) \in B$.

Theorem 6.2. Suppose $A \leq_m B$.

1. If B is recursive, then A is recursive.
2. If B is r.e., then A is r.e.

Proof. Let $A = \{x \in N \mid f(x) \in B\}$, where f is computable, and let $P_B(x)$ be the characteristic function of B . Then

$$A = \{x \in N \mid P_B(f(x))\},$$

and if B is recursive then $P_B(f(x))$, the characteristic function of A , is computable.

Now suppose that B is r.e. Then $B = \{x \in N \mid g(x) \downarrow\}$ for some partially computable function g , and $A = \{x \in N \mid g(f(x)) \downarrow\}$. But $g(f(x))$ is partially computable, so A is r.e. ■

We generally use Theorem 6.2 in the form: If A is not recursive (r.e.), then B is not recursive (respectively, not r.e.). For example, let

$$K_0 = \{x \in N \mid \Phi_{r(x)}(l(x)) \downarrow\} = \{\langle x, y \rangle \mid \Phi_y(x) \downarrow\}.$$

K_0 is clearly r.e. However, we can show by reducing K to K_0 , that is, by showing that $K \leq_m K_0$, that K_0 is not recursive: $x \in K$ if and only if $\langle x, x \rangle \in K_0$, and the function $f(x) = \langle x, x \rangle$ is computable. In fact, it is easy to show that every r.e. set is many-one reducible to K_0 : if A is r.e., then

$$\begin{aligned} A &= \{x \in N \mid g(x) \downarrow\} && \text{for some partially computable } g \\ &= \{x \in N \mid \Phi(x, z_0) \downarrow\} && \text{for some } z_0 \\ &= \{x \in N \mid \langle x, z_0 \rangle \in K_0\}. \end{aligned}$$

Definition. A set A is *m-complete* if

1. A is r.e., and
2. for every r.e. set B , $B \leq_m A$.

So K_0 is m-complete. We can also show that K is m-complete. First we show that $K_0 \leq_m K$. This argument is somewhat more involved because K_0 seems, at first glance, to contain more information than K . K_0 represents the halting behavior of all partially computable functions on all inputs, while K represents only the halting behavior of partially computable functions on a single argument. We wish to take a pair $\langle n, q \rangle$ and transform it to a number $f(\langle n, q \rangle)$ of a single program such that

$$\Phi_q(n) \downarrow \text{ if and only if } \Phi_{f(\langle n, q \rangle)}(f(\langle n, q \rangle)) \downarrow,$$

i.e., such that $\langle n, q \rangle \in K_0$ if and only if $f(\langle n, q \rangle) \in K$. The parameter theorem turns out to be very useful here. Let \mathcal{P} be the program

$$Y \leftarrow \Phi^{(1)}(l(X_2), r(X_2))$$

and let $p = \#(\mathcal{P})$. Then $\psi_{\mathcal{P}}(x_1, x_2) = \Phi^{(1)}(l(x_2), r(x_2))$, and

$$\psi_{\mathcal{P}}(x_1, x_2) = \Phi^{(2)}(x_1, x_2, p) = \Phi^{(1)}(x_1, S_1^1(x_2, p))$$

by the parameter theorem, so for any pair $\langle n, q \rangle$,

$$\Phi^{(1)}(n, q) = \psi_{\mathcal{P}}(x_1, \langle n, q \rangle) = \Phi_{S_1^1(\langle n, q \rangle, p)}^{(1)}(x_1). \quad (6.1)$$

Now, (6.1) holds for all values of x_1 , so, in particular,

$$\Phi^{(1)}(n, q) = \Phi_{S_1^1(\langle n, q \rangle, p)}^{(1)}(S_1^1(\langle n, q \rangle, p)),$$

and therefore

$$\Phi^{(1)}(n, q) \downarrow \text{ if and only if } \Phi_{S_1^1(\langle n, q \rangle, p)}^{(1)}(S_1^1(\langle n, q \rangle, p)) \downarrow,$$

i.e.,

$$\langle n, q \rangle \in K_0 \text{ if and only if } S_1^1(\langle n, q \rangle, p) \in K.$$

With p held constant $S_1^1(x, p)$ is a computable unary function, so $K_0 \leq_m K$.

To complete the argument that K is m-complete we need

Theorem 6.3. If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$.

Proof. Let $A = \{x \in N \mid f(x) \in B\}$ and $B = \{x \in N \mid g(x) \in C\}$. Then $A = \{x \in N \mid g(f(x)) \in C\}$, and $g(f(x))$ is computable. ■

As an immediate consequence we have

Corollary 6.4. If A is m-complete, B is r.e., and $A \leq_m B$, then B is m-complete.

Proof. If C is r.e. then $C \leq_m A$, and $A \leq_m B$ by assumption, so $C \leq_m B$. ■

Thus, K is m-complete. Informally, testing membership in an m-complete set is “at least as difficult as” testing membership in any r.e. set. So an m-complete set is a good choice for showing by a reducibility argument that a given set is not computable. We expand on this subject in Chapter 8.

Actually, we have shown both $K \leq_m K_0$ and $K_0 \leq_m K$, so in a sense, testing membership in K and testing membership in K_0 are “equally difficult” problems.

Definition. $A \equiv_m B$ means that $A \leq_m B$ and $B \leq_m A$.

In general, for sets A and B , if $A \equiv_m B$ then testing membership in A has the “same difficulty as” testing membership in B .

To summarize, we have proved

Theorem 6.5.

1. K and K_0 are m-complete.
2. $K \equiv_m K_0$.

We can also use reducibility arguments to show that certain sets are not r.e. Let

$$\text{EMPTY} = \{x \in N \mid W_x = \emptyset\}.$$

Theorem 6.6. EMPTY is not r.e.

Proof. We will show that $\bar{K} \leq_m \text{EMPTY}$. \bar{K} is not r.e., so by Theorem 6.2, EMPTY is not r.e. Let \mathcal{P} be the program

$$Y \leftarrow \Phi(X_2, X_2),$$

and let $p = \#(\mathcal{P})$. \mathcal{P} ignores its first argument, so for a given z ,

$$\psi_{\mathcal{P}}^{(2)}(x, z) \downarrow \text{ for all } x \text{ if and only if } \Phi(z, z) \downarrow.$$

By the parameter theorem

$$\psi_{\mathcal{P}}^{(2)}(x_1, x_2) = \Phi^{(2)}(x_1, x_2, p) = \Phi^{(1)}(x_1, S_1^1(x_2, p)),$$

so, for any z ,

$$\begin{aligned} z \in \bar{K} &\text{ if and only if } \Phi(z, z) \uparrow \\ &\text{if and only if } \psi_{\mathcal{P}}^{(2)}(x, z) \uparrow \text{ for all } x \\ &\text{if and only if } \Phi^{(1)}(x, S_1^1(z, p)) \uparrow \text{ for all } x \\ &\text{if and only if } W_{S_1^1(z, p)} = \emptyset \\ &\text{if and only if } S_1^1(z, p) \in \text{EMPTY}. \end{aligned}$$

$f(z) = S_1^1(z, p)$ is computable, so $\bar{K} \leq_m \text{EMPTY}$. ■

Exercises

1. Show that the proof of Theorem 4.7 is a diagonalization argument.
2. Prove by diagonalization that there is no enumeration f_0, f_1, f_2, \dots of all total unary (not necessarily computable) functions on N .
3. Let $A = \{x \in N \mid \Phi_x(x) \downarrow \text{ and } \Phi_x(x) > x\}$.
 - (a) Show that A is r.e.
 - (b) Show by diagonalization that A is not recursive.
4. Show how the diagonalization argument in the proof of Theorem 6.1 fails for the set of all numbers p such that p is the number of a program that computes a partial function, i.e., the set N .
5. Let A, B be sets of numbers. Prove
 - (a) $A \leq_m A$.
 - (b) $A \leq_m B$ if and only if $\bar{A} \leq_m \bar{B}$.
6. Prove that no m-complete set is recursive.
7. Let A, B be m-complete. Show that $A \equiv_m B$.
8. Prove that $\bar{K} \not\leq_m K$, i.e., \bar{K} is not many-one reducible to K .
9. For every number n , let $A_n = \{x \mid n \in W_x\}$.
 - (a) Show that A_i is r.e. but not recursive, for all i .
 - (b) Show that $A_i \equiv_m A_j$ for all i, j .
10. Define the predicate $P(x) \Leftrightarrow \Phi_x(x) = 1$. Show that $P(x)$ is not computable.
11. Define the predicate

$Q(x) \Leftrightarrow$ the variable Y assumes the value 1 sometime during the computation of $\psi_{\mathcal{P}}(x)$, where $\#(\mathcal{P}) = x$.

Show that $Q(x)$ is not computable. [Hint: Use the parameter theorem and a version of the universal program \mathcal{U}_1 .]

12. Let $\text{INF} = \{x \in N \mid W_x \text{ is infinite}\}$. Show that $\text{INF} \equiv_m \text{TOTAL}$.
13. Let $\text{FIN} = \{x \in N \mid W_x \text{ is finite}\}$. Show that $\bar{K} \leq_m \text{FIN}$.
- 14.* Let

$$\text{MONOTONE} = \{y \in N \mid \Phi_y(x) \text{ is total and}$$

$$\Phi_y(x) \leq \Phi_y(x+1) \text{ for all } x\}.$$

- (a) Show by diagonalization that MONOTONE is not r.e.
- (b) Show that $\text{MONOTONE} \equiv_m \text{TOTAL}$.

7. Rice's Theorem

Using the reducibility method we can prove a theorem that gives us, at a single stroke, a wealth of interesting unsolvable problems concerning programs.

Let Γ be some collection of partially computable functions of one variable. We may associate with Γ the set (usually called an *index set*)

$$R_\Gamma = \{t \in N \mid \Phi_t \in \Gamma\}.$$

R_Γ is a recursive set just in case the predicate $g(t)$, defined $g(t) \Leftrightarrow \Phi_t \in \Gamma$, is computable. Consider the examples:

1. Γ is the set of computable functions;
2. Γ is the set of primitive recursive functions;
3. Γ is the set of partially computable functions that are defined for all but a finite number of values of x .

These examples make it plain that it would be interesting to be able to show that R_Γ is computable for various collections Γ . Invoking Church's thesis, we can say that R_Γ is a recursive set just in case there is an algorithm that accepts *programs* \mathcal{P} as input and returns the value TRUE or FALSE depending on whether or not the function $\psi_{\mathcal{P}}^{(1)}$ does or does not belong to Γ . In fact, those who work with computer programs would be very pleased to possess algorithms that accept a program as input and which return as output some useful property of the partial function computed by that program. Alas, such algorithms are not to be found! This dismal conclusion follows from Rice's theorem.

Theorem 7.1 (Rice's Theorem). Let Γ be a collection of partially computable functions of one variable. Let there be partially computable functions $f(x), g(x)$ such that $f(x)$ belongs to Γ but $g(x)$ does not. Then R_Γ is not recursive.

Proof. Let $h(x)$ be the function such that $h(x) \uparrow$ for all x . We assume first that $h(x)$ does not belong to Γ . Let q be the number of

$$\begin{aligned} Z &\leftarrow \Phi(X_2, X_2) \\ Y &\leftarrow f(X_1) \end{aligned}$$

Then, for any i , $S_1^1(i, q)$ is the number of

$$\begin{aligned} X_2 &\leftarrow i \\ Z &\leftarrow \Phi(X_2, X_2) \\ Y &\leftarrow f(X_1) \end{aligned}$$

Now

$$\begin{aligned} i \in K &\text{ implies } \Phi(i, i) \downarrow \\ &\text{implies } \Phi_{S_1^1(i, q)}(x) = f(x) \text{ for all } x \\ &\text{implies } \Phi_{S_1^1(i, q)} \in \Gamma \\ &\text{implies } S_1^1(i, q) \in R_\Gamma, \end{aligned}$$

and

$$\begin{aligned} i \notin K &\text{ implies } \Phi(i, i) \uparrow \\ &\text{implies } \Phi_{S_1^1(i, q)}(x) \uparrow \text{ for all } x \\ &\text{implies } \Phi_{S_1^1(i, q)} = h \\ &\text{implies } \Phi_{S_1^1(i, q)} \notin \Gamma \\ &\text{implies } S_1^1(i, q) \notin R_\Gamma, \end{aligned}$$

so $K \leq_m R_\Gamma$. By Theorem 6.2, R_Γ is not recursive.

If $h(x)$ does belong to Γ , then the same argument with Γ and $f(x)$ replaced by $\bar{\Gamma}$ and $g(x)$ shows that $R_{\bar{\Gamma}}$ is not recursive. But $R_{\bar{\Gamma}} = R_\Gamma$, so, by Theorem 4.1, R_Γ is not recursive in this case either. ■

Corollary 7.2. There are no algorithms for testing a given program \mathcal{P} of the language \mathcal{S} to determine whether $\psi_{\mathcal{P}}^{(1)}(x)$ belongs to any of the classes described in Examples 1–3.

Proof. In each case we only need find the required functions $f(x), g(x)$ to show that R_Γ is not recursive. The corollary then follows by Church's

thesis. For 1, 2, or 3 we can take, for example, $f(x) = u_1^1(x)$ and $g(x) = 1 - x$ [so that $g(x)$ is defined only for $x = 0, 1$]. ■

Exercises

1. Show that Rice's theorem is false if the requirement for functions $f(x), g(x)$ is omitted.
2. Show there is no algorithm to determine of a given program \mathcal{P} in the language \mathcal{S} whether $\psi_{\mathcal{P}}(x) = x^2$ for all x .
3. Show that there is no algorithm to determine of a pair of numbers u, v whether $\Phi_u(x) = \Phi_v(x)$ for all x .
4. Show that the set $A = \{x \mid \Phi_x \text{ is defined for at least one input}\}$ is r.e. but not recursive.
5. Use Rice's theorem to show that the following sets are not recursive. [See Section 6 for the definitions of the sets.]
 - (a) TOT;
 - (b) EMPTY;
 - (c) INF;
 - (d) FIN;
 - (e) MONOTONE;
 - (f) $\{y \in N \mid \Phi_y^{(1)} \text{ is a predicate}\}$.
6. Let Γ be a collection of partially computable functions of m variables, $m > 1$, and let $R_{\Gamma}^{(m)} = \{t \in N \mid \Phi_t^{(m)} \in \Gamma\}$. State and prove a version of Rice's theorem for collections of partially computable functions of m variables, $m > 1$.
7. Define the predicate

$$\text{PROPER}(n) \Leftrightarrow \min_z [\Phi_n^{(2)}(x, z) = 3] \text{ is an application of proper minimalization to the predicate } \Phi_n^{(2)}(x, z) = 3.$$

Show that $\text{PROPER}(x)$ is not computable.

8. Let Γ be a set of partially computable functions of one variable. Show that R_{Γ} is r.e. if and only if it is m-complete.

*8. The Recursion Theorem

In the proof that $\text{HALT}(x, y)$ is not computable, we gave (assuming $\text{HALT}(x, y)$ to be computable) a program \mathcal{P} such that

$$\text{HALT}(\#(\mathcal{P}), \#(\mathcal{P})) \Leftrightarrow \sim \text{HALT}(\#(\mathcal{P}), \#(\mathcal{P})).$$

We get a contradiction when we consider the behavior of the program \mathcal{P} on input $\#(\mathcal{P})$. The phenomenon of a program acting on its own description is sometimes called *self-reference*, and it is the source of many fundamental results in computability theory. Indeed, the whole point of diagonalization in the proof of Theorem 2.1 is to get a contradictory self-reference. We turn now to a theorem which packages, so to speak, a general technique for obtaining self-referential behavior. It is one of the most important applications of the parameter theorem.

Theorem 8.1 (Recursion Theorem). Let $g(z, x_1, \dots, x_m)$ be a partially computable function of $m + 1$ variables. Then there is a number e such that

$$\Phi_e^{(m)}(x_1, \dots, x_m) = g(e, x_1, \dots, x_m).$$

Discussion. Let $e = \#(\mathcal{P})$, so that $\psi_{\mathcal{P}}^{(m)}(x_1, \dots, x_m) = \Phi_e^{(m)}(x_1, \dots, x_m)$. The equality in the theorem says that the m -ary function $\psi_{\mathcal{P}}^{(m)}(x_1, \dots, x_m)$ is equal to $g(z, x_1, \dots, x_m)$ when the first argument of g is held constant at e . That is, \mathcal{P} is a program that, in effect, gets access to its own number, e , and computes the m -ary function $g(e, x_1, \dots, x_m)$. Note that since x_1, \dots, x_m can be arbitrary values, e generally does not appear among the inputs to $\psi_{\mathcal{P}}^{(m)}(x_1, \dots, x_m)$, so \mathcal{P} must somehow compute e . One might suppose that \mathcal{P} might contain e copies of an instruction such as $Z \leftarrow Z + 1$, that is, an expansion of the macro $Z \leftarrow e$, but if \mathcal{P} has at least e instructions, then certainly $\#(\mathcal{P}) > e$. The solution is to write \mathcal{P} so that it computes e without having e “built in” to the program. In particular, we build into \mathcal{P} a “partial description” of \mathcal{P} , and then have \mathcal{P} compute e from the partial description. Let \mathcal{Q} be the program

$$Z \leftarrow S_m^1(X_{m+1}, X_{m+1})$$

$$Y \leftarrow g(Z, X_1, \dots, X_m)$$

We prefix $\#(\mathcal{Q})$ copies of the instruction $X_{m+1} \leftarrow X_{m+1} + 1$ to get the program \mathcal{R} :

$$X_{m+1} \leftarrow X_{m+1} + 1$$

$$\vdots \qquad \vdots$$

$$X_{m+1} \leftarrow X_{m+1} + 1$$

$$Z \leftarrow S_m^1(X_{m+1}, X_{m+1})$$

$$Y \leftarrow g(Z, X_1, \dots, X_m)$$

After the first $\#(\mathcal{Q})$ instructions are executed, X_{m+1} holds the value $\#(\mathcal{Q})$, and $S_m^1(\#(\mathcal{Q}), \#(\mathcal{Q}))$, as defined in the proof of the parameter theorem, computes the number of the program consisting of $\#(\mathcal{Q})$ copies of $X_{m+1} \leftarrow X_{m+1} + 1$ followed by program \mathcal{Q} . But that program is \mathcal{R} . So $Z \leftarrow S_m^1(X_{m+1}, X_{m+1})$ gives Z the value $\#(\mathcal{R})$, and $Y \leftarrow g(Z, X_1, \dots, X_m)$ causes \mathcal{R} to output $g(\#(\mathcal{R}), x_1, \dots, x_m)$. We take e to be $\#(\mathcal{R})$ and we have

$$\Phi_e^{(m)}(x_1, \dots, x_m) = \psi_{\mathcal{R}}^{(m)}(x_1, \dots, x_m) = g(e, x_1, \dots, x_m).$$

We now formalize this argument.

Proof. Consider the partially computable function

$$g(S_m^1(v, v), x_1, \dots, x_m)$$

where S_m^1 is the function that occurs in the parameter theorem. Then we have for some number z_0 ,

$$\begin{aligned} g(S_m^1(v, v), x_1, \dots, x_m) &= \Phi^{(m+1)}(x_1, \dots, x_m, v, z_0) \\ &= \Phi^{(m)}(x_1, \dots, x_m, S_m^1(v, z_0)), \end{aligned}$$

where we have used the parameter theorem. Setting $v = z_0$ and $e = S_m^1(z_0, z_0)$, we have

$$g(e, x_1, \dots, x_m) = \Phi^{(m)}(x_1, \dots, x_m, e) = \Phi_e^{(m)}(x_1, \dots, x_m). \quad \blacksquare$$

We can use the recursion theorem to give another self-referential proof that $\text{HALT}(x, y)$ is not computable. If $\text{HALT}(x, y)$ were computable, then

$$f(x, y) = \begin{cases} \uparrow & \text{if } \text{HALT}(y, x) \\ 0 & \text{otherwise} \end{cases}$$

would be partially computable, so by the recursion theorem there would be a number e such that

$$\Phi_e(y) = f(e, y) = \begin{cases} \uparrow & \text{if } \text{HALT}(y, e) \\ 0 & \text{otherwise,} \end{cases}$$

that is,

$$\sim \text{HALT}(y, e) \Leftrightarrow \text{HALT}(y, e).$$

So $\text{HALT}(x, y)$ is not computable. The self-reference occurs when Φ_e computes e , tests $\text{HALT}(y, e)$, and then does the opposite of what $\text{HALT}(y, e)$ says it does.

One of the many uses of the recursion theorem is to allow us to write down definitions of functions that involve the program used to compute the function as part of its definition. For a simple example we give

Corollary 8.2. There is a number e such that for all x

$$\Phi_e(x) = e.$$

Proof. We consider the computable function

$$g(z, x) = u_1^2(z, x) = z.$$

Applying the recursion theorem we obtain a number e such that

$$\Phi_e(x) = g(e, x) = e$$

and we are done. ■

It is tempting to be a little metaphorical about this result. The program with number e “consumes” its “environment” (i.e., the input x) and outputs a “copy” of itself. That is, it is, in miniature, a self-reproducing organism. This program has often been cited in considerations of the comparison between living organisms and machines.

For another example, let

$$f(x, t) = \begin{cases} k & \text{if } t = 0 \\ g(t - 1, \Phi_x(t - 1)) & \text{otherwise,} \end{cases}$$

where $g(x, y)$ is computable. It is clear that $f(x, t)$ is partially computable, so by the recursion theorem there is a number e such that

$$\Phi_e(t) = f(e, t) = \begin{cases} k & \text{if } t = 0 \\ g(t - 1, \Phi_e(t - 1)) & \text{otherwise.} \end{cases}$$

An easy induction argument on t shows that Φ_e is a total, and therefore computable, function. Now, Φ_e satisfies the equations

$$\Phi_e(0) = k$$

$$\Phi_e(t + 1) = g(t, \Phi_e(t)),$$

that is, Φ_e is obtained from g by primitive recursion of the form (2.1) in Chapter 3, so the recursion theorem gives us another proof of Theorem 2.1 in Chapter 3. In fact, the recursion theorem can be used to justify definitions based on much more general forms of recursion, which explains how it came by its name.¹ We give one more example, in which we wish to

¹ For more on this subject, see Part 5.

know if there are partially computable functions f, g that satisfy the equations

$$\begin{aligned} f(0) &= 1 \\ f(t+1) &= g(2t) + 1 \\ g(0) &= 3 \\ g(2t+2) &= f(t) + 2. \end{aligned} \tag{8.1}$$

Let $F(z, t)$ be the partially computable function

$$F(z, x) = \begin{cases} 1 & \text{if } x = \langle 0, 0 \rangle \\ \Phi_z(\langle 1, 2(r(x) \dot{-} 1) \rangle) + 1 & \text{if } (\exists y)_{\leq x}(x = \langle 0, y + 1 \rangle) \\ 3 & \text{if } x = \langle 1, 0 \rangle \\ \Phi_z(\langle 0, \lfloor (r(x) \dot{-} 2)/2 \rfloor \rangle) + 2 & \text{if } (\exists y)_{\leq x}(x = \langle 1, 2y + 2 \rangle). \end{cases}$$

By the recursion theorem there is a number e such that

$$\Phi_e(x) = F(e, x)$$

$$= \begin{cases} 1 & \text{if } x = \langle 0, 0 \rangle \\ \Phi_e(\langle 1, 2(r(x) \dot{-} 1) \rangle) + 1 & \text{if } (\exists y)_{\leq x}(x = \langle 0, y + 1 \rangle) \\ 3 & \text{if } x = \langle 1, 0 \rangle \\ \Phi_e(\langle 0, \lfloor (r(x) \dot{-} 2)/2 \rfloor \rangle) + 2 & \text{if } (\exists y)_{\leq x}(x = \langle 1, 2y + 2 \rangle). \end{cases}$$

Now, setting

$$f(x) = \Phi_e(\langle 0, x \rangle) \quad \text{and} \quad g(x) = \Phi_e(\langle 1, x \rangle)$$

we have

$$f(0) = \Phi_e(\langle 0, 0 \rangle) = 1$$

$$f(t+1) = \Phi_e(\langle 0, t+1 \rangle) = \Phi_e(\langle 1, 2t \rangle) + 1 = g(2t) + 1$$

$$g(0) = \Phi_e(\langle 1, 0 \rangle) = 3$$

$$g(2t+2) = \Phi_e(\langle 1, 2t+2 \rangle) = \Phi_e(\langle 0, t \rangle) + 2 = f(t) + 2,$$

so f, g satisfy (8.1).

Another application of the recursion theorem is

Theorem 8.3 (Fixed Point Theorem). Let $f(z)$ be a computable function. Then there is a number e such that

$$\Phi_{f(e)}(x) = \Phi_e(x)$$

for all x .

Proof. Let $g(z, x) = \Phi_{f(z)}(x)$, a partially computable function. By the recursion theorem, there is a number e such that

$$\Phi_e(x) = g(e, x) = \Phi_{f(e)}(x). \quad \blacksquare$$

Usually a number n is considered to be a fixed point of a function $f(x)$ if $f(n) = n$. Clearly there are computable functions that have no fixed point in this sense, e.g., $s(x)$. The fixed point theorem says that for every computable function $f(x)$, there is a number e of a program that *computes the same function as* the program with number $f(e)$.

For example, let $P(x)$ be a computable predicate, let $g(x)$ be a computable function, and let $\text{while}(n) = \#(\mathcal{Q}_n)$, where \mathcal{Q}_n is the program

```

 $X_2 \leftarrow n$ 
 $Y \leftarrow X$ 
[A] IF  $\sim P(Y)$  GOTO E
 $Y \leftarrow \Phi_{X_2}(g(Y))$ 

```

It should be clear that $\text{while}(x)$ is a computable, in fact primitive recursive, function, so by the fixed point theorem there is a number e such that

$$\Phi_e(x) = \Phi_{\text{while}(e)}(x).$$

It follows from the construction of $\text{while}(e)$ that

$$\Phi_e(x) = \Phi_{\text{while}(e)}(x) = \begin{cases} x & \text{if } \sim P(x) \\ \Phi_e(g(x)) & \text{otherwise.} \end{cases}$$

Moreover,

$$\Phi_e(g(x)) = \Phi_{\text{while}(e)}(g(x)) = \begin{cases} g(x) & \text{if } \sim P(g(x)) \\ \Phi_e(g(g(x))) & \text{otherwise,} \end{cases}$$

so

$$\Phi_e(x) = \Phi_{\text{while}(e)}(x) = \begin{cases} x & \text{if } \sim P(x) \\ g(x) & \text{if } P(x) \& \sim P(g(x)) \\ \Phi_e(g(g(x))) & \text{otherwise,} \end{cases}$$

and continuing in this fashion we get

$$\Phi_e(x) = \Phi_{\text{while}(e)}(x) = \begin{cases} x & \text{if } \sim P(x) \\ g(x) & \text{if } P(x) \& \sim P(g(x)) \\ g(g(x)) & \text{if } P(x) \& P(g(x)) \& \sim P(g(g(x))) \\ \vdots & \vdots \end{cases}$$

In other words, program e behaves like the pseudo-program

```

 $Y \leftarrow X$ 
WHILE  $P(Y)$  DO
 $Y \leftarrow g(Y)$ 
END

```

We end this discussion of the recursion theorem by giving another proof of Rice's theorem. Let Γ , $f(x)$, $g(x)$ be as in the statement of Theorem 7.1.

*Alternative Proof of Rice's Theorem.*² Suppose that R_Γ were computable. Let

$$P_\Gamma(t) = \begin{cases} 1 & \text{if } t \in R_\Gamma \\ 0 & \text{otherwise.} \end{cases}$$

That is, P_Γ is the characteristic function of R_Γ . Let

$$h(t, x) = \begin{cases} g(x) & \text{if } t \in R_\Gamma \\ f(x) & \text{otherwise.} \end{cases}$$

Then, since (as in the proof of Theorem 5.4, Chapter 3)

$$h(t, x) = g(x) \cdot P_\Gamma(t) + f(x) \cdot \alpha(P_\Gamma(t)),$$

$h(t, x)$ is partially computable. Thus, by the recursion theorem, there is a number e such that

$$\Phi_e(x) = h(e, x) = \begin{cases} g(x) & \text{if } \Phi_e \text{ belongs to } \Gamma \\ f(x) & \text{otherwise.} \end{cases}$$

² This elegant proof was called to our attention by John Case.

Does e belong to R_Γ ? Recalling that $f(x)$ belongs to Γ but $g(x)$ does not, we have

$$\begin{aligned} e \in R_\Gamma &\quad \text{implies} \quad \Phi_e(x) = g(x) \\ &\quad \text{implies} \quad \Phi_e \text{ is not in } \Gamma \\ &\quad \text{implies} \quad e \notin R_\Gamma. \end{aligned}$$

But likewise,

$$\begin{aligned} e \notin R_\Gamma &\quad \text{implies} \quad \Phi_e(x) = f(x) \\ &\quad \text{implies} \quad \Phi_e \text{ is in } \Gamma \\ &\quad \text{implies} \quad e \in R_\Gamma. \end{aligned}$$

This contradiction proves the theorem. ■

Exercises

1. Use the proof of Corollary 8.2 and the discussion preceding the proof of the recursion theorem to write a program \mathcal{P} such that $\psi_{\mathcal{P}}(x) = \#(\mathcal{P})$.
2. Let $A = \{x \in N \mid \Phi_x(x) \downarrow \text{ and } \Phi_x(x) > x\}$. Use the recursion theorem to show that A is not recursive.
3. Show that there is a number e such that $W_e = \{e\}$.
4. Show that there is a program \mathcal{P} such that $\psi_{\mathcal{P}}(x) \downarrow$ if and only if $x = \#(\mathcal{P})$.
5. (a) Show that there is a partially computable function f that satisfies the equations

$$\begin{aligned} f(x, 0) &= x + 2 \\ f(x, 1) &= 2 \cdot f(x, 2x) \\ f(x, 2t + 2) &= 3 \cdot f(x, 2t) \\ f(x, 2t + 3) &= 4 \cdot f(x, 2t + 1). \end{aligned}$$

What is $f(2, 5)$?

- (b) Prove that f is total.
- (c) Prove that f is unique. (That is, only one function satisfies the given equations.)
6. Give two distinct partially computable functions f, g that satisfy the equations

$$\begin{aligned} f(0) &= 2 & g(0) &= 2 \\ f(2t + 2) &= 3 \cdot f(2t) & g(2t + 2) &= 3 \cdot g(2t). \end{aligned}$$

For the specific functions f, g that you give, what are $f(1)$ and $g(1)$?

7. Let $f(x) = x + 1$. Use the proof of the fixed point theorem and the discussion preceding the proof of the recursion theorem to give a program \mathcal{P} such that $\Phi_{\#(\mathcal{P})}(x) = \Phi_{f(\#(\mathcal{P}))}(x)$. What unary function does \mathcal{P} compute?
8. Give a function $f(y)$ such that, for all y , $f(y) > y$ and $\Phi_y(x) = \Phi_{f(y)}(x)$.
9. Give a function $f(y)$ such that, for all y , if $\Phi_y(x) = \Phi_{f(y)}(x)$, then $\Phi_y(x)$ is not total.
10. Show that the function $\text{while}(x)$ defined following the fixed point theorem is primitive recursive. [Hint: Use the parameter theorem.]
11. (a) Prove that the recursion theorem can be strengthened to read:
There are infinitely many numbers e such that

$$\Phi_e^{(m)}(x_1, \dots, x_m) = g(e, x_1, \dots, x_m).$$

- (b) Prove that the fixed point theorem can be strengthened to read:
There are infinitely many numbers e such that

$$\Phi_{f(e)}(x) = \Phi_e(x).$$

12. Prove the following version of the recursion theorem: There is a primitive recursive function $\text{self}(x)$ such that for all z

$$\Phi_{\text{self}(z)}(x) = \Phi_z^{(2)}(\text{self}(z), x).$$

13. Prove the following version of the fixed point theorem: There is a primitive recursive function $\text{fix}(u)$ such that for all x, u ,

$$\Phi_{\text{fix}(u)}(x) = \Phi_{\Phi_u(\text{fix}(u))}(x).$$

- 14.* Let S be an acceptable programming system with universal functions $\Psi^{(m)}$. Prove the following: For every partially computable function $g(z, x_1, \dots, x_m)$ there is a number e such that

$$\Psi_e^{(m)}(x_1, \dots, x_m) = g(e, x_1, \dots, x_m).$$

That is, a version of the recursion theorem holds for S . [See Exercise 5.4 for the definition of acceptable programming systems.]

*9. A Computable Function That Is Not Primitive Recursive

In Chapter 3 we showed that all primitive recursive functions are computable, but we did not settle the question of whether all computable

functions are primitive recursive. We shall deal with this matter by showing how to obtain a function $h(x)$ that is computable but is not primitive recursive. Our method will be to construct a computable function $\phi(t, x)$ that enumerates all of the unary primitive recursive functions. That is, it will be the case that

1. for each fixed value $t = t_0$, the function $\phi(t_0, x)$ will be primitive recursive;
2. for each unary primitive recursive function $f(x)$, there will be a number t_0 such that $f(x) = \phi(t_0, x)$.

Once we have this function ϕ at our disposal, we can diagonalize, obtaining the unary computable function $\phi(x, x) + 1$ which must be different from all primitive recursive functions. (If it were primitive recursive, we would have

$$\phi(x, x) + 1 = \phi(t_0, x)$$

for some fixed t_0 , and setting $x = t_0$ would lead to a contradiction.)

We will obtain our enumerating function by giving a new characterization of the unary primitive recursive functions. However, we begin by showing how to reduce the number of parameters needed in the operation of primitive recursion which, as defined in Chapter 3 (Eq. (2.2)), proceeds from the total n -ary function f and the total $n + 2$ -ary function g to yield the $n + 1$ -ary function h such that

$$\begin{aligned} h(x_1, \dots, x_n, 0) &= f(x_1, \dots, x_n) \\ h(x_1, \dots, x_n, t + 1) &= g(t, h(x_1, \dots, x_n, t), x_1, \dots, x_n). \end{aligned}$$

If $n > 1$ we can reduce the number of parameters needed from n to $n - 1$ by using the pairing functions. That is, let

$$\begin{aligned} \tilde{f}(x_1, \dots, x_{n-1}) &= f(x_1, \dots, x_{n-2}, l(x_{n-1}), r(x_{n-1})), \\ \tilde{g}(t, u, x_1, \dots, x_{n-1}) &= g(t, u, x_1, \dots, x_{n-2}, l(x_{n-1}), r(x_{n-1})), \\ \tilde{h}(x_1, \dots, x_{n-1}, t) &= h(x_1, \dots, x_{n-2}, l(x_{n-1}), r(x_{n-1}), t). \end{aligned}$$

Then, we have

$$\begin{aligned} \tilde{h}(x_1, \dots, x_{n-1}, 0) &= \tilde{f}(x_1, \dots, x_{n-1}) \\ \tilde{h}(x_1, \dots, x_{n-1}, t + 1) &= \tilde{g}(t, \tilde{h}(x_1, \dots, x_{n-1}, t), x_1, \dots, x_{n-1}). \end{aligned}$$

Finally, we can retrieve the original function h from the equation

$$h(x_1, \dots, x_n, t) = \tilde{h}(x_1, \dots, x_{n-2}, \langle x_{n-1}, x_n \rangle, t).$$

By iterating this process we can reduce the number of parameters to 1, that is, to recursions of the form

$$\begin{aligned} h(x, 0) &= f(x) \\ h(x, t + 1) &= g(t, h(x, t), x) \end{aligned} \tag{9.1}$$

Recursions with no parameters, as in Eq. (2.1) in Chapter 3, can also readily be put into the form (9.1). Namely, to deal with

$$\begin{aligned} \psi(0) &= k \\ \psi(t + 1) &= \theta(t, \psi(t)), \end{aligned}$$

we set $f(x) = k$ (which can be obtained by k compositions with $s(x)$ beginning with $n(x)$) and

$$g(x_1, x_2, x_3) = \theta(u_1^3(x_1, x_2, x_3), u_2^3(x_1, x_2, x_3))$$

in the recursion (9.1). Then, $\psi(t) = h(x, t)$ for all x . In particular, $\psi(t) = h(u_1^1(t), u_1^1(t))$.

We can simplify recursions of the form (9.1) even further by using the pairing functions to combine arguments. Namely, we set

$$\tilde{h}(x, t) = \langle h(x, t), \langle x, t \rangle \rangle.$$

Then, we have

$$\begin{aligned} \tilde{h}(x, 0) &= \langle f(x), \langle x, 0 \rangle \rangle \\ \tilde{h}(x, t + 1) &= \langle h(x, t + 1), \langle x, t + 1 \rangle \rangle \\ &= \langle g(t, h(x, t), x), \langle x, t + 1 \rangle \rangle \\ &= \tilde{g}(\tilde{h}(x, t)), \end{aligned}$$

where

$$\tilde{g}(u) = \langle g(r(r(u)), l(u), l(r(u)), \langle l(r(u)), r(r(u)) + 1 \rangle) \rangle.$$

Once again, the original function h can be retrieved from \tilde{h} ; we can use the equation

$$h(x, t) = l(\tilde{h}(x, t)).$$

Now this reduction in the complexity of recursions was only possible using the pairing functions. Nevertheless, we can use it to get a simplified characterization of the class of primitive recursive functions by adding the pairing functions to our initial functions. We may state the result as a theorem.

Theorem 9.1. The primitive recursive functions are precisely the functions obtainable from the initial functions

$$s(x), n(x), l(z), r(z), \langle x, y \rangle \text{ and } u_i^n, \quad 1 \leq i \leq n$$

using the operations of composition and primitive recursion of the particular form

$$\begin{aligned} h(x, 0) &= f(x) \\ h(x, t + 1) &= g(h(x, t)). \end{aligned}$$

The promised characterization of the unary primitive recursive functions is as follows.

Theorem 9.2. The unary primitive recursive functions are precisely those obtained from the initial functions $s(x) = x + 1$, $n(x) = 0$, $l(x)$, $r(x)$ by applying the following three operations on unary functions:

1. to go from $f(x)$ and $g(x)$ to $f(g(x))$;
2. to go from $f(x)$ and $g(x)$ to $\langle f(x), g(x) \rangle$;
3. to go from $f(x)$ and $g(x)$ to the function defined by the recursion

$$h(0) = 0$$

$$h(t + 1) = \begin{cases} f\left(\frac{t}{2}\right) & \text{if } t + 1 \text{ is odd,} \\ g\left(h\left(\frac{t+1}{2}\right)\right) & \text{if } t + 1 \text{ is even.} \end{cases}$$

Proof. Let us write **PR** for the set of all functions obtained from the initial functions listed in the theorem using operations 1 through 3. We will show that **PR** is precisely the set of unary primitive recursive functions.

To see that all the functions in **PR** are primitive recursive, it is necessary only to consider operation 3. That is, we need to show that if f and g are primitive recursive, and h is obtained using operation 3, then h is also primitive recursive. What is different about operation 3 is that $h(t + 1)$ is computed, not from $h(t)$ but rather from $h(t/2)$ or $h((t + 1)/2)$, depending on whether t is even or odd. To deal with this we make use of Gödel numbering, setting

$$\tilde{h}(0) = 0,$$

$$\tilde{h}(n) = [h(0), \dots, h(n - 1)] \text{ if } n > 0.$$

We will show that \tilde{h} is primitive recursive and then conclude that the same is true of h by using the equation³

$$h(n) = (\tilde{h}(n + 1))_{n+1}.$$

Then (recalling that p_n is the n th prime number) we have

$$\begin{aligned}\tilde{h}(n + 1) &= \tilde{h}(n) \cdot p_{n+1}^{h(n)} \\ &= \begin{cases} \tilde{h}(n) \cdot p_{n+1}^{f(\lfloor n/2 \rfloor)} & \text{if } n \text{ is odd,} \\ \tilde{h}(n) \cdot p_{n+1}^{g((\tilde{h}(n))_{\lfloor n/2 \rfloor})} & \text{if } n \text{ is even.} \end{cases}\end{aligned}$$

Here, we have used $\lfloor n/2 \rfloor$ because it gives the correct value whether n is even or odd and because we know from Chapter 3 that it is primitive recursive.

Next we will show that every unary primitive recursive function belongs to **PR**. For this purpose we will call a function $g(x_1, \dots, x_n)$ *satisfactory* if it has the property that for any unary functions $h_1(t), \dots, h_n(t)$ that belong to **PR**, the function $g(h_1(t), \dots, h_n(t))$ also belongs to **PR**. Note that a *unary* function $g(t)$ that is satisfactory must belong to **PR** because $g(t) = g(u_1^1(t))$ and $u_1^1(t) = \langle l(t), r(t) \rangle$ belongs to **PR**. Thus, we can obtain our desired result by proving that all primitive recursive functions are satisfactory.⁴

We shall use the characterization of the primitive recursive functions of Theorem 9.1. Among the initial functions, we need consider only the pairing function $\langle x_1, x_2 \rangle$ and the projection functions u_i^n where $1 \leq i \leq n$. If $h_1(t)$ and $h_2(t)$ are in **PR**, then using operation 2 in the definition of **PR**, we see that $\langle h_1(t), h_2(t) \rangle$ is also in **PR**. Hence, $\langle x_1, x_2 \rangle$ is satisfactory. And evidently, if $h_1(t), \dots, h_n(t)$ belong to **PR**, then $u_i^n(h_1(t), \dots, h_n(t))$, which is simply equal to $h_i(t)$, certainly belongs to **PR**, so u_i^n is satisfactory.

To deal with composition, let

$$h(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$$

where g_1, \dots, g_k and f are satisfactory. Let $h_1(t), \dots, h_n(t)$ be given functions that belong to **PR**. Then, setting

$$\tilde{g}_i(t) = g_i(h_1(t), \dots, h_n(t))$$

³ This is a general technique for dealing with recursive definitions for a given value in terms of smaller values, so-called *course-of-value recursions*. See Exercise 8.5 in Chapter 3.

⁴ This is an example of what was called an *induction loading device* in Chapter 1.

for $1 \leq i \leq k$ we see that each \tilde{g}_i belongs to **PR**. Hence

$$h(h_1(t), \dots, h_n(t)) = f(\tilde{g}_1(t), \dots, \tilde{g}_n(t))$$

belongs to **PR**, and so, h is satisfactory.

Finally, let

$$\begin{aligned} h(x, 0) &= f(x) \\ h(x, t + 1) &= g(h(x, t)) \end{aligned}$$

where f and g are satisfactory. Let $\psi(0) = 0$ and let $\psi(t + 1) = h(r(t), l(t))$. Recalling that

$$\langle a, b \rangle = 2^a(2b + 1) - 1,$$

we consider two cases according to whether $t + 1 = 2^a(2b + 1)$ is even or odd. If $t + 1$ is even, then $a > 0$ and

$$\begin{aligned} \psi(t + 1) &= h(b, a) \\ &= g(h(b, a - 1)) \\ &= g(\psi(2^{a-1}(2b + 1))) \\ &= g(\psi((t + 1)/2)). \end{aligned}$$

On the other hand, if $t + 1$ is odd, then $a = 0$ and

$$\begin{aligned} \psi(t + 1) &= h(b, 0) \\ &= f(b) \\ &= f(t/2). \end{aligned}$$

In other words,

$$\begin{aligned} \psi(0) &= 0 \\ \psi(t + 1) &= \begin{cases} f\left(\frac{t}{2}\right) & \text{if } t + 1 \text{ is odd,} \\ g\left(\psi\left(\frac{t+1}{2}\right)\right) & \text{if } t + 1 \text{ is even.} \end{cases} \end{aligned}$$

Now f and g are satisfactory, and, being unary, they are therefore in **PR**. Since ψ is obtained from f and g using operation 3, ψ also belongs to **PR**. To retrieve h from ψ we can use $h(x, y) = \psi(\langle x, y \rangle + 1)$. So,

$$h(h_1(t), h_2(t)) = \psi(s(\langle h_1(t), h_2(t) \rangle))$$

from which we see that if h_1 and h_2 both belong to **PR**, then so does $h(h_1(t), h_2(t))$. Hence h is satisfactory. ■

Now we are ready to define the function $\phi(t, x)$, which we shall also write as $\phi_t(x)$, that will enumerate the unary primitive recursive functions:

$$\phi_t(x) = \begin{cases} x + 1 & \text{if } t = 0 \\ 0 & \text{if } t = 1 \\ l(x) & \text{if } t = 2 \\ r(x) & \text{if } t = 3 \\ \phi_{l(n)}(\phi_{r(n)}(x)) & \text{if } t = 3n + 1, n > 0 \\ \langle \phi_{l(n)}(x), \phi_{r(n)}(x) \rangle & \text{if } t = 3n + 2, n > 0 \\ 0 & \text{if } t = 3n + 3, n > 0 \text{ and } x = 0 \\ \phi_{l(n)}((x - 1)/2) & \text{if } t = 3n + 3, n > 0 \text{ and } x \text{ is odd} \\ \phi_{r(n)}(\phi_t(x/2)) & \text{if } t = 3n + 3, n > 0 \text{ and } x \text{ is even} \end{cases}$$

Here $\phi_0(x), \phi_1(x), \phi_2(x), \phi_3(x)$ are the four initial functions. For $t > 3$, t is represented as $3n + i$ where $n > 0$ and $i = 1, 2$ or 3 ; the three operations of Theorem 9.2 are then dealt with for values of t with the corresponding value of i . The pairing functions are used to guarantee all functions obtained for any value of t are eventually used in applying each of the operations. It should be clear from the definition that $\phi(t, x)$ is a total function and that it does enumerate all the unary primitive recursive functions. Although it is pretty clear that the definition provides an algorithm for computing the values of ϕ for any given inputs, for a rigorous proof more is needed. Fortunately, the recursion theorem makes it easy to provide such a proof. Namely, we set

$g(z, t, x)$

$$= \begin{cases} x + 1 & \text{if } t = 0 \\ 0 & \text{if } t = 1 \\ l(x) & \text{if } t = 2 \\ r(x) & \text{if } t = 3 \\ \Phi_z^{(2)}(l(n), \Phi_z^{(2)}(r(n), x)) & \text{if } t = 3n + 1, n > 0 \\ \langle \Phi_z^{(2)}(l(n), x), \Phi_z^{(2)}(r(n), x) \rangle & \text{if } t = 3n + 2, n > 0 \\ 0 & \text{if } t = 3n + 3, n > 0 \text{ and } x = 0 \\ \Phi_z^{(2)}(l(n), [x/2]) & \text{if } t = 3n + 3, n > 0 \text{ and } x \text{ is odd} \\ \Phi_z^{(2)}(r(n), \Phi_z^{(2)}(t, [x/2])) & \text{if } t = 3n + 3, n > 0 \text{ and } x \text{ is even} \end{cases}$$

Then, $g(z, t, x)$ is partially computable, and by the recursion theorem, there is a number e such that

$$g(e, t, x) = \Phi_e^{(2)}(t, x).$$

Then, since $g(e, t, x)$ satisfies the definition of $\phi(t, x)$ and that definition determines ϕ uniquely as a total function, we must have

$$\phi(t, x) = g(e, t, x),$$

so that ϕ is computable.

The discussion at the beginning of this section now applies and we have our desired result.

Theorem 9.3. The function $\phi(x, x) + 1$ is a computable function that is not primitive recursive.

Exercises

1. Show that $\phi(t, x)$ is not primitive recursive.
2. Give a direct proof that $\phi(t, x)$ is computable by showing how to obtain an \mathcal{S} program that computes ϕ . [Hint: Use the pairing functions to construct a stack for handling recursions.]

5

Calculations on Strings

1. Numerical Representation of Strings

So far we have been dealing exclusively with computations on numbers. Now we want to extend our point of view to include computations on strings of symbols on a given alphabet. In order to extend computability theory to strings on an alphabet A , we wish to associate numbers with elements of A^* in a one-one manner. We now describe one convenient way of doing this: Let A be some given alphabet. Since A is a set, there is no order implied among the symbols. However, *we will assume in this chapter that the elements of A have been placed in some definite order*. In particular, when we write $A = \{s_1, \dots, s_n\}$, we think of the sequence s_1, \dots, s_n as corresponding to this given order. Now, let $w = s_{i_k} s_{i_{k-1}} \cdots s_{i_1} s_{i_0}$. Then we associate with w the integer

$$x = i_k \cdot n^k + i_{k-1} \cdot n^{k-1} + \cdots + i_1 \cdot n + i_0. \quad (1.1)$$

With $w = 0$, we associate the number 0. (It is for this reason that we use the same symbol for both.) For example, let A consist of the symbols a, b, c given in the order shown, and let $w = baacb$. Then, the corresponding integer is

$$x = 2 \cdot 3^4 + 1 \cdot 3^3 + 1 \cdot 3^2 + 3 \cdot 3^1 + 2 = 209.$$

In order to see that the representation (1.1) is unique, we show how to retrieve the subscripts i_0, i_1, \dots, i_k from x assuming that $x \neq 0$. We define the primitive recursive functions:

$$R^+(x, y) = \begin{cases} R(x, y) & \text{if } \sim(y|x) \\ y & \text{otherwise,} \end{cases}$$

$$Q^+(x, y) = \begin{cases} \lfloor x/y \rfloor & \text{if } \sim(y|x) \\ \lfloor x/y \rfloor - 1 & \text{otherwise,} \end{cases}$$

where the functions $R(x, y)$ and $\lfloor x/y \rfloor$ are as defined in Chapter 3, Section 7. Then, as we shall easily show, for $y \neq 0$,

$$\frac{x}{y} = Q^+(x, y) + \frac{R^+(x, y)}{y}, \quad 0 < R^+(x, y) \leq y.$$

This equation expresses ordinary division with quotient and remainder:

$$\frac{x}{y} = \lfloor x/y \rfloor + \frac{R(x, y)}{y},$$

as long as y is not a divisor of x . If y is a divisor of x we have

$$\frac{x}{y} = \lfloor x/y \rfloor = (\lfloor x/y \rfloor - 1) + \frac{y}{y} = Q^+(x, y) + \frac{R^+(x, y)}{y}.$$

Thus, what we are doing differs from ordinary division with remainders in that “remainders” are permitted to take on values between 1 and y rather than between 0 and $y - 1$.

Now, let us set

$$u_0 = x, \quad u_{m+1} = Q^+(u_m, n). \quad (1.2)$$

Thus, by (1.1)

$$\begin{aligned} u_0 &= i_k \cdot n^k + i_{k-1} \cdot n^{k-1} + \cdots + i_1 \cdot n + i_0, \\ u_1 &= i_k \cdot n^{k-1} + i_{k-1} \cdot n^{k-2} + \cdots + i_1, \\ &\vdots \\ u_k &= i_k. \end{aligned} \quad (1.3)$$

Therefore,

$$i_m = R^+(u_m, n), \quad m = 0, 1, \dots, k. \quad (1.4)$$

Hence, for any number x satisfying (1.1), the string w can be retrieved. It is worth noting that this can be accomplished using primitive recursive functions. If we write

$$\begin{aligned} g(0, n, x) &= x, \\ g(m + 1, n, x) &= Q^+(g(m, n, x), n), \end{aligned}$$

then

$$g(m, n, x) = u_m \quad (1.5)$$

as defined by (1.2), where, of course, g is primitive recursive. Moreover, if we let $h(m, n, x) = R^+(g(m, n, x), n)$, then h is also primitive recursive, and by (1.4)

$$i_m = h(m, n, x), \quad m = 0, 1, \dots, k. \quad (1.6)$$

This method of representing strings by numbers is clearly related to the usual base n notation for numbers. To explore the connection, it is instructive to consider the alphabet

$$D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, X\}$$

in the order shown. Then the number associated with the *string* 45 is

$$4 \cdot 10 + 5 = 45.$$

On the other hand, the number associated with $2X$ is

$$2 \cdot 10 + 10 = 30.$$

(Perhaps we should read $2X$ as “twenty-ten”!) Clearly a string on D that does not include X is simply the usual decimal notation for the number it represents. It is numbers whose decimal representation includes a 0 which now require an X .

Thus, in the general case of an alphabet A consisting of s_1, \dots, s_n , ordered as shown, we see that we are simply using a base n representation in which the “digits” range from 1 to n instead of the usual 0 to $n - 1$. We are proceeding in this manner simply to avoid the lack of uniqueness of the usual base n representation:

$$79 = 079 = 0079 = 00079 = \text{etc.}$$

This lack of uniqueness is of course caused by the fact that leading zeros do not change the number being represented.

It is interesting to observe that the rules of elementary arithmetic (including the use of “carries”) work perfectly well with our representation. Here are a few examples:

$$\begin{array}{r} 1\ 7 \\ + 1\ X\ 3 \\ \hline 2\ 1\ X \end{array} \quad \text{which corresponds to} \quad \begin{array}{r} 17 \\ + 203 \\ \hline 220 \end{array}$$

$$\begin{array}{r} 2\ 9 \\ - 1\ X \\ \hline 9 \end{array} \quad \text{which corresponds to} \quad \begin{array}{r} 29 \\ - 20 \\ \hline 9 \end{array}$$

$$\begin{array}{r} X\ 5 \\ \times 2\ X \\ \hline X\ 4\ X \\ 1\ X\ X \\ \hline 3\ 1\ 4\ X \end{array} \quad \text{which corresponds to} \quad \begin{array}{r} 105 \\ \times 30 \\ \hline 3150 \end{array}$$

(Incidentally, this shows that the common belief that the modern rules of calculation required the introduction of a digit for 0 is unjustified.) Note in particular the following examples of adding 1:

$$\begin{array}{r} X\ 1 \\ + 1 \\ \hline X\ 2 \end{array} \quad \begin{array}{r} 3\ X \\ + 1 \\ \hline 4\ 1 \end{array} \quad \begin{array}{r} 3\ X\ X \\ + 1 \\ \hline 4\ 1\ 1 \end{array} \quad \begin{array}{r} 7\ 3\ X\ X \\ + 1 \\ \hline 7\ 4\ 1\ 1 \end{array} \quad \begin{array}{r} 4\ 9 \\ + 1 \\ \hline 4\ X \end{array}$$

Adding 1 to X gives a result of 1 with a carry of 1. If the string ends in more than one X , the carry propagates. Subtracting 1 is similar, with a propagating carry produced by a string ending in 1:

$$\begin{array}{r} 1\ X \\ - 1 \\ \hline 1\ 9 \end{array} \quad \begin{array}{r} X\ 1 \\ - 1 \\ \hline 9\ X \end{array} \quad \begin{array}{r} 7\ 1\ 1 \\ - 1 \\ \hline 6\ X\ X \end{array}$$

Now we return to the general case. Given the alphabet A consisting of s_1, \dots, s_n in the order shown, the string $w = s_{i_k}s_{i_{k-1}} \dots s_{i_1}s_{i_0}$ is called the *base n notation for the number x* defined by (1.1). (0 is the base n notation for the null string 0 for every n .) Thus when n is fixed we can regard a partial function of one or more variables on A^* as a function of the corresponding numbers. (That is, the numbers are just those which the given strings represent in base n notation.) It now makes perfect sense to speak of an m -ary partial function on A^* with values in A^* as being *partially computable*, or when it is total, as being *computable*. Similarly we

can say that an m -ary function on A^* is *primitive recursive*. Note that for any alphabet $A = \{s_1, \dots, s_n\}$ with the symbols ordered as shown, s_1 denotes 1 in base n . Thus an m -ary predicate on A^* is simply a total m -ary function on A^* all of whose values are either s_1 or 0. And it now makes sense as well to speak of an m -ary predicate on A^* as being computable.

As was stated in Chapter 1, for a given alphabet A , any subset of A^* is called a *language* on A . Once again, by associating with the elements of A^* the corresponding numbers, we can speak of a language on A as being r.e., or recursive, or primitive recursive.

It is important to observe that whereas the usual base n notation using a 0 digit works only for $n \geq 2$, the representation (1.1) is valid even for $n = 1$. For an alphabet consisting of the single symbol 1, the string $1^{[x]}$ of length x is the base 1 notation for the number $\sum_{i=0}^{x-1} 1 \cdot (1)^i = \sum_{i=0}^{x-1} 1 = x$. That is, the base 1 (or *unary*) representation of the number x is simply a string of ones of length x .

In thinking of numbers (that is, elements of N) as inputs to and outputs from programs written in our language \mathcal{S} , no particular representation of these numbers was specified or required. Numbers occur in the theory as purely abstract entities, just as they do in ordinary mathematics. However, when we wish to refer to particular numbers, we do so in the manner familiar to all of us, by writing their decimal representations. These representations are, of course, really strings on the alphabet that consists of the decimal digits:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

But it is essential to avoid confusing such strings with the numbers they represent. For this reason, for the remainder of this chapter we shall avoid the use of decimal digits as symbols in our alphabets. Thus, a string of decimal digits will *always be meant to refer to a number*.

Now, let A be some fixed alphabet containing exactly n symbols, say $A = \{s_1, s_2, \dots, s_n\}$. For each $m \geq 1$, we define $\text{CONCAT}_n^{(m)}$ as follows:

$$\begin{aligned}\text{CONCAT}_n^{(1)}(u) &= u, \\ \text{CONCAT}_n^{(m+1)}(u_1, \dots, u_m, u_{m+1}) &= zu_{m+1},\end{aligned}\tag{1.7}$$

where

$$z = \text{CONCAT}_n^{(m)}(u_1, \dots, u_m).$$

Thus, for given strings $u_1, \dots, u_m \in A^*$, $\text{CONCAT}_n^{(m)}(u_1, \dots, u_m)$ is simply the string obtained by placing the strings u_1, \dots, u_m one after the other,

or, as is usually said, by *concatenating* them. We will usually omit the superscript, so that, for example we may write

$$\text{CONCAT}_2(s_2s_1, s_1s_1s_2) = s_2s_1s_1s_1s_2.$$

Likewise,

$$\text{CONCAT}_6(s_2s_1, s_1s_1s_2) = s_2s_1s_1s_1s_2.$$

However, the string s_2s_1 represents the number 5 in base 2 and the number 13 in base 6. Also, the string $s_1s_1s_2$ represents the number 8 in base 2 and the number 44 in base 6. Finally, the string $s_2s_1s_1s_1s_2$ represents 48 in base 2 and 2852 in base 6. If we wish to think of CONCAT as defining functions on N (as will be necessary, for example, in showing that the functions (1.7) are primitive recursive), then the example we have been considering becomes

$$\text{CONCAT}_2(5, 8) = 48 \quad \text{and} \quad \text{CONCAT}_6(13, 44) = 2852.$$

The same example in base 10 gives

$$\text{CONCAT}_{10}(21, 112) = 21112.$$

Bearing this discussion in mind, we now proceed to give a list of primitive recursive functions (on A^* or N , depending on one's point of view) that we will need later.

1. $f(u) = |u|$. This “length” function is most naturally understood as being defined on A^* and taking values in N . For each x , the number $\sum_{j=0}^x n^j$ has the base n representation $s_1^{[x+1]}$; hence this number is the smallest number whose base n representation contains $x + 1$ symbols. Thus,

$$|u| = \min_{x \leq u} \left[\sum_{j=0}^x n^j > u \right].$$

2. $g(u, v) = \text{CONCAT}_n(u, v)$. The primitive recursiveness of this function follows from the equation

$$\text{CONCAT}_n(u, v) = u \cdot n^{|v|} + v.$$

3. $\text{CONCAT}_n^{(m)}(u_1, \dots, u_m)$, as defined in (1.7), is primitive recursive for each $m, n \geq 1$. This follows at once from the previous example using composition.
4. $\text{RTEND}_n(w) = h(0, n, w)$, where h is as in (1.6). As a function of A^* , RTEND_n gives the rightmost symbol of a given word, as is clear from (1.3) and (1.6).

5. $\text{LTEND}_n(w) = h(|w| - 1, n, w)$. LTEND_n gives the leftmost symbol of a given nonempty word.
6. $\text{RTRUNC}_n(w) = g(1, n, w)$. RTRUNC_n gives the result of removing the rightmost symbol from a given nonempty word, as is clear from (1.3) and (1.5). When we can omit reference to the base n , we often write w^- for $\text{RTRUNC}_n(w)$. Note that $0^- = 0$.
7. $\text{LTRUNC}_n(w) = w \dot{-} (\text{LTEND}_n(w) \cdot n^{|w|-1})$. In the notation of (1.3), for a given nonempty word w , $\text{LTRUNC}_n(w) = w - i_k \cdot n^k$, i.e., $\text{LTRUNC}_n(w)$ is the result of removing the leftmost symbol from w .

We will now use the list of primitive recursive functions that we have just given to prove the computability of a pair of functions that can be used in changing base. Thus, let $1 \leq n < l$. Let $A \subset \tilde{A}$, where A is an alphabet of n symbols and \tilde{A} is an alphabet of l symbols. Thus a string that belongs to A^* also belongs to \tilde{A}^* . For any $x \in N$, let w be the word in A^* that represents x in base n . Then, we write $\text{UPCHANGE}_{n,l}(x)$ for the number which w represents in base l . For example, referring to our previous example, we have $\text{UPCHANGE}_{2,6}(5) = 13$, $\text{UPCHANGE}_{2,6}(8) = 44$, $\text{UPCHANGE}_{2,6}(48) = 2852$. Also $\text{UPCHANGE}_{2,10}(5) = 21$ and $\text{UPCHANGE}_{6,10}(13) = 21$.

Next, for $x \in N$, let w be the string in \tilde{A}^* which represents x in base l , and let w' be obtained from w by crossing out all of the symbols that belong to $\tilde{A} - A$. Then, $w' \in A^*$, and we write $\text{DOWNCHANGE}_{n,l}(x)$ for the number which w' represents in base n . For example, the string $s_2 s_6 s_1$ represents the number 109 in base 6. To obtain $\text{DOWNCHANGE}_{2,6}(109)$ we cross out the s_6 , obtaining the string $s_2 s_1$, which represents 5 in base 2; thus $\text{DOWNCHANGE}_{2,6}(10) = 5$.

Although $\text{UPCHANGE}_{n,l}$ and $\text{DOWNCHANGE}_{n,l}$ are actually primitive recursive functions, we will content ourselves with proving that they are computable:

Theorem 1.1. Let $0 < n < l$. Then the functions $\text{UPCHANGE}_{n,l}$ and $\text{DOWNCHANGE}_{n,l}$ are computable.

Proof. We begin with $\text{UPCHANGE}_{n,l}$. We write a program which extracts the successive symbols of the word that the given number represents in base n and uses them in computing the number that the given word represents in base l :

```
[A]      IF X = 0 GOTO E
          Z ← LTENDn(X)
          X ← LTRUNCn(X)
          Y ← l · Y + Z
          GOTO A
```

$\text{DOWNCHANGE}_{n,l}$ is handled similarly. Our program will extract the successive symbols of the word that the given number represents in base l . However, these symbols will only be used if they belong to the smaller alphabet, i.e., if as numbers they are $\leq n$:

```
[A]      IF  $X = 0$  GOTO E
          Z  $\leftarrow \text{LTEND}_l(X)$ 
          X  $\leftarrow \text{LTRUNC}_l(X)$ 
          IF  $Z > n$  GOTO A
          Y  $\leftarrow n \cdot Y + Z$ 
          GOTO A
```

■

Exercises

1. (a) Write the numbers 40 and 12 in base 3 notation using the “digits” $\{1, 2, 3\}$.
 (b) Work out the multiplication $40 \cdot 12 = 480$ in base 3.
 (c) Compute $\text{CONCAT}_n(12, 15)$ for $n = 3, 5$, and 10. Why is no calculation required in the last case?
 (d) Compute the following: $\text{UPCHANGE}_{3,7}(15)$, $\text{UPCHANGE}_{2,7}(15)$, $\text{UPCHANGE}_{2,10}(15)$, $\text{DOWNCHANGE}_{3,7}(15)$, $\text{DOWNCHANGE}_{2,7}(15)$, $\text{DOWNCHANGE}_{2,10}(20)$.
2. Compute each of the following for $n = 3$.
 - $\text{CONCAT}_n^{(2)}(17, 32)$.
 - $\text{CONCAT}_n^{(3)}(17, 32, 11)$.
 - $\text{RTEND}_n(23)$.
 - $\text{LTEND}_n(29)$.
 - $\text{RTRUNC}_n(19)$.
 - $\text{LTRUNC}_n(18)$.
3. Do the previous exercise for $n = 4$.
4. Show that the function f whose value is the string formed of the symbols occurring in the odd-numbered places in the input [i.e., $f(a_1a_2a_3 \cdots a_n) = a_1a_3 \cdots$] is computable.
5. Let $A = \{s_1, \dots, s_n\}$, and let $P(x)$ be the predicate on N which is true just when the string in A^* that represents x has an even number of symbols. Show that $P(x)$ is primitive recursive.

6. If $u \neq 0$, let $\#(u, v)$ be the number of occurrences of u as a part of v [e.g., $\#(bab, ababab) = 2$]. Also, let $\#(0, v) = 0$. Prove that $\#(u, v)$ is primitive recursive.
7. Show that $\text{UPCHANGE}_{n,l}$ and $\text{DOWNCHANGE}_{n,l}$ are primitive recursive.
8. Show that when $|u|$ is calculated with respect to base n notation, $|u| \leq \lfloor \log_n u \rfloor + 1$ for all $u \in N$.

2. A Programming Language for String Computations

From the point of view of string computations, the language \mathcal{S} seems quite artificial. For example, the instruction

$$V \leftarrow V + 1$$

which is so basic for integers, seems entirely unnatural as a basic instruction for string calculations. Thus, for the alphabet $\{a, b, c\}$, applying this instruction to $bacc$ produces $bbaa$ because a carry is propagated. (This will perhaps seem more evident if, momentarily ignoring our promise to avoid the decimal digits as symbols in our alphabets, we use the alphabet $\{1, 2, 3\}$ and write

$$2133 + 1 = 2211.)$$

We are now going to introduce, for each $n > 0$, a programming language \mathcal{S}_n , which is specifically designed for string calculations on an alphabet of n symbols. The languages \mathcal{S}_n will be supplied with the same input, output, and local variables as \mathcal{S} , except that we now think of them as having values in the set A^* , where A is an n symbol alphabet. Variables not otherwise initialized are to be initialized to 0. We use the same symbols as labels in \mathcal{S}_n as in \mathcal{S} and the same conventions regarding their use. The instruction types are shown in Table 2.1.

The formal rules of syntax in \mathcal{S}_n are entirely analogous to those for \mathcal{S} , and we omit them. Similarly, we use macro expansions quite freely. An m -ary partial function on A^* which is computed by a program in \mathcal{S}_n is said to be *partially computable* in \mathcal{S}_n . If the function is total and partially computable in \mathcal{S}_n , it is called *computable* in \mathcal{S}_n .

Although the instructions of \mathcal{S}_n refer to strings, we can just as well think of them as referring to the numbers that the corresponding strings represent in base n . For example, the numerical effect of the instruction

$$X \leftarrow s_i X$$

Table 2.1

Instruction	Interpretation
$V \leftarrow \sigma V$ for each symbol σ in the alphabet A	Place the symbol σ to the left of the string which is the value of V .
$V \leftarrow V^-$	Delete the final symbol of the string which is the value of V . If the value of V is 0, leave it unchanged.
If V ENDS σ GOTO L for each symbol σ in the alphabet A and each label L	If the value of V ends in the symbol σ , execute next the first instruction labeled L ; otherwise proceed to the next instruction.

in the n symbol alphabet $\{s_1, \dots, s_n\}$ ordered as shown is to replace the numerical value x by $i \cdot n^{|x|} + x$. Just as the instructions of \mathcal{S} are natural as basic numerical operations, but complex as string operations, so the instructions of \mathcal{S}_n are natural as basic string operations, but complex as numerical operations.

We now give some macros for use in \mathcal{S}_n with the corresponding expansions.

1. The macro IF $V \neq 0$ GOTO L has the expansion

IF V ENDS s_1 GOTO L

IF V ENDS s_2 GOTO L

⋮

IF V ENDS s_n GOTO L

2. The macro $V \leftarrow 0$ has the expansion

$[A] \quad V \leftarrow V^-$

IF $V \neq 0$ GOTO A

3. The macro GOTO L has the expansion

$Z \leftarrow 0$

$Z \leftarrow s_1 Z$

IF Z ENDS s_1 GOTO L

4. The macro $V' \leftarrow V$ has the expansion shown in Fig. 2.1.

The macro expansion of $V' \leftarrow V$ in \mathcal{S}_n is quite similar to that in \mathcal{S} .

$Z \leftarrow 0$ $V' \leftarrow 0$ [A] IF V ENDS s_1 GOTO B_1 IF V ENDS s_2 GOTO B_2 \vdots IF V ENDS s_n GOTO B_n GOTO C	$V \leftarrow V^-$ $V' \leftarrow s_i V'$ $Z \leftarrow s_i Z$ [B_i] $\left. \begin{array}{l} \\ \\ \end{array} \right\} i = 1, 2, \dots, n$ GOTO A
[C] IF Z ENDS s_1 GOTO D_1 IF Z ENDS s_2 GOTO D_2 \vdots IF Z ENDS s_n GOTO D_n GOTO E	$V \leftarrow V^-$ $V \leftarrow s_i V$ [D_i] $\left. \begin{array}{l} \\ \\ \end{array} \right\} i = 1, 2, \dots, n$ GOTO C

Figure 2.1. Macro expansion of $V' \leftarrow V$ in \mathcal{S}_n .

The block of instructions

```

IF  $V$  ENDS  $s_1$  GOTO  $B_1$ 
IF  $V$  ENDS  $s_2$  GOTO  $B_2$ 
 $\vdots$ 
IF  $V$  ENDS  $s_n$  GOTO  $B_n$ 

```

is usually written simply

$$\text{IF } V \text{ ENDS } s_i \text{ GOTO } B_i \quad (1 \leq i \leq n)$$

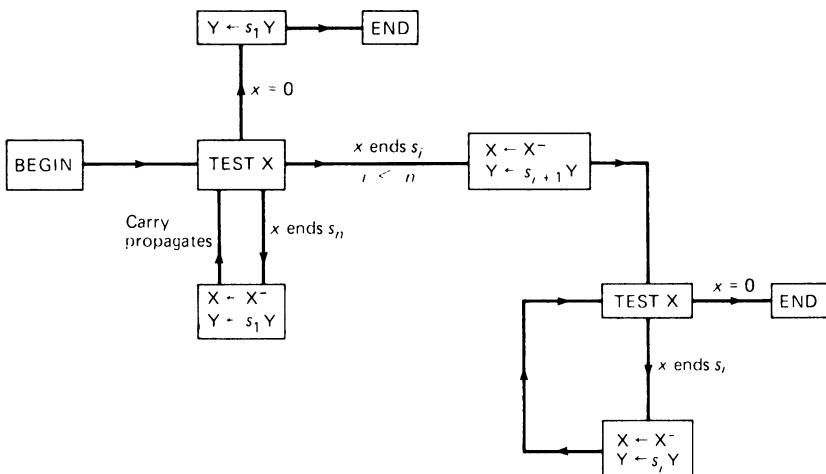
Such a block of instructions is referred to as a *filter* for obvious reasons. Note that at the point in the computation when the first “GOTO C ” is executed, V' and Z will both have the original value of V , whereas V will have the value 0. On exiting, Z has the value 0, while V' retains the original value of V and V has been restored to its original value.

If $f(x_1, \dots, x_m)$ is any function that is partially computable in \mathcal{S}_n , we permit the use in \mathcal{S}_n of macros of the form

$$V \leftarrow f(V_1, \dots, V_m)$$

The corresponding expansions are carried out in a manner entirely analogous to that discussed in Chapter 2, Section 5.

We conclude this section with two examples of functions that are computable in \mathcal{S}_n for every n . The general results in the next section will

Figure 2.2. Flow chart for computing $x + 1$ in \mathcal{S}_n .

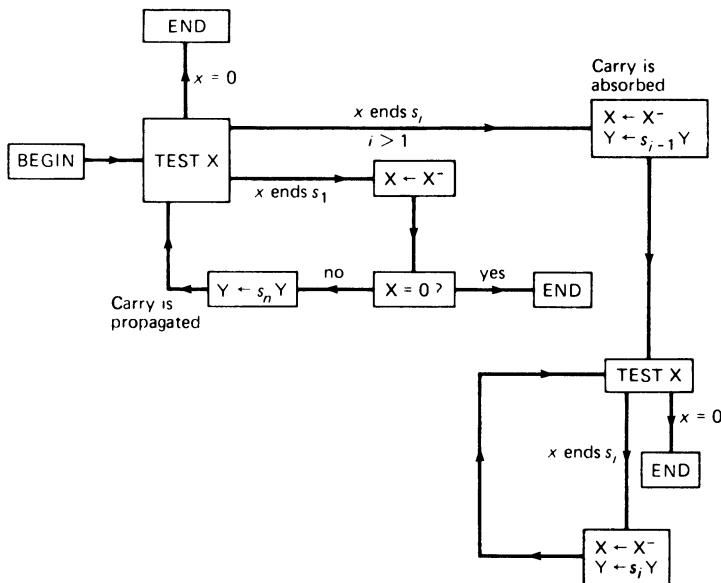
make it clear that these two examples are the only bit of programming in \mathcal{S}_n that we shall need to carry out explicitly.

We want to show that the function $x + 1$ is computable in \mathcal{S}_n . We let our alphabet consist of the symbols s_1, s_2, \dots, s_n ordered as shown. The desired program is exhibited in Fig. 2.3; a flow chart that shows how the program works is shown in Fig. 2.2.

Our final example is a program that computes $x \dot{-} 1$ base n . A flow chart is given in Fig. 2.4 and the actual program in \mathcal{S}_n is exhibited in Fig. 2.5. The reader should check both of these programs with some examples.

[B]	IF X ENDS s_i GOTO A_i	$(1 \leq i \leq n)$
	$Y \leftarrow s_1 Y$	
	GOTO E	
[A _i]	$X \leftarrow X^-$	
	$Y \leftarrow s_{i+1} Y$	$\left. \right\} 1 \leq i < n$
	GOTO C	
[A _n]	$X \leftarrow X^-$	
	$Y \leftarrow s_1 Y$	
	GOTO B	
[C]	IF X ENDS s_i GOTO D_i	$(1 \leq i \leq n)$
	GOTO E	
[D _i]	$X \leftarrow X^-$	
	$Y \leftarrow s_i Y$	$\left. \right\} 1 \leq i \leq n$
	GOTO C	

Figure 2.3. Program that computes $x + 1$ in \mathcal{S}_n .

Figure 2.4. Flow chart for computing $x - 1$ in \mathcal{S}_n .

[B]	IF X ENDS s_i GOTO A_i	$(1 \leq i \leq n)$
	GOTO E	
[A_i]	$X \leftarrow X^-$	
	$Y \leftarrow s_{i-1}Y$	$1 < i \leq n$
	GOTO C	
[A_1]	$X \leftarrow X^-$	
	IF $X \neq 0$ GOTO C_2	
	GOTO E	
[C_2]	$Y \leftarrow s_n Y$	
	GOTO B	
[C]	IF X ENDS s_i GOTO D_i	$(1 \leq i \leq n)$
	GOTO E	
[D_i]	$X \leftarrow X^-$	
	$Y \leftarrow s_i Y$	$1 \leq i \leq n$
	GOTO C	

Figure 2.5. Program that computes $x - 1$ in \mathcal{S}_n .

Exercises

- Let $A = \{s_1, s_2\}$. Write out the complete expansion of the macro $X \leftarrow Y$ in \mathcal{S}_2 .

2. Write a program in \mathcal{S}_n to compute the function f defined in Exercise 1.4.
3. Show that $f(u, v) = \widehat{uv}$ is computable in \mathcal{S}_n . (\widehat{uv} is the concatenation of u and v , defined in Chapter 1.)
4. Let $A = \{s_1, \dots, s_n\}$, and let $P(x)$ be the predicate on A^* which is true just when x has an even number of symbols. Show that $P(x)$ is computable in \mathcal{S}_n .
5. Write a program in \mathcal{S}_n to compute $\#(u, v)$ as defined in Exercise 1.6.
6. Give an expansion in \mathcal{S}_n for the macro $V \leftarrow V\sigma$, which means: Place the symbol σ to the right of the string that is the value of V .
7. Show that $f(x) = x^R$ is computable in \mathcal{S}_n . (x^R is defined in Chapter 1, Section 3.)
8. Let $A = \{s_1, \dots, s_n\}$, and let $g(u) = w$ for all strings u in A^* , where w is the base n notation for the number of symbols in u . Show that g is computable in \mathcal{S}_n .
9. Let $A = \{s_1, s_2\}$, and let \mathcal{P} be the \mathcal{S}_2 program

$$Y \leftarrow X + 1$$

Write out the computation of \mathcal{P} for input $x = s_2s_2$.

10. Let $A = \{s_1, s_2, s_3\}$, and let \mathcal{P} be the \mathcal{S}_3 program

$$Y \leftarrow X \div 1$$

Write out the computation of \mathcal{P} for input $x = s_1s_1$.

11. (a) Show that Theorem 1.1 in Chapter 3 holds if we substitute “computable in \mathcal{S}_n ” for “computable.”
- (b) Show that Theorems 2.1 and 2.2 in Chapter 3 hold if we substitute “computable in \mathcal{S}_n ” for “computable.”
- (c) Show that if $f(x_1, \dots, x_n)$ is primitive recursive, then it is computable in \mathcal{S}_n .

3. The Languages \mathcal{S} and \mathcal{S}_n

We now want to compare the functions that can be computed in the various languages we have been considering, namely, \mathcal{S} and the different \mathcal{S}_n . For the purpose of making this comparison, we take the point of view that, in all of the languages, computations are “really” dealing with numbers, and that strings on an n letter alphabet are simply data objects being used to represent numbers (using base n of course).

We shall see that in fact all of these languages are equivalent. That is, a function f is partially computable if and only if it is partially computable in each \mathcal{S}_n and therefore, also, f is partially computable in any one \mathcal{S}_n if and only if it is partially computable in all of them.

To begin with we have

Theorem 3.1. A function is partially computable if and only if it is partially computable in \mathcal{S}_1 .

Proof. It is easy to see that the languages \mathcal{S} and \mathcal{S}_1 are really the same. That is, the numerical effect of the instructions

$$V \leftarrow s_1 V \quad \text{and} \quad V \leftarrow V^-$$

in \mathcal{S}_1 is the same as that of the corresponding instructions in \mathcal{S} :

$$V \leftarrow V + 1 \quad \text{and} \quad V \leftarrow V - 1.$$

Furthermore, the condition “ V ENDS s_1 ” in \mathcal{S}_1 is equivalent to the condition $V \neq 0$ in \mathcal{S} . (Since s_1 is the only symbol, ending in s_1 is equivalent to being different from the null string.) ■

This theorem shows that results we obtain about the languages \mathcal{S}_n can always be specialized to give results about \mathcal{S} by setting $n = 1$.

Next we shall prove

Theorem 3.2. If a function is partially computable, then it is also partially computable in \mathcal{S}_n for each n .

Proof. Let the function f be computed by a program \mathcal{P} in the language \mathcal{S} . We translate \mathcal{P} into a program in \mathcal{S}_n by replacing each instruction of \mathcal{P} by a macro in \mathcal{S}_n as follows.

We replace each instruction $V \leftarrow V + 1$ by the macro $V \leftarrow V + 1$, each instruction $V \leftarrow V - 1$ by the macro $V \leftarrow V \dot{-} 1$, and each instruction IF $V \neq 0$ GOTO L by the macro IF $V \neq 0$ GOTO L . Here we are using the fact, proved at the end of the preceding section, that $x + 1$ and $x \dot{-} 1$ are both computable in base n , and hence can each be used to define a macro in \mathcal{S}_n .

It is then obvious that the new program computes in \mathcal{S}_n the same function f that \mathcal{P} computes in \mathcal{S} . ■

This is the first of many proofs by the method of *simulation*: a program in one language is “simulated” step by step by a corresponding program in a different language.

We could now prove directly that if a function is partially computable in \mathcal{S}_n for any particular n , then it is in fact partially computable in our original sense. But it will be easier to delay doing so since the result will be an automatic consequence of our work on Post-Turing programs.

Exercises

- Give a primitive recursive function $b_1(n, x)$ such that any partial function computed by an \mathcal{S} program with x instructions is computed by some \mathcal{S}_n program with no more than $b_1(n, x)$ instructions.
- Give a primitive recursive function $b_2^{(m)}(n, x_1, \dots, x_m, y)$ such that any partial function $f(x_1, \dots, x_m)$ computed by an \mathcal{S} program in y steps on inputs x_1, \dots, x_m is computed by some \mathcal{S}_n program in no more than $b_2^{(m)}(n, x_1, \dots, x_m, y)$ steps. [Hint: Note that after y steps no variable holds a value larger than $\max\{x_1, \dots, x_m\} + y$.]
- Let n be some fixed number > 0 , and let $\#(\mathcal{P})$ be a numbering scheme for \mathcal{S}_n programs defined exactly like the numbering scheme for \mathcal{S} programs given in Chapter 4, except that $\#(I) = \langle a, \langle b, c \rangle \rangle$, where

$$b = \begin{cases} 0 & \text{if the statement in } I \text{ is } V \leftarrow V^- \\ i & \text{if the statement in } I \text{ is } V \leftarrow s_i V \\ \#(L') \cdot n + i & \text{if the statement in } I \text{ is IF } V \text{ ENDS } s_i \text{ GOTO } L'. \end{cases}$$

- (a) Define

$\text{HALT}_n(x, y) \Leftrightarrow \mathcal{S}_n$ program y eventually halts on input x .

Show that the predicate $\text{HALT}_n(x, y)$ is not computable in \mathcal{S}_n .

- (b) Define the universal function $\Phi_n^{(m)}$ for m -ary functions partially computable in \mathcal{S}_n as follows:

$$\Phi_n^{(m)}(x_1, \dots, x_m, y) = \psi_{\mathcal{P}}^{(m)}(x_1, \dots, x_m), \quad \text{where } \#(\mathcal{P}) = y.$$

(Of course, $\psi_{\mathcal{P}}^{(m)}$ is the m -ary partial function computed by the \mathcal{S}_n program \mathcal{P} .) Show that for each $m > 0$, the function $\Phi_n^{(m)}(x_1, \dots, x_m, y)$ is partially computable in \mathcal{S}_n .

- (c)* State and prove a version of the parameter theorem for \mathcal{S}_n .
 (d)* State and prove a version of the recursion theorem for \mathcal{S}_n .
 (e)* Show that \mathcal{S}_n is an acceptable programming system. [See Exercise 5.4 in Chapter 4 for the definition of acceptable programming systems.]
- 4.* Give an upper bound on the length of the shortest \mathcal{S}_1 program which computes the function $\Phi_y(x)$ defined in Chapter 4. [See Exercise 3.6 in Chapter 4.]

4. Post – Turing Programs

In this section, we will study yet another programming language for string manipulation, the Post–Turing language \mathcal{T} . Unlike \mathcal{S}_n , the language \mathcal{T} has no variables. All of the information being processed is placed on one linear tape. We can conveniently think of the tape as ruled into squares each of which can carry a single symbol (see Fig. 4.1). The tape is thought of as infinite in both directions. Each step of a computation is sensitive to just one symbol on the tape, the symbol on the square being “scanned.” We can think of the tape passing through a device (like a tape recorder), or we can think of the computer as a tapehead that moves along the tape and is at each moment on one definite square (or we might say “tile”). With this simple scheme, there are not many steps we can imagine. The symbol being scanned can be altered. (That is, a new symbol can be “printed” in its place.) Or which instruction of a program is to be executed next can depend on which symbol is currently being scanned. Or, finally, the head can move one square to the left or right of the square presently scanned. We are led to the language shown in Table 4.1.

Although the formulation of \mathcal{T} we have presented is closer in spirit to that originally given by Emil Post, it was Turing’s analysis of the computation process that has made this formulation seem so appropriate. This language has played a fundamental role in theoretical computer science.

Turing’s analysis was obtained by abstracting from the process carried out by a human being engaged in calculating according to a mechanical deterministic algorithm. Turing reasoned that there was no loss of generality in assuming that the person used a linear paper (like the paper tape in an old-fashioned adding machine or a printing calculator) instead of two-dimensional sheets of paper. Such a calculator is then engaged in observing symbols and writing symbols. Again without loss of generality, we can assume that only one symbol at a time is observed, since any finite group of symbols can be regarded as a single “megasymbol.” Finally, we can assume that when the calculator shifts attention it is to an immediately adjacent symbol. For, to look, say, three symbols to the left is equivalent to moving one symbol to the left three successive times. And now we have arrived at precisely the Post–Turing language.

In order to speak of a function being computed by a Post–Turing program, we will need to deal with input and output. Let us suppose that

--	--	--	--	--	--

Figure 4.1

Table 4.1

Instruction	Interpretation
PRINT σ	Replace the symbol on the square being scanned by σ .
IF σ GOTO L	GOTO the first instruction labeled L if the symbol currently scanned is σ ; otherwise, continue to the next instruction.
RIGHT	Scan the square immediately to the <i>right</i> of the square presently scanned.
LEFT	Scan the square immediately to the <i>left</i> of the square presently scanned.

we are dealing with string functions on the alphabet $A = \{s_1, s_2, \dots, s_n\}$. We will use an additional symbol, written s_0 , which we call the *blank* and use as a punctuation mark. *Often we write B for the blank instead of s_0 .* All of our computations will be arranged so that *all but a finite number of squares on the tape are blank*, i.e., contain the symbol B . We show the contents of a tape by exhibiting a finite section containing all of the nonblank squares. We indicate the square currently being scanned by an arrow pointing up, just below the scanned square.

For example we can write

$$s_1 \ s_2 \ B \ s_2 \ s_1$$

↑

to indicate that the tape consists of $s_1 s_2 B s_2 s_1$ with blank squares to the left and right, and that the square currently scanned contains the s_2 furthest to the right. We speak of a *tape configuration* as consisting of the tape contents together with a specification of one square as being currently scanned.

Now, to compute a partial function $f(x_1, \dots, x_m)$ of m variables on A^* , we need to place the m strings x_1, \dots, x_m on the tape initially. We do this using the initial tape configuration:

$$B \ x_1 \ B \ x_2 \ \dots \ B \ x_m,$$

↑

That is, the inputs are separated by single blanks, and the symbol initially scanned is the blank immediately to the left of x_1 . Here are a few examples:

1. $n = 1$, so the alphabet is $\{s_1\}$. We want to compute a function $f(x_1, x_2)$ and the initial values are $x_1 = s_1 s_1$, $x_2 = s_1$. Then the tape configuration initially will be

$$B \ s_1 \ s_1 \ B \ s_1.$$

↑

Of course, there are infinitely many blank squares to the left and right of the finite section we have shown:

$$\dots \ B \ B \ B \ B \ s_1 \ s_1 \ B \ s_1 \ B \ B \ B \ \dots .$$

↑

2. $n = 2$, $x_1 = s_1s_2$, $x_2 = s_2s_1$, $x_3 = s_2s_2$. Then the tape configuration is initially

$$B \ s_1 \ s_2 \ B \ s_2 \ s_1 \ B \ s_2 \ s_2.$$

↑

3. $n = 2$, $x_1 = 0$, $x_2 = s_2s_1$, $x_3 = s_2$. Then the tape configuration is initially

$$B \ B \ s_2 \ s_1 \ B \ s_2.$$

↑

4. $n = 2$, $x_1 = s_1s_2$, $x_2 = s_2s_1$, $x_3 = 0$. Then the tape configuration is initially

$$B \ s_1 \ s_2 \ B \ s_2 \ s_1 \ B.$$

↑

Note that there is no way to distinguish this initial tape configuration from that for which there are only two inputs $x_1 = s_1s_2$ and $x_2 = s_2s_1$. In other words, with this method of placing inputs on the tape, the number of arguments must be provided externally. It cannot be read from the tape.

A simple example of a Post-Turing program is given in Fig. 4.2.

Beginning with input x , this program outputs s_2s_1x . More explicitly, beginning with a tape configuration

$$B \ x$$

↑

this program halts with the tape configuration

$$B \ s_2 \ s_1 \ x.$$

↑

Figure 4.2

```
PRINT s1
LEFT
PRINT s2
LEFT
```

[A]	RIGHT IF s_1 GOTO A IF s_2 GOTO A IF s_3 GOTO A PRINT s_1 RIGHT PRINT s_1
[C]	LEFT IF s_1 GOTO C IF s_2 GOTO C IF s_3 GOTO C

Figure 4.3

Next, for a slightly more complicated example, we consider Fig. 4.3. Here we are assuming that the alphabet is $\{s_1, s_2, s_3\}$. Let x be a string on this alphabet. Beginning with a tape configuration

$$\begin{matrix} B & x \\ & \uparrow \end{matrix}$$

this program halts with the tape configuration

$$\begin{matrix} B & x & s_1 & s_1 \\ & \uparrow & & \end{matrix}.$$

The computation proceeds by first moving right until the blank to the right of x is located. The symbol s_1 is then printed twice and then the computation proceeds by moving left until the blank to the left of x is again located.

Figure 4.4 exhibits another example, this time with the alphabet $\{s_1, s_2\}$. The effect of this program is to “erase” all of the occurrences of s_2 in the input string, that is to replace each s_2 by B . For the purpose of reading output values off the tape, these additional B s are ignored. Thus, if $f(x)$ is the function which this last program computes, we have, for example,

$$\begin{aligned} f(s_2 s_1 s_2) &= s_1, \\ f(s_1 s_2 s_1) &= s_1 s_1, \\ f(0) &= 0. \end{aligned}$$

Of course, the initial tape configuration

$$\begin{matrix} B & s_1 & s_2 & s_1 \\ & \uparrow & & \end{matrix}$$

[C]	RIGHT IF B GOTO E IF s_2 GOTO A IF s_1 GOTO C
[A]	PRINT B IF B GOTO C

Figure 4.4

[A]	RIGHT IF B GOTO E PRINT M
[B]	RIGHT IF s_1 GOTO B
[C]	RIGHT IF s_1 GOTO C PRINT s_1
[D]	LEFT IF s_1 GOTO D IF B GOTO D PRINT s_1 IF s_1 GOTO A

Figure 4.5

leads to the final tape configuration

$$B \ s_1 \ B \ s_1 \ B$$

↑

but the blanks are ignored in reading the output.

For our final example we are computing a string function on the alphabet $\{s_1\}$. However, the program uses three symbols, B , s_1 , and M . The symbol M is a *marker* to keep track of a symbol being copied. The program is given in Fig. 4.5. Beginning with the tape configuration

$$B \ u$$

↑

where u is a string in which only the symbol s_1 occurs, this program will terminate with the tape configuration

$$B \ u \ B \ u.$$

↑

(Thus we can say that this program computes the function $2x$ using unary notation.) The computation proceeds by replacing each successive s_1 (going from left to right) by the marker M and then copying the s_1 on the right.

We conclude this section with some definitions. Let $f(x_1, \dots, x_m)$ be an m -ary partial function on the alphabet $\{s_1, \dots, s_n\}$. Then the program \mathcal{P} in the Post-Turing language \mathcal{T} is said to *compute* f if when started in the tape configuration

$$\begin{array}{ccccccc} B & x_1 & B & \dots & B & x_m \\ & \uparrow & & & & & \end{array}$$

it eventually halts if and only if $f(x_1, \dots, x_m)$ is defined and if, on halting, the string $f(x_1, \dots, x_m)$ can be read off the tape by ignoring all symbols other than s_1, \dots, s_n . (That is, any “markers” left on the tape as well as blanks are to be ignored.) Note that we are thus permitting \mathcal{P} to contain instructions that mention symbols other than s_1, \dots, s_n .

The program \mathcal{P} will be said to compute f *strictly* if two additional conditions are met:

1. no instruction in \mathcal{P} mentions any symbol other than s_0, s_1, \dots, s_n ;
2. whenever \mathcal{P} halts, the tape configuration is of the form

$$\begin{array}{ccccccccc} \dots & B & B & B & B & y & B & B & \dots \\ & & & & & \uparrow & & & \end{array}$$

where the string y contains no blanks.

Thus when \mathcal{P} computes f strictly, the output is available in a consecutive block of squares on the tape.

Exercises

1. Write out the computation performed by the Post-Turing program in Fig. 4.4 on input string $s_1s_2s_2s_1$. Do the same for input $s_1s_2s_3s_1$.
2. Write out the computation performed by the Post-Turing program in Fig. 4.5 on input string $s_1s_1Bs_1s_1s_1$. Do the same for input $s_1s_1Bs_1Bs_1s_1$.
3. For each of the following functions, construct a Post-Turing program that computes the function strictly.
 - (a) $f(u, v) = \widehat{uv}$.
 - (b) the predicate $P(x)$ given in Exercise 2.4.
 - (c) the function $f(x) = x^R$ (see Exercise 2.7).
 - (d) the function $\#(u, v)$ given in Exercise 1.6.

4. For each of the following functions, construct a Post–Turing program using only the symbols s_0, s_1 that computes the function in base 1 strictly.
 - (a) $f(x, y) = x + y$.
 - (b) $f(x) = 2x$.
 - (c) $f(x, y) = x \dot{-} y$.
 - (d) $f(x, y) = 2x + y \dot{-} 1$.
5. Construct a Post–Turing program using only the symbols s_0, s_1, s_2 that computes the function $s(x) = x + 1$ in base 2 strictly.

5. Simulation of \mathcal{S}_n in \mathcal{T}

In this section we will prove

Theorem 5.1. If $f(x_1, \dots, x_m)$ is partially computable in \mathcal{S}_n , then there is a Post–Turing program that computes f strictly.

Let \mathcal{P} be a program in \mathcal{S}_n which computes f . We assume that in addition to the input variables X_1, \dots, X_m and the output variable Y , \mathcal{P} uses the local variables Z_1, \dots, Z_k . Thus, altogether \mathcal{P} uses $m + k + 1$ variables:

$$X_1, \dots, X_m, Z_1, \dots, Z_k, Y.$$

We set $l = m + k + 1$ and write these variables, in the same order, as

$$V_1, \dots, V_l.$$

We shall construct a Post–Turing program \mathcal{Q} that simulates \mathcal{P} step by step. Since all of the information available to \mathcal{Q} will be on the tape, we must allocate space on the tape to contain the values of the variables V_1, \dots, V_l . Our scheme is simply that at the beginning of each simulated step, the tape configuration will be as follows:

$$B \ x_1 \ B \ x_2 \ B \ \dots \ B \ x_m \ B \ z_1 \ B \ \dots \ B \ z_k \ B \ y,$$

↑

where $x_1, x_2, \dots, x_m, z_1, \dots, z_k, y$ are the current values computed for the variables $X_1, X_2, \dots, X_m, Z_1, \dots, Z_k, Y$. This scheme is especially convenient in that the initial tape configuration

$$B \ x_1 \ B \ x_2 \ B \ \dots \ B \ x_m$$

↑

is already in the correct form, since the remaining variables are initialized to be 0. So we must show how to program the effect of each instruction

type of \mathcal{S}_n in the language \mathcal{T} . Various macros in \mathcal{T} will be useful in doing this, and we now present them.

The macro

GOTO L

has the expansion

```
IF  $s_0$  GOTO  $L$ 
IF  $s_1$  GOTO  $L$ 
:
IF  $s_n$  GOTO  $L$ 
```

The macro

RIGHT TO NEXT BLANK

has the expansion

```
[A]      RIGHT
          IF  $B$  GOTO  $E$ 
          GOTO  $A$ 
```

Similarly the macro

LEFT TO NEXT BLANK

has the expansion

```
[A]      LEFT
          IF  $B$  GOTO  $E$ 
          GOTO  $A$ 
```

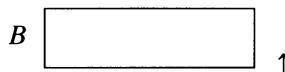
The macro

MOVE BLOCK RIGHT

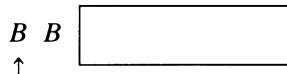
has the expansion

```
[C]      LEFT
          IF  $s_0$  GOTO  $A_0$ 
          IF  $s_1$  GOTO  $A_1$ 
          :
          IF  $s_n$  GOTO  $A_n$ 
[Ai]  RIGHT }  
          PRINT  $s_i$  }  
          LEFT }  
          GOTO  $C$  }  
          } i = 1, 2, ..., n
[A0]  RIGHT
          PRINT  $B$ 
          LEFT
```

The effect of the macro MOVE BLOCK RIGHT beginning with a tape configuration



in which the string in the rectangular box contains no blanks, is to terminate with the tape configuration



Finally we will use the macro

ERASE A BLOCK

whose expansion is

```
[A]      RIGHT
          IF B GOTO E
          PRINT B
          GOTO A
```

This program causes the head to move to the right, with everything erased between the square at which it begins and the first blank to its right.

We adopt the convention that a number ≥ 0 in square brackets after the name of a macro indicates that the macro is to be repeated that number of times. For example,

RIGHT TO NEXT BLANK [3]

is short for

```
RIGHT TO NEXT BLANK
RIGHT TO NEXT BLANK
RIGHT TO NEXT BLANK
```

We are now ready to show how to simulate the three instruction types in the language \mathcal{S}_n by Post-Turing programs. We begin with

$$V_j \leftarrow s_i V_j$$

In order to place the symbol s_i to the left of the j th variable on the tape, the values of the variables V_j, \dots, V_l must all be moved over one square to the right to make room. After the s_i has been inserted, we must remember

to go back to the blank at the left of the value of V_1 in order to be ready for the next simulated instruction. The program is

```
RIGHT TO NEXT BLANK [ $l$ ]
MOVE BLOCK RIGHT [ $l - j + 1$ ]
RIGHT
PRINT  $s_i$ 
LEFT TO NEXT BLANK [ $j$ ]
```

Next we must show how to simulate

$$V_j \leftarrow V_j^-$$

The complication is that if the value of V_j is the null word, we want it left unchanged. So we move to the blank immediately to the right of the value of V_j . By moving one square to the left we can detect whether the value of V_j is null (if it is, there are two consecutive blanks). Here is the program:

```
RIGHT TO NEXT BLANK [ $j$ ]
LEFT
IF  $B$  GOTO  $C$ 
MOVE BLOCK RIGHT [ $j$ ]
RIGHT
GOTO  $E$ 
[C] LEFT TO NEXT BLANK [ $j - 1$ ]
```

The final instruction type in \mathcal{S}_n is

IF V_j ENDS s_i GOTO L

and the corresponding Post-Turing program is

```
RIGHT TO NEXT BLANK [ $j$ ]
LEFT
IF  $s_i$  GOTO  $C$ 
GOTO  $D$ 
[C] LEFT TO NEXT BLANK [ $j$ ]
GOTO  $L$ 
[D] RIGHT
LEFT TO NEXT BLANK [ $j$ ]
```

This completes the simulation of the three instruction types of \mathcal{S}_n . Thus, given our program \mathcal{P} in the language \mathcal{S}_n , we can compile a

corresponding program of \mathcal{T} . When this corresponding program terminates, the tape configuration will be

$$\dots B \ B \ B \ x_1 \ B \ \dots B \ x_m \ B \ z_1 \ B \ \dots B \ z_k \ B \ y \ B \ B \ B \ \dots,$$

↑

where the values between blanks are those of the variables of \mathcal{P} on its termination. However, we wish only y to remain as output. Hence to obtain our program \mathcal{Q} in the language \mathcal{T} we put at the end of the compiled Post-Turing program the following:

ERASE A BLOCK [l - 1]

After this last has been executed, all but the last block will have been erased and the tape configuration will be

$$\dots B \ B \ B \ B \ y \ B \ B \ B \ \dots$$

↑

Thus, the output is in precisely the form required for us to be able to assert that our Post-Turing program computes f strictly.

Exercises

1. (a) Use the construction in the proof of Theorem 5.1 to give a Post-Turing program that computes the function $f(x)$ computed by the \mathcal{S}_2 program

[A]	IF X ENDS s_1 GOTO B $X \leftarrow X^-$ IF $X \neq 0$ GOTO A GOTO E
[B]	$Y \leftarrow s_1 Y$ $X \leftarrow X^-$ GOTO A

- (b) Do the same as (a) for $f(x_1, x_2)$.
2. Answer question 1(a) with the instruction [B] $Y \leftarrow s_1 Y$ replaced by [B] $Y \leftarrow Y + 1$.
3. Give a primitive recursive function $b_1(n, x, z)$ such that any partial function computed by an \mathcal{S}_n program that has x instructions and that uses only variables among $X_1, \dots, X_l, Z_1, \dots, Z_k, Y$ is computed strictly by a Post-Turing program with no more than $b_1(n, x, l + k + 1)$ instructions.

4. Give a primitive recursive function $b_2^{(m)}(n, x_1, \dots, x_m, y, z)$ such that any partial function computed by an \mathcal{S}_n program in y steps on input x_1, \dots, x_m , using only variables among $X_1, \dots, X_l, Z_1, \dots, Z_k, Y$, is computed strictly by some Post-Turing program in no more than $b_2^{(m)}(n, x_1, \dots, x_m, y, l + k + 1)$ steps. [Hint: Note that after y steps no variable holds a value larger than $\max\{x_1, \dots, x_m\} + y$.]
- 5.* Give an upper bound on the length of the shortest Post-Turing program that computes $\Phi_y(x)$. [See Exercise 3.4.]

6. Simulation of \mathcal{T} in \mathcal{S}

In this section we will prove

Theorem 6.1. If there is a Post-Turing program that computes the partial function $f(x_1, \dots, x_m)$, then f is partially computable.

What this theorem asserts is that if the m -ary partial function f on A^* is computed by a program of \mathcal{T} , then there is a program of \mathcal{S} that computes f (regarded as an m -ary partial function on the base n numerical values of the strings). Before giving the proof we observe some of the consequences of this theorem. As shown in Fig. 6.1, the theorem completes a “circle” of implications. Thus all of the conditions in the figure are equivalent. To summarize:

Theorem 6.2. Let f be an m -ary partial function on A^* , where A is an alphabet of n symbols. Then the following conditions are all equivalent:

1. f is partially computable;
2. f is partially computable in \mathcal{S}_n ;

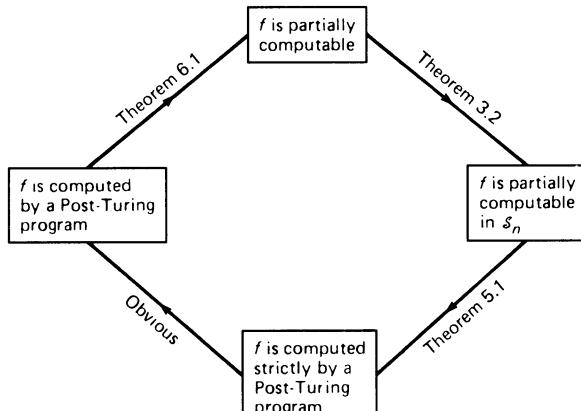


Figure 6.1

3. f is computed strictly by a Post–Turing program;
4. f is computed by a Post–Turing program.

The equivalence of so many different notions of computability constitutes important evidence for the correctness of our identification of intuitive computability with these notions, i.e., for the correctness of Church’s thesis.

Shifting our point of view to that of an m -ary partial function on N , we have

Corollary 6.3. For any $n, l \geq 1$, an m -ary partial function f on N is partially computable in \mathcal{S}_n if and only if it is also partially computable in \mathcal{S}_l .

Proof. Each of these conditions is equivalent to the function f being partially computable. ■

By considering the language \mathcal{S}_1 we have

Corollary 6.4. Every partially computable function is computed strictly by some Post–Turing program that uses only the symbols s_0, s_1 .

Now we return to the proof of Theorem 6.1. Let \mathcal{P} be a Post–Turing program that computes f . We want to construct a program \mathcal{Q} in the language \mathcal{S} that computes f . \mathcal{Q} will consist of three sections:

BEGINNING
MIDDLE
END

The MIDDLE section will simulate \mathcal{P} in a step-by-step “interpretive” manner. The task of BEGINNING is to arrange the input to \mathcal{Q} in the appropriate format for MIDDLE, and the task of END is to extract the output.

Let us suppose that f is an m -ary partial function on A^* , where $A = \{s_1, \dots, s_n\}$. The Post–Turing program \mathcal{P} will also use the blank B and perhaps additional symbols (we are not assuming that the computation is *strict!*) s_{n+1}, \dots, s_r . We write the symbols that \mathcal{P} uses in the order

$$s_1, \dots, s_n, s_{n+1}, \dots, s_r, B.$$

The program \mathcal{Q} will simulate \mathcal{P} by using the numbers that strings on this alphabet represent in base $r + 1$ as “codes” for the corresponding strings. Note that as we have arranged the symbols, the blank B represents the number $r + 1$. For this reason we will write the blank as s_{r+1} instead of s_0 . The tape configuration at a given stage in the computation by \mathcal{P} will be

kept track of by \mathcal{Q} using three numbers stored in the variables L , H , and R . The value of H will be the numerical value of the symbol currently being scanned by the *head*. The value of L will be a number which represents in base $r + 1$ a string of symbols w such that the tape contents to the left of the head consists of infinitely many blanks followed by w . The value of R represents in a similar manner the string of symbols to the right of the head. For example, consider the tape configuration

$$\dots \ B \ B \ B \ B \ s_2 \ s_1 \ B \ s_3 \ s_1 \ s_2 \ B \ B \ B \dots$$

\uparrow

Here $r = 3$, so we will use the base 4. Then we would have

$$H = 3.$$

We might have

$$L = 2 \cdot 4^2 + 1 \cdot 4 + 4 = 40,$$

$$R = 1 \cdot 4 + 2 = 6.$$

An alternative representation could show some of the blanks on the left or right explicitly. For example, recalling that B represents $r + 1 = 4$,

$$L = 4 \cdot 4^3 + 2 \cdot 4^2 + 1 \cdot 4 + 4 = 296,$$

$$R = 1 \cdot 4^3 + 2 \cdot 4^2 + 4 \cdot 4 + 4 = 116.$$

Now it is easy to simulate the instruction types of \mathcal{T} by programs of \mathcal{S} .

An instruction PRINT s_i is simulated by

$$H \leftarrow i$$

An instruction IF s_i GOTO L is simulated by

$$\text{IF } H = i \text{ GOTO } L$$

An instruction RIGHT is simulated by

```

 $L \leftarrow \text{CONCAT}_{r+1}(L, H)$ 
 $H \leftarrow \text{LTEND}_{r+1}(R)$ 
 $R \leftarrow \text{LTRUNC}_{r+1}(R)$ 
 $\text{IF } R \neq 0 \text{ GOTO } E$ 
 $R \leftarrow r + 1$ 

```

Similarly an instruction LEFT is simulated by

$$\begin{aligned} R &\leftarrow \text{CONCAT}_{r+1}(H, R) \\ H &\leftarrow \text{RTEND}_{r+1}(L) \\ L &\leftarrow \text{RTRUNC}_{r+1}(L) \\ \text{IF } L &\neq 0 \text{ GOTO } E \\ L &\leftarrow r + 1 \end{aligned}$$

Now the section MIDDLE of \mathcal{Q} can be assembled simply by replacing each instruction of \mathcal{P} by its simulation.

In writing BEGINNING and END we must deal with the fact that f is an m -ary function on $\{s_1, \dots, s_n\}^*$. Thus the initial values of X_1, \dots, X_m for \mathcal{Q} will be numbers that represent the input strings in base n . Theorem 1.1 will enable us to change base as required. The section BEGINNING has the task of calculating the initial values of L, H, R , that is, the values corresponding to the tape configuration

$$B \ x_1 \ B \ x_2 \ B \ \dots \ B \ x_m,$$

↑

where the numbers x_1, \dots, x_m are represented in base n notation. Thus the section BEGINNING of \mathcal{Q} can simply be taken to be

$$\begin{aligned} L &\leftarrow r + 1 \\ H &\leftarrow r + 1 \\ Z_1 &\leftarrow \text{UPCHANGE}_{n, r+1}(X_1) \\ Z_2 &\leftarrow \text{UPCHANGE}_{n, r+1}(X_2) \\ &\vdots \\ Z_m &\leftarrow \text{UPCHANGE}_{n, r+1}(X_m) \\ R &\leftarrow \text{CONCAT}_{r+1}(Z_1, r + 1, Z_2, r + 1, \dots, r + 1, Z_m) \end{aligned}$$

Finally, the section END of \mathcal{Q} can be taken simply to be

$$\begin{aligned} Z &\leftarrow \text{CONCAT}_{r+1}(L, H, R) \\ Y &\leftarrow \text{DOWNCHANGE}_{n, r+1}(Z) \end{aligned}$$

We have now completed the description of the program \mathcal{Q} that simulates \mathcal{P} , and our proof is complete. ■

Exercises

1. Use the construction in the proof of Theorem 6.1 to give an \mathcal{S} program that computes the same unary function as the Post-Turing

2. For any Post–Turing program \mathcal{P} , let $\#(\mathcal{P})$ be $\#(\mathcal{Q})$, where \mathcal{Q} is the \mathcal{S} program obtained for \mathcal{P} in the proof of Theorem 6.1, and let $\text{HALT}_{\mathcal{T}}(x, y)$ be defined

$\text{HALT}_{\mathcal{T}}(x, y) \Leftrightarrow y$ is the number of a Post–Turing program
that eventually halts on input x .

Show that $\text{HALT}_{\mathcal{T}}(x, y)$ is not a computable predicate.

- 3.* Show that the Post–Turing programs, under an appropriate ordering $\mathcal{P}_0, \mathcal{P}_1, \dots$, are an acceptable programming system. [See Exercise 5.4 in Chapter 4 for the definition of acceptable programming systems.]

6

Turing Machines

1. Internal States

Now we turn to a variant of the Post–Turing language that is closer to Turing’s original formulation. Instead of thinking of a list of instructions, we imagine a device capable of various internal *states*. The device is, at any particular instant, scanning a square on a linear tape just like the one used by Post–Turing programs. The combination of the current internal state with the symbol on the square currently scanned is then supposed to determine the next “action” of the device. As suggested by Turing’s analysis of the computation process (see Chapter 5, Section 4), we can take the next action to be either “printing” a symbol on the scanned square or moving one square to the right or left. Finally, the device must be permitted to enter a new state.

We use the symbols q_1, q_2, q_3, \dots to represent states and we write s_0, s_1, s_2, \dots to represent symbols that can appear on the tape, where as usual $s_0 = B$ is the “blank.” By a *quadruple* we mean an expression of one of the following forms consisting of four symbols:

1. $q_i \ s_j \ s_k \ q_l,$
2. $q_i \ s_j \ R \ q_l,$
3. $q_i \ s_j \ L \ q_l.$

We intend a quadruple of type 1 to signify that in state q_i scanning symbol s_j , the device will print s_k and go into state q_l . Similarly, a quadruple of type 2 signifies that in state q_i scanning s_j the device will move one square to the right and then go into state q_l . Finally, a quadruple of type 3 is like one of type 2 except that the motion is to the left.

We now define a *Turing machine* to be a finite set of quadruples, no two of which begin with the same pair $q_i s_j$. Actually, any finite set of quadruples is called a *nondeterministic Turing machine*. But for the present we will deal only with *deterministic Turing machines*, which satisfy the additional “consistency” condition forbidding two quadruples of a given machine to begin with the same pair $q_i s_j$, thereby guaranteeing that at any stage a Turing machine is capable of only one action. Nondeterministic Turing machines are discussed in Section 5.

The *alphabet* of a given Turing machine \mathcal{M} consists of all of the symbols s_i which occur in quadruples of \mathcal{M} except s_0 .

We stipulate that a Turing machine always begins in state q_1 . Moreover, a Turing machine will *halt* if it is in state q_i scanning s_j and *there is no quadruple of the machine which begins $q_i s_j$* . With these understandings, and using the same conventions concerning input and output that were employed in connection with Post-Turing programs, it should be clear what it means to say that some given Turing machine \mathcal{M} *computes* a partial function f on A^* for a given alphabet A .

Just as for Post-Turing programs, we may speak of a Turing machine \mathcal{M} that computes a function *strictly*, namely: assuming that \mathcal{M} computes f where f is a partial function on A^* , we say that \mathcal{M} *computes f strictly* if

1. the alphabet of \mathcal{M} is a subset of A ;
2. whenever \mathcal{M} halts, the final configuration has the form

By

↑

q_i

where y contains no blanks.

Writing $s_0 = B$, $s_1 = 1$ consider the Turing machine with alphabet {1}:

q_1	B	R	q_2
q_2	1	R	q_2
q_2	B	1	q_3
q_3	1	R	q_3
q_3	B	1	q_1 .

Table 1.1

Symbol	State		
	q_1	q_2	q_3
B	$R\ q_2$	$1\ q_3$	$1\ q_1$
1		$R\ q_2$	$R\ q_3$

We can check the computation:

$$\begin{array}{ccccccc}
 B111, B111, \dots, B111B, B1111, B1111B, B11111 \\
 \uparrow & \uparrow & & \uparrow & \uparrow & \uparrow & \uparrow \\
 q_1 & q_2 & & q_2 & q_3 & q_3 & q_1
 \end{array}$$

The computation halts because there is no quadruple beginning $q_1 1$. Clearly, this Turing machine computes (but not strictly) the function $f(x) = x + 2$, where we are using unary (base 1) notation. The steps of the computation, which explicitly exhibit the state of the machine, the string of symbols on the tape, as well as the individual square on the tape being scanned, are called *configurations*.

It is sometimes helpful to exhibit a Turing machine by giving a state versus symbol table. Thus, for example the preceding Turing machine could be represented as shown in Table 1.1.

Another useful representation is by a state transition diagram. The Turing machine being discussed thus could be represented by the diagram shown in Fig. 1.1.

We now prove

Theorem 1.1. Any partial function that can be computed by a Post-Turing program can be computed by a Turing machine using the same alphabet.

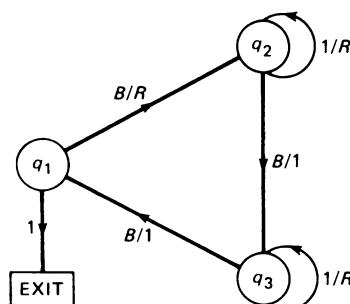


Figure 1.1

Proof. Let \mathcal{P} be a given Post–Turing program consisting of the instructions I_1, \dots, I_K , and let s_0, s_1, \dots, s_n be a list that includes all of the symbols mentioned in \mathcal{P} . We shall construct a Turing machine \mathcal{M} that simulates \mathcal{P} .

The idea is that \mathcal{M} will be in state q_i precisely when \mathcal{P} is about to execute instruction I_i . Thus, if I_i is “PRINT s_k ,” then we place in \mathcal{M} all of the quadruples

$$q_i \ s_j \ s_k \ q_{i+1}, \quad j = 0, 1, \dots, n.$$

If I_i is “RIGHT,” then we place in \mathcal{M} all of the quadruples

$$q_i \ s_j \ R \ q_{i+1}, \quad j = 0, 1, \dots, n.$$

If I_i is “LEFT,” then we place in \mathcal{M} all of the quadruples

$$q_i \ s_j \ L \ q_{i+1}, \quad j = 0, 1, \dots, n.$$

Finally, if I_i is “IF s_k GOTO L ,” let m be the least number such that I_m is labeled L if there is an instruction of \mathcal{P} labeled L ; otherwise let $m = K + 1$. We place in \mathcal{M} the quadruple

$$q_i \ s_k \ s_k \ q_m$$

as well as all of the quadruples:

$$q_i \ s_j \ s_j \ q_{i+1}, \quad j = 0, 1, \dots, n; \quad j \neq k.$$

It is clear that the actions of \mathcal{M} correspond precisely to the instructions of \mathcal{P} , so we are done. ■

Using Corollary 6.4 from Chapter 5 and the *proof* of Theorem 1.1, we have

Theorem 1.2. Let f be an m -ary partially computable function on A^* for a given alphabet A . Then there is a Turing machine \mathcal{M} that computes f strictly.

It is particularly interesting to apply this theorem to the case $A = \{1\}$. Thus, if $f(x_1, \dots, x_m)$ is any partially computable function on N , there is a Turing machine that computes f using only the symbols B and 1 . The initial configuration corresponding to inputs x_1, \dots, x_m is

$$\begin{array}{ccccccc} B & 1^{[x_1]} & B & \dots & B & 1^{[x_m]} \\ & \uparrow & & & & & \\ & q_1 & & & & & \end{array}$$

and the final configuration when $f(x_1, \dots, x_m) \downarrow$ will be

$$\begin{array}{c} B = 1^{[f(x_1, \dots, x_m)]}. \\ \uparrow \\ q_{K+1} \end{array}$$

Next we shall consider a variant notion of Turing machines: machines that consist of quintuples instead of quadruples. There are two kinds of quintuples:

$$\begin{array}{ccccc} q_i & s_j & s_k & R & q_l, \\ q_i & s_j & s_k & L & q_l. \end{array}$$

The first quintuple signifies that when the machine is in state q_i scanning s_j it will print s_k and *then* move one square to the right and go into state q_l . And naturally, the second quintuple is the same, except that the motion is to the left. A finite set of quintuples no two of which begin with the same pair $q_i s_j$ is called a *quintuple Turing machine*. We can easily prove

Theorem 1.3. Any partial function that can be computed by a Turing machine can be computed by a quintuple Turing machine using the same alphabet.

Proof. Let \mathcal{M} be a Turing machine with states q_1, \dots, q_K and alphabet $\{s_1, \dots, s_n\}$. We construct a quintuple Turing machine $\bar{\mathcal{M}}$ to simulate \mathcal{M} . The states of $\bar{\mathcal{M}}$ will be $q_1, \dots, q_K, q_{K+1}, \dots, q_{2K}$.

For each quadruple of \mathcal{M} of the form $q_i s_j R q_l$ we place the corresponding quintuple $q_i s_j s_k R q_l$ in $\bar{\mathcal{M}}$. Similarly, for each quadruple $q_i s_j L q_l$ in \mathcal{M} , we place the quintuple $q_i s_j s_k L q_l$ in $\bar{\mathcal{M}}$. And, for each quadruple $q_i s_j s_k q_l$ in \mathcal{M} , we place in $\bar{\mathcal{M}}$ the quintuple $q_i s_j s_k R q_{K+1}$. Finally we place in $\bar{\mathcal{M}}$ all quintuples of the form

$$q_{K+i} s_j s_k L q_i, \quad i = 1, \dots, K; \quad j = 0, 1, \dots, n.$$

Quadruples requiring motion are simulated easily by quintuples. But a quadruple requiring a “print” necessitates using a quintuple which causes a motion after the “print” has taken place. The final list of quintuples undoes the effect of this unwanted motion. The extra states q_{K+1}, \dots, q_{2K} serve to “remember” that we have gone a square too far to the right. ■

Finally, we will complete another circle by proving

Theorem 1.4. Any partial function that can be computed by a quintuple Turing machine can be computed by a Post-Turing program using the same alphabet.

Combining Theorems 1.1, 1.3, and 1.4, we will have

Corollary 1.5. For a given partial function f , the following are equivalent:

1. f can be computed by a Post–Turing program;
2. f can be computed by a Turing machine;
3. f can be computed by a quintuple Turing machine.

Proof of Theorem 1.4. Let \mathcal{M} be a given quintuple Turing machine with states q_1, \dots, q_K and alphabet $\{s_1, \dots, s_n\}$. We associate with each state q_i a label A_i and with each pair $q_i s_j$ a label B_{ij} . Each label A_i is to be placed next to the first instruction in the filter:

$[A_i]$	IF s_0 GOTO B_{i0}
	IF s_1 GOTO B_{i1}
	⋮
	IF s_n GOTO B_{in}

If \mathcal{M} contains the quintuple $q_i s_j s_k R q_l$, then we introduce the block of instructions

$[B_{ij}]$	PRINT s_k
	RIGHT
	GOTO A_l

Similarly, if \mathcal{M} contains the quintuple $q_i s_j s_k L q_l$, then we introduce the block of instructions:

$[B_{ij}]$	PRINT s_k
	LEFT
	GOTO A_l

Finally, if there is no quintuple in \mathcal{M} beginning $q_i s_j$, we introduce the block

$[B_{ij}]$	GOTO E
------------	----------

Then we can easily construct a Post–Turing program that simulates \mathcal{M} simply by putting all of these blocks and filters one under the other. The order is irrelevant except for one restriction: The filter labeled A_1 must begin the program. The entire program is listed in Figure 1.2. ■

$[A_1]$	IF s_0 GOTO B_{10} ⋮ IF s_n GOTO B_{1n}
$[A_2]$	IF s_0 GOTO B_{20} ⋮ IF s_n GOTO B_{2n}
$[A_K]$	IF s_0 GOTO B_{K0} ⋮ IF s_n GOTO B_{Kn}
$[B_{i_1j_1}]$	PRINT s_{k_1} RIGHT GOTO A_{l_1}
$[B_{i_2j_2}]$	PRINT s_{k_2} ⋮

Figure 1.2

Exercises

1. Let T be the Turing machine consisting of the quadruples

$$\begin{array}{cccc}
 q_1 & B & R & q_2 \\
 q_2 & 1 & R & q_3 \\
 q_3 & B & R & q_4 \\
 q_4 & 1 & B & q_1 \\
 q_4 & B & R & q_4 .
 \end{array}$$

For each integer x , let $g(x)$ be the number of occurrences of 1 on the tape when and if T halts when started with the read–write head one square to the left of the initial 1, with input $1^{[x]}$. What is the function $g(x)$?

2. Write out the quadruples constituting a Turing machine that computes the function

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$$

in base 1. Exhibit the state transition diagram for your machine.

3. Give precise definitions of *configuration*, *computation*, and *Turing machine \mathcal{M} computes the function f* . (Compare Chapter 2, Section 3.)

4. For each of the following functions, construct a Turing machine that computes the function strictly.
 - (a) $f(u, v) = \widehat{uv}$.
 - (b) $P(x) \Leftrightarrow x$ has an even number of symbols.
 - (c) $f(x)$ given in Exercise 1.4 in Chapter 5.
 - (d) $f(x) = x^R$. [x^R is defined in Chapter 1, Section 3.]
 - (e) $\#(u, v)$ given in Exercise 1.6 in Chapter 5.
5. Construct Turing machines for Exercise 4.4 in Chapter 5.
6. Construct a Turing machine for Exercise 4.5 in Chapter 5.
7. Using the construction in the proof of Theorem 1.1, transform the Post-Turing program in Figure 4.4 of Chapter 5 into an equivalent Turing machine.
8. Using the construction in the proof of Theorem 1.3, transform the Turing machine in Table 1.1 into an equivalent quintuple Turing machine.
9. Construct a quintuple Turing machine that computes $f(x, y) = x \div y$ in base 1 strictly.
- 10.* Show that any partially computable function can be computed by a quintuple Turing machine with two states. [Hint: A quintuple Turing machine \mathcal{M} with n states and m symbols (including s_0) can be simulated by a quintuple Turing machine \mathcal{M}' with two states and $4mn + m$ symbols. The $4mn$ new symbols represent the current state and currently scanned symbol of \mathcal{M} , as well as additional bookkeeping information. Transferring this stored information to an adjacent square can be done by a “loop” that moves the tape head back and forth.]

2. A Universal Turing Machine

Let us now recall the partially computable function $\Phi(x, z)$ from Chapter 4. For fixed z , $\Phi(x, z)$ is the unary partial function computed by the program whose number is z . Let \mathcal{M} be a Turing machine (in either quadruple or quintuple form) that computes this function with alphabet $\{1\}$. For reasons that we will explain, it is appropriate to call this machine \mathcal{M} *universal*.

Let $g(x)$ be any partially computable function of one variable and let z_0 be the number of some program in the language \mathcal{S} that computes g . Then

if we begin with a configuration

$$\begin{array}{c} B \ x \ B \ z_0 \\ \uparrow \\ q_1 \end{array}$$

(where x and z_0 are written as blocks of ones, i.e., in unary notation), and let \mathcal{M} proceed to compute, \mathcal{M} will compute $\Phi(x, z_0)$, i.e., $g(x)$. Thus, \mathcal{M} can be used to compute any partially computable function of one variable.

\mathcal{M} provides a suggestive model of an all-purpose computer, in which data and programs are stored together in a single “memory.” We can think of z_0 as a coded version of the program for computing g and x as the input to that program. Turing’s construction of a universal computer in 1936 provided reason to believe that, at least in principle, an all-purpose computer would be possible, and was thus an anticipation of the modern digital computer.

Exercises

- 1.* (a) Define a numbering $\#(\mathcal{M})$ of Turing machines like the numbering $\#(\mathcal{P})$ of \mathcal{S} programs given in Chapter 4.
 (b) Prove a version of the parameter theorem for Turing machines.
 (c) Prove a version of the recursion theorem for Turing machines.
 (d) Show that there is a Turing machine \mathcal{M} that prints $\#(\mathcal{M})$ when started with any input tape.
 (e) Show that Turing machines are an acceptable programming system. [Acceptable programming systems are defined in Exercise 5.4 in Chapter 4.]
- 2.* Give an upper bound on the size of the smallest universal Turing machine. [See Exercise 5.5 in Chapter 5.]

3. The Languages Accepted by Turing Machines

Given a Turing machine \mathcal{M} with alphabet $A = \{s_1, \dots, s_n\}$, a word $u \in A^*$ is said to be *accepted* by \mathcal{M} if when \mathcal{M} begins with the configuration

$$\begin{array}{c} s_0 \ u \\ \uparrow \\ q_1 \end{array}$$

it will eventually halt. The set of all words $u \in A^*$ that \mathcal{M} accepts is called the *language accepted by \mathcal{M}* . An important problem in the theory of computation involves characterizing the languages accepted by various kinds of computing devices. It is easy for us to solve this problem for Turing machines.

Theorem 3.1. A language is accepted by some Turing machine if and only if the language is r.e.

Proof. Let L be the language accepted by a Turing machine \mathcal{M} with alphabet A . Let $g(x)$ be the unary function on A^* that \mathcal{M} computes. Then g is a partially computable function (by Corollary 1.5 and by Theorem 6.2 in Chapter 5). Now,

$$L = \{x \in A^* \mid g(x) \downarrow\}. \quad (3.1)$$

Hence L is r.e.

Conversely, let L be r.e. Then there is a partially computable function $g(x)$ such that (3.1) holds. Using Theorem 1.2, let \mathcal{M} be a Turing machine with alphabet $\{s_1, \dots, s_n\}$ that computes $g(x)$ strictly. Then \mathcal{M} accepts L . ■

Naturally Theorem 3.1 is also true for quintuple Turing machines.

Let us consider the special case $A = \{1\}$. Then we have

Theorem 3.2. A set U of numbers is r.e. if and only if there is a Turing machine \mathcal{M} with alphabet $\{1\}$ that accepts $1^{[x]}$ if and only if $x \in U$.

Proof. This follows immediately from Theorem 3.1 and the fact that the base 1 representation of the number x is the string $1^{[x]}$. ■

This is an appropriate place to consider some annoying ambiguities in our notation of r.e. language. Thus, for example, consider the language

$$L_0 = \{a^{[n]} \mid n > 0\},$$

on the alphabet $\{a, b\}$. According to our definitions, to say that L_0 is an r.e. language is to say that the set of numbers which the strings in L_0 represent in base 2 is an r.e. set of numbers. But, *this set of numbers is not determined until an order is specified for the letters of the alphabet*. If we take a, b in the order shown, then the set of numbers which represent strings in L_0 is clearly

$$Q_1 = \{2^n - 1 \mid n > 0\},$$

while if we take the letters in the order b, a , the set of numbers which represents strings in L_0 is

$$Q_2 = \{2x \mid x \in Q_1\} = \{2^{n+1} - 2 \mid n > 0\}.$$

Now, although there is no difficulty whatever in showing that Q_1 and Q_2 are both r.e. sets, it is nevertheless a thoroughly unsatisfactory state of affairs to be forced to be concerned with such matters in asserting that L_0 is an r.e. language. Here Theorem 3.1 comes to our rescue. The notion of a given string being accepted by a Turing machine does not involve imposing any order on the symbols of the alphabet. Hence, Theorem 3.1 implies immediately that whether a particular language on a given alphabet is r.e. is *independent of how the symbols of the alphabet are ordered*. The same is clearly true of a language L on a given alphabet A being recursive since this is equivalent to L and $A^* - L$ both being r.e.

Another ambiguity arises from the fact that a particular language may be considered with respect to more than one alphabet. Thus, let A be an n -letter alphabet and let \tilde{A} be an m -letter alphabet containing A , so that $m > n$. Then a language L on the alphabet A is simply some subset of A^* , so that L is also a language on the larger alphabet \tilde{A} . Thus, depending on whether we are thinking of L as a language on A or as a language on \tilde{A} , we will have to read the strings in L as being the notation for integers in *base n or in base m, respectively*. Hence, we are led to the unpleasant possibility that whether L is r.e. might actually depend on which alphabet we are considering. As an example, we may take $A = \{a\}$ and $\tilde{A} = \{a, b\}$, and consider the language L_0 above, where

$$L_0 \subseteq A^* \subseteq \tilde{A}^*.$$

We have already seen that our original definition of L_0 's being r.e. as a language on the alphabet \tilde{A} amounts to requiring that the set of numbers Q_1 or Q_2 (depending on the order of the symbols a, b) be r.e. However, if we take our alphabet to be A , then the relevant set of numbers is

$$Q_3 = \{n \in N \mid n > 0\}.$$

We remove all such ambiguities by proving

Theorem 3.3. Let $A \subseteq \tilde{A}$ where A and \tilde{A} are alphabets and let $L \subseteq A^*$. Then L is an r.e. language on the alphabet A if and only if L is an r.e. language on \tilde{A} .

Proof. Let L be r.e. on A and let \mathcal{M} be a Turing machine with alphabet A that accepts L . Without loss of generality, we can assume that \mathcal{M} begins

by moving right until it finds a blank and then returns to its original position. Let $\tilde{\mathcal{M}}$ be obtained from \mathcal{M} by adjoining to it the quadruples $qssq$ for each symbol $s \in \tilde{A} - A$, and each state q of \mathcal{M} . Thus $\tilde{\mathcal{M}}$ will enter an “infinite loop” if it ever encounters a symbol in $\tilde{A} - A$. Since $\tilde{\mathcal{M}}$ has alphabet \tilde{A} and accepts the language L , we conclude from Theorem 3.1 that L is an r.e. language on \tilde{A} .

Conversely, let L be r.e. as a language on \tilde{A} , and let \mathcal{M} be a Turing machine with alphabet \tilde{A} that accepts L . Let $g(x)$ be the function on A^* that \mathcal{M} computes. (The symbols belonging to $\tilde{A} - A$ thus serve as “markers.”) Since $L \subseteq A^*$, we have

$$L = \{x \in A^* \mid g(x) \downarrow\}.$$

Since $g(x)$ is partially computable, it follows that L is an r.e. language on A . ■

Corollary 3.4. Let A, \tilde{A}, L be as in Theorem 3.3. Then L is a recursive language on A if and only if L is a recursive language on \tilde{A} .

Proof. First let L be a recursive language on A . Then L and $A^* - L$ are r.e. languages on A and therefore on \tilde{A} . Moreover, since

$$\tilde{A}^* - L = (\tilde{A}^* - A^*) \cup (A^* - L),$$

and since $\tilde{A}^* - A^*$ is r.e., as the reader can easily show (see Exercise 6), it follows from Theorem 4.5 in Chapter 4 that $A^* - L$ is r.e. Hence, L is a recursive language on \tilde{A} .

Conversely, if L is a recursive language on \tilde{A} , then L and $\tilde{A}^* - L$ are r.e. languages on \tilde{A} and therefore L is an r.e. language on A . Moreover, since

$$A^* - L = (\tilde{A}^* - L) \cap A^*,$$

and since A^* is obviously r.e. (as a language on A and therefore on \tilde{A}), it follows from Theorem 4.5 in Chapter 4 that $A^* - L$ is an r.e. language on \tilde{A} and hence on A . Thus, L is a recursive language on A . ■

Exercises

1. Write out the quadruples constituting a Turing machine that accepts the language consisting of all words on the alphabet $\{a, b\}$ of the form $a^{[i]}ba^{[i]}$.
2. Give a Turing machine that accepts $\{1^{[i]}B1^{[j]}B1^{[i+j]} \mid i, j \in N\}$.
3. Give a Turing machine that accepts $\{w \in \{a, b\}^* \mid w = w^R\}$.

4. Show that there is a Turing machine that accepts the language $\{1^{[x]}B1^{[y]} \mid \Phi_y(x) \downarrow\}$.
5. Show that there is no Turing machine that accepts the language $\{1^{[y]} \mid \Phi_y(x) \downarrow \text{ for all } x \in N\}$.
6. Complete the proof of Corollary 3.4 by showing that $\tilde{A}^* - A^*$ is an r.e. language.

4. The Halting Problem for Turing Machines

We can use the results of the previous section to obtain a sharpened form of the unsolvability of the halting problem.

By the halting problem for a *fixed* given Turing machine \mathcal{M} we mean the problem of finding an algorithm *to determine whether \mathcal{M} will eventually halt starting with a given configuration*. We have

Theorem 4.1. There is a Turing machine \mathcal{K} with alphabet {1} that has an unsolvable halting problem.

Proof. Take for the set U in Theorem 3.2, some r.e. set that is not recursive (e.g., the set K from Chapter 4). Let \mathcal{K} be the corresponding Turing machine. Thus \mathcal{K} accepts a string of ones if and only if its length belongs to U . Hence, $x \in U$ if and only if \mathcal{K} eventually halts when started with the configuration

$$\begin{array}{c} B \quad 1^{[x]} \\ \uparrow \\ q_1 \end{array}$$

Thus, if there were an algorithm for solving the halting problem for \mathcal{K} , it could be used to test a given number x for membership in U . Since U is not recursive, such an algorithm is impossible. ■

This is really a stronger result than was obtained in Chapter 4. What we can prove about Turing machines just using Theorem 2.1 in Chapter 4 is that there is no algorithm that can be used, given a Turing machine and an initial configuration, to determine whether the Turing machine will ever halt. Our present result gives a *fixed* Turing machine whose halting problem is unsolvable. Actually, this result could also have been easily obtained from the earlier one by using a universal Turing machine.

Next, we show how the unsolvability of the halting problem can be used to obtain another unsolvable problem concerning Turing machines. We begin with a Turing machine \mathcal{K} with alphabet {1} that has an unsolvable

halting problem. Let the states of \mathcal{K} be q_1, \dots, q_k . We will construct a Turing machine $\tilde{\mathcal{K}}$ by adjoining to the quadruples of \mathcal{K} the following quadruples:

$$q_i \ B \ B \ q_{k+1}$$

for $i = 1, 2, \dots, k$ for which no quadruple of \mathcal{K} begins $q_i B$, and

$$q_i \ 1 \ 1 \ q_{k+1}$$

for $i = 1, 2, \dots, k$ when no quadruple of \mathcal{K} begins $q_i 1$. Thus, \mathcal{K} eventually halts beginning with a given configuration if and only if $\tilde{\mathcal{K}}$ eventually is in state q_{k+1} . We conclude

Theorem 4.2. There is a Turing machine $\tilde{\mathcal{K}}$ with alphabet $\{1\}$ and a state q_m such that there is no algorithm that can determine whether $\tilde{\mathcal{K}}$ will ever arrive at state q_m when it begins at a given configuration.

Exercises

1. Prove that there is a Turing machine \mathcal{M} such that there is no algorithm that can determine of a given configuration whether \mathcal{M} will eventually halt with a completely blank tape when started with the given tape configuration.
2. Prove that there is a Turing machine \mathcal{M} with alphabet $\{s_1, s_2\}$ such that there is no algorithm that can determine whether \mathcal{M} starting with a given configuration will ever print the symbol s_2 .
3. Let $\mathcal{M}_0, \mathcal{M}_1, \dots$ be a list of all Turing machines, and let f_i be the unary partial function computed by \mathcal{M}_i , $i = 0, 1, \dots$. Suppose $g(x)$ is a total function such that for all $x \geq 0$ and all $0 \leq i \leq x$, if $f_i(x) \downarrow$ then $f_i(x) < g(x)$. Show that $g(x)$ is not computable.
4. Jill and Jack have been working as programmers for a year. They are discussing their work. We listen in:

JACK: We are working on a wonderful program, AUTOCHECK.

AUTOCHECK will accept Pascal programs as inputs and will return the values OK or LOOPS depending on whether the given program is or is not free of infinite loops.

JILL: Big deal! We have a mad mathematician in our firm who has developed an algorithm so complicated that no program can be written to execute it no matter how much space and time is allowed.

Comment on and criticize Jack and Jill's statements.

5. Nondeterministic Turing Machines

As already mentioned, a *nondeterministic Turing machine* is simply an arbitrary finite set of quadruples. Thus, what we have been calling a Turing machine is simply a special kind of nondeterministic Turing machine. For emphasis, we will sometimes refer to ordinary Turing machines as *deterministic*.

A configuration

$$\dots \ s_j \ \dots \\ \uparrow \\ q_i$$

is called *terminal* with respect to a given nondeterministic Turing machine (and the machine is said to *halt*) if it contains no quadruple beginning $q_i s_j$. (This, of course, is exactly the same as for deterministic Turing machines.) We use the symbol \vdash (borrowed from logic) placed between a pair of configurations to indicate that the transition from the configuration on the left to the one on the right is permitted by one of the quadruples of the machine under consideration.

As an example, consider the nondeterministic Turing machine given by the quadruples

q_1	B	R	q_2
q_2	1	R	q_3
q_2	B	B	q_4
q_3	1	R	q_2
q_3	B	B	q_3
q_4	B	R	q_4
q_4	B	B	q_5

Then we have

$$B \ 1 \ 1 \ 1 \ 1 \vdash B \ 1 \ 1 \ 1 \ 1 \vdash B \ 1 \ 1 \ 1 \ 1 \vdash B \ 1 \ 1 \ 1 \ 1 \vdash B \ 1 \ 1 \ 1 \ 1 \\ \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \\ q_1 \qquad \qquad q_2 \qquad \qquad q_3 \qquad \qquad q_2 \qquad \qquad q_3$$

$$\vdash B \ 1 \ 1 \ 1 \ 1 \ B \ B \vdash B \ 1 \ 1 \ 1 \ 1 \ B. \\ \uparrow \qquad \qquad \qquad \uparrow \\ q_2 \qquad \qquad \qquad q_4$$

So far the computation has been entirely determined; however, at this point the nondeterminism plays a role. We have

$$\begin{array}{c} B \ 1 \ 1 \ 1 \ 1 \ B \vdash B \ 1 \ 1 \ 1 \ 1 \ B, \\ \uparrow \qquad \qquad \qquad \uparrow \\ q_4 \qquad \qquad \qquad q_5 \end{array}$$

at which the machine halts. But we also have

$$\begin{array}{c} B \ 1 \ 1 \ 1 \ 1 \ B \vdash B \ 1 \ 1 \ 1 \ 1 \ B \ B \vdash B \ 1 \ 1 \ 1 \ 1 \ B \ B \ B \vdash \cdots . \\ \uparrow \qquad \qquad \qquad \uparrow \qquad \qquad \qquad \uparrow \\ q_4 \qquad \qquad \qquad q_4 \qquad \qquad \qquad q_4 \end{array}$$

Let $A = \{s_1, \dots, s_n\}$ be a given alphabet and let $u \in A^*$. Then the nondeterministic Turing machine \mathcal{M} is said to *accept* u if there exists a sequence of configurations $\gamma_1, \gamma_2, \dots, \gamma_m$ such that γ_1 is the configuration

$$\begin{array}{c} s_0 \ u \\ \uparrow \\ q_1 \end{array}$$

γ_m is terminal with respect to \mathcal{M} , and $\gamma_1 \vdash \gamma_2 \vdash \gamma_3 \vdash \cdots \vdash \gamma_m$. In this case, the sequence $\gamma_1, \gamma_2, \dots, \gamma_m$ is called an *accepting computation* by \mathcal{M} for u . If A is the alphabet of \mathcal{M} , then the *language accepted* by \mathcal{M} is the set of all $u \in A^*$ that are accepted by \mathcal{M} .

Of course, for deterministic Turing machines, this definition gives nothing new. However, it is important to keep in mind the distinctive feature of acceptance by nondeterministic Turing machines. It is perfectly possible to have an infinite sequence

$$\gamma_1 \vdash \gamma_2 \vdash \gamma_3 \vdash \cdots$$

of configurations, where γ_1 is

$$\begin{array}{c} s_0 \ u \\ \uparrow \\ q_1 \end{array}$$

even though u is accepted by \mathcal{M} . It is only necessary that there be *some* sequence of transitions leading to a terminal configuration. One sometimes expresses this by saying, “The machine is always permitted to guess the correct next step.”

Thus in the example given above, taking the alphabet $A = \{1\}$, we have that \mathcal{M} accepts 1111. In fact the language accepted by \mathcal{M} is $\{1^{[2^n]}\}$. (See Exercise 3.)

Since a Turing machine is also a nondeterministic Turing machine, Theorem 3.1 can be weakened to give

Theorem 5.1. For every r.e. language L , there is a nondeterministic Turing machine \mathcal{M} that accepts L .

The converse is also true: the language accepted by a nondeterministic Turing machine must be r.e. By Church's thesis, it is clear that this should be true. It is only necessary to "run" a nondeterministic Turing machine \mathcal{M} on a given input u , *following all alternatives at each step*, and giving the value (say) 0, if termination is reached along any branch. This defines a function that is *intuitively* partially computable and whose domain is the language accepted by \mathcal{M} . However, a detailed proof along these lines would be rather messy.

Fortunately the converse of Theorem 5.1 will be an easy consequence of the methods we will develop in the next chapter.

Exercises

1. Explain why nondeterministic Turing machines are unsuitable for defining functions.
2. Let L be the set of all words on the alphabet $\{a, b\}$ that contain at least two consecutive occurrences of b . Construct a nondeterministic Turing machine that *never moves left* and accepts L .
3. Show that the nondeterministic Turing machine \mathcal{M} used as an example in this section accepts the set $\{1^{[2^n]}\}$.
4. Let

$$L_1 = \{w \in \{a, b\}^* \mid w \text{ has an even number of } a\text{'s}\},$$

$$L_2 = \{w \in \{a, b\}^* \mid w \text{ has an odd number of } b\text{'s}\}.$$

- (a) Give deterministic Turing machines $\mathcal{M}_1, \mathcal{M}_2$ that accept L_1, L_2 , respectively, and combine them to get a nondeterministic Turing machine that accepts $L_1 \cup L_2$.
- (b) Give a deterministic Turing machine that accepts $L_1 \cup L_2$.
5. Give a nondeterministic Turing machine that accepts $\{1^{[n]} \mid n \text{ is prime}\}$.
6. If we replace "the first instruction labeled L " by "some instruction labeled L " in the interpretation of Post–Turing instructions of the form IF σ GOTO L , then we get *nondeterministic Post–Turing programs*. Show that a language is accepted by a nondeterministic Post–

Turing program if and only if it is accepted by a nondeterministic Turing machine (where acceptance of a language by a Post–Turing program is defined just like acceptance by a Turing machine).

6. Variations on the Turing Machine Theme

So far we have three somewhat different formulations of Turing’s conception of computation: the Post–Turing programming language, Turing machines as made up of quadruples, and quintuple Turing machines. The proof that these formulations are equivalent was quite simple. This is true in part because all three involved a single tapehead on a single two-way infinite tape. But it is easy to imagine other arrangements. In fact, Turing’s original formulation was in terms of a tape that was infinite in only one direction, that is, with a first or leftmost square (see Fig. 6.1). We can also think of permitting several tapes, each of which can be one-way or two-way infinite and each with its own tapehead. There might even be several tapeheads per tape. As one would expect, programs can be shorter when several tapes are available. But, if we believe Church’s thesis, we certainly would expect all of these formulations to be equivalent. In this section we will indicate briefly how this equivalence can be demonstrated.

Let us begin by considering one-way infinite tapes. To make matters definite, we assume that we are representing a Turing machine as a set of quadruples. It is necessary to make a decision about the effect of a quadruple $q_i s_j L q_k$ in case the tapehead is already at the left end of the tape. There are various possibilities, and it really does not matter very much which we adopt. For definiteness we assume that an instruction to move left will be interpreted as a *halt* in case the tapehead is already at the leftmost square. Now it is pretty obvious that anything that a Turing machine could do on a one-way infinite tape could also be done on a two-way infinite tape, and we leave details to the reader.

How can we see that any partially computable function can be computed by a Turing machine on a one-way infinite tape? One way is by simply examining the proof of Theorem 5.1 in Chapter 5, which shows how a

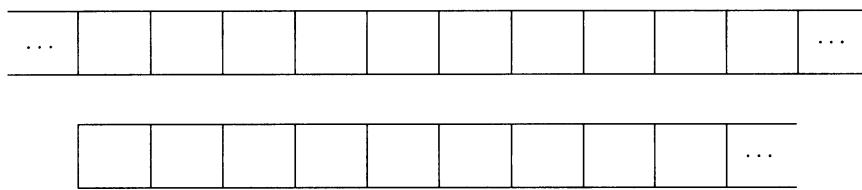


Figure 6.1. Two-way infinite versus one-way infinite tape.

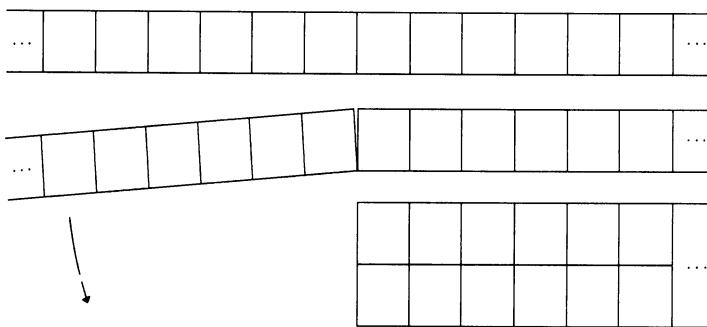


Figure 6.2

computation in any of the languages \mathcal{S}_n can be simulated by a program in the Post–Turing language \mathcal{T} . In fact, the program \mathcal{Q} in the language \mathcal{T} which is constructed to simulate a given program \mathcal{P} in the language \mathcal{S}_n has the particular property that when \mathcal{Q} is executed, the tapehead never moves to the left of the square initially scanned. Hence, the program \mathcal{Q} would work exactly as well on a one-way infinite tape whose leftmost square is initially scanned. And, it is an easy matter, as in the proof of Theorem 1.1, to convert \mathcal{Q} into a Turing machine.

Although this is an entirely convincing argument, we would like to mention another approach which is interesting in its own right, namely, we directly face the question, how can the information contained in a two-way infinite tape be handled by a Turing machine with one tapehead on a one-way infinite tape? The intuitive idea is to think of a two-way infinite tape as being “hinged” so it can be folded as in Fig. 6.2. Thus our two-way infinite tape can be represented by a one-way infinite tape with two “tracks,” an “upper” and a “lower.” Moreover, by adding enough symbols to the alphabet, we can code each pair consisting of an upper and a lower symbol by a single symbol.

Thus, let us begin with a Turing machine \mathcal{M} with alphabet $A = \{s_1, \dots, s_n\}$ and states q_1, \dots, q_K . Let \mathcal{M} compute a unary¹ partial function g on A_0^* , where $A_0 \subseteq A$. Thus the input configuration when \mathcal{M} is computing $g(x)$ for $x \in A_0^*$ will be

$B \quad x$
 ↑
 q_1

¹ The restriction to unary functions is, of course, not essential.

We will construct a Turing machine \bar{M} that computes g on a one-way infinite tape. The initial configuration for \bar{M} will be

$$\begin{array}{c} \# \ B \ x \\ \uparrow \\ q_1 \end{array}$$

where $\#$ is a special symbol that will occupy the leftmost square on the tape for most of the computation. The alphabet of \bar{M} will be

$$A \cup \{\#\} \cup \{b_j^i \mid 0 \leq i, j \leq n\},$$

where we think of the symbol b_j^i as indicating that s_i is on the upper track and s_j is on the lower track. The states of \bar{M} are q_1, q_2, q_3, q_4, q_5 , and

$$\{\bar{q}_i, \tilde{q}_i \mid i = 1, 2, \dots, K\}$$

as well as certain additional states.

We can think of the quadruples constituting \bar{M} as made up of three sections: BEGINNING, MIDDLE, and END. BEGINNING serves to copy the input on the upper track putting blanks on the corresponding lower track of each square. BEGINNING consists of the quadruples

$$\begin{array}{cccccc} q_1 & B & R & q_2 & & \\ q_2 & s_i & R & q_2 & i = 1, 2, \dots, n, & \\ q_2 & B & L & q_3 & & \\ q_3 & s_i & b_0^i & q_3 & i = 0, 1, 2, \dots, n, & \\ q_3 & b_0^i & L & q_3 & i = 0, 1, 2, \dots, n, & \\ q_3 & \# & R & \bar{q}_1. & & \end{array}$$

Thus, starting with the configuration

$$\begin{array}{c} \# \ B \ s_2 \ s_1 \ s_3 \\ \uparrow \\ q_1 \end{array}$$

BEGINNING will halt in the configuration

$$\begin{array}{c} \# \ b_0^0 \ b_0^2 \ b_0^1 \ b_0^3 \ B. \\ \uparrow \\ \bar{q}_1 \end{array}$$

Note that b_0^0 is different from $s_0 = B$. MIDDLE will consist of quadruples corresponding to those of M as well as additional quadruples as indicated

Table 6.1

Quadruple of \mathcal{M}				Quadruple of $\bar{\mathcal{M}}$			
(a) $q_i \ s_j \ s_k \ q_l$	\bar{q}_i	b_m^j	b_m^k	\bar{q}_l		$m = 0, 1, \dots, n$	
	\tilde{q}_i	b_j^m	b_k^m	\tilde{q}_l		$m = 0, 1, \dots, n$	
(b) $q_i \ s_j \ R \ q_l$	\bar{q}_i	b_m^j	R	\bar{q}_l		$m = 0, 1, \dots, n$	
	\tilde{q}_i	b_j^m	L	\tilde{q}_l		$m = 0, 1, \dots, n$	
(c) $q_i \ s_j \ L \ q_l$	\bar{q}_i	b_m^j	L	\bar{q}_l		$m = 0, 1, \dots, n$	
	\tilde{q}_i	b_j^m	R	\tilde{q}_l		$m = 0, 1, \dots, n$	
(d) —————	\bar{q}_i	B	b_0^0	\bar{q}_i		$i = 1, 2, \dots, K$	
	\tilde{q}_i	B	b_0^0	\bar{q}_i		$i = 1, 2, \dots, K$	
(e) —————	\bar{q}_i	#	R	\tilde{q}_i		$i = 1, 2, \dots, K$	
	\tilde{q}_i	#	R	\bar{q}_i		$i = 1, 2, \dots, K$	

in Table 6.1. The states \bar{q}_i, \tilde{q}_i correspond to actions on the upper track and lower track, respectively. Note in (b) and (c) that on the lower track left and right are reversed. The quadruples in (d) replace single blanks B by double blanks b_0^0 as needed. The quadruples (e) arrange for switchover from the upper to the lower track. It should be clear that MIDDLE simulates \mathcal{M} .

END has the task of translating the output into a word on the original alphabet A . This task is complicated by the fact that the output is split between the two tracks. To begin with, END contains the following quadruples:

$$\left. \begin{array}{ll} \bar{q}_i & b_m^j \quad b_m^j \quad q_4 \\ \tilde{q}_i & b_j^m \quad b_j^m \quad q_4 \end{array} \right\} \begin{array}{l} \text{whenever } \mathcal{M} \text{ contains no quadruple} \\ \text{beginning } q_i s_j, \text{ for } m = 0, 1, \dots, n; 0 \leq i, j \leq n, \end{array}$$

$$\begin{array}{ll} q_4 & b_j^i \quad L \quad q_4, \\ q_4 & \# \quad B \quad q_5. \end{array}$$

For each initial configuration for which \mathcal{M} halts, the effect of BEGINNING, MIDDLE, and this part of END is to ultimately produce a configuration of the form

$$B \ b_{j_1}^{i_1} \ b_{j_2}^{i_2} \ \cdots \ b_{j_k}^{i_k}.$$

\uparrow
 q_5

The remaining task of END is to convert the tape contents into

$$s_{j_k} \ s_{j_{k-1}} \ \cdots \ s_{j_1} \ s_{i_1} \ s_{i_2} \ \cdots \ s_{i_k}.$$

[D]	RIGHT TO NEXT BLANK MOVE BLOCK RIGHT
[C]	RIGHT RIGHT IF b_j^i GOTO A_j^i ($0 \leq i, j \leq n$) IF B GOTO F GOTO C
$[A_j^i]$	PRINT s_i ($0 < i \leq n, 0 \leq j \leq n$) GOTO B_j
$[A_j^0]$	PRINT # ($0 \leq j \leq n$) GOTO B_j
$[B_j]$	LEFT TO NEXT BLANK ($0 < j \leq n$) PRINT s_j GOTO D
$[B_0]$	LEFT TO NEXT BLANK PRINT # GOTO D
[F]	LEFT IF s_j GOTO F ($0 < j \leq n$) IF # GOTO G IF B GOTO E
[G]	PRINT B GOTO F

Figure 6.3

Instead of giving quadruples for accomplishing this, we exhibit a program in the Post–Turing language \mathcal{T} , so that we can make use of some of the macros available in that language. Of course, this program can easily be translated into a set of quadruples using the method of proof of Theorem 1.1. Because our macros for \mathcal{T} were designed for use with “blocks” of symbols containing no blanks, we will use # instead of $s_0 = B$ in carrying out our translation. One final pass will be needed to replace each # by B . The program is given in Fig. 6.3.

Each b_j^i is processed going from left to right. b_j^i is replaced by s_i (or by # if $i = 0$) and s_j (or # if $j = 0$) is printed on the left. The “MOVE BLOCK RIGHT” macro is used to make room on the tape for printing the successive symbols from the “lower” track. As an example let us apply the program of Fig. 6.3 to the configuration

$$B \ b_1^2 \ b_1^0 \ b_0^1. \\ \uparrow$$

B	b_1^2	b_1^0	b_0^1				D	
\uparrow								
B	B	b_1^2	b_1^0	b_0^1	B		C	
\uparrow								
B	B	b_1^2	b_1^0	b_0^1	B		A_1^2	
\uparrow								
B	B	s_2	b_1^0	b_0^1	B		B_1	
\uparrow								
B	s_1	s_2	b_1^0	b_0^1	B		D	
\uparrow								
B	B	s_1	s_2	b_1^0	b_0^1	B	A_1^0	
\uparrow								
B	s_1	s_1	s_2	#	b_0^1	B	D	
\uparrow								
B	#	s_1	s_1	s_2	#	s_1	B	D
\uparrow								
B	B	#	s_1	s_1	s_2	#	s_1	B
								F
B	B	B	s_1	s_1	s_2	B	s_1	B
								E

Figure 6.4

In Fig. 6.4 we show various stages in the computation; in each case the tape configuration is followed by the label on the next instruction to be executed.

The technique of thinking of the tape of a Turing machine as decomposed into a number of parallel tracks has numerous uses. (It will appear again in Chapter 11.) For the moment we note that it can be used to simulate the behavior of a multitape Turing machine by an ordinary Turing machine. For, in the first place a second track can be used to show the position of a tapehead on a one-tape machine as in the example shown in Fig. 6.5; the 1 under the s_3 shows the position of the head. In an entirely similar manner the contents of k tapes and the position of the tapehead on each can be represented as a single tape with $2k$ tracks. Using this representation, it is easy to see how to simulate any computation by a k -tape Turing machine using only one tape. The same result can also be obtained indirectly using the method of proof of Theorem 6.1 in Chapter 5 to show that any function computed by a k -tape Turing machine is partially computable.

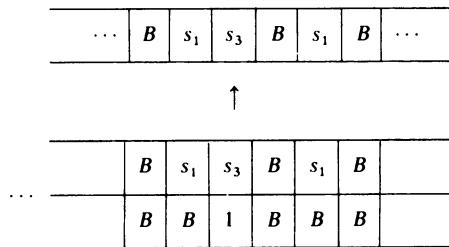


Figure 6.5

Exercises

1. Give a formal description of a Turing machine that uses three tapes: one with a “read only” head for input, one with a “write only” head for output, and one for “working.” Give an appropriate definition of computability by such machines and prove the equivalence with computability by ordinary Turing machines.
2. Do the same for a Turing machine with input tape, output tape, and k working tapes for any $k \geq 1$.
3. Let the Post–Turing language be augmented by the instructions UP, DOWN so that it can deal with computations on a two-dimensional “tape” infinite in all four directions. Supply an appropriate definition of what it means to compute a function by a program in this language, and prove that any function computed by such a program is partially computable.
4. Adapt the construction in this section so that it works for binary functions.

7

Processes and Grammars

1. Semi-Thue Processes

In this chapter we will see how the methods of computability theory can be used to deal with combinatorial problems involving substitution of one substring for another in a string.

Definition. Given a pair of words g, \bar{g} on some alphabet, the expression

$$g \rightarrow \bar{g}$$

is called a *semi-Thue production* or simply a *production*. The term *rewrite rule* is also used.

Thue is from Axel Thue, a Norwegian mathematician, and is pronounced too-ay.

If P is the semi-Thue production $g \rightarrow \bar{g}$, then we write

$$u \xrightarrow{P} v$$

to mean that there are (possibly null) words r, s such that

$$u = rgs \quad \text{and} \quad v = r\bar{g}s.$$

(In other words, v is obtained from u by a replacement of g by \bar{g} .)

Definition. A *semi-Thue process* is a finite set of semi-Thue productions.

If Π is a semi-Thue process, we write

$$u \xrightarrow{\Pi} v$$

to mean that

$$u \xrightarrow{P} v$$

for some production P which belongs to Π . Finally, we write

$$u \xrightarrow{* \Pi} v$$

if there is a sequence

$$u = u_1 \xrightarrow{\Pi} u_2 \xrightarrow{\Pi} \cdots \xrightarrow{\Pi} u_n = v.$$

The sequence u_1, u_2, \dots, u_n is then called a *derivation of v from u* . In particular (taking $n = 1$)

$$u \xrightarrow{* \Pi} u.$$

When no ambiguity results we often omit the explicit reference to Π , writing simply $u \Rightarrow v$ and $u \xrightarrow{*} v$.

Here is a simple example: We let $\Pi = \{ab \rightarrow aa, ba \rightarrow bb\}$. Then we have

$$aba \Rightarrow abb \Rightarrow aab \Rightarrow aaa.$$

Thus,

$$aba \xrightarrow{*} aaa,$$

and the sequence of words aba, abb, aab, aaa is a derivation of aaa from aba .

Exercises

- Let Π be the semi-Thue process with the production $ba \rightarrow ab$.
 - Give two different derivations of $aaabb$ from $abbaba$.
 - Give the set of all words in $\{a, b\}^*$ from which $aabb$ can be derived.
 - Give the set of all words which can be derived from $bbaa$.
- Let Π be the semi-Thue process with productions $ba \rightarrow ab$ and $ab \rightarrow ba$. Show that for all words $u, v \in \{a, b\}^*$, $u \xrightarrow{* \Pi} v$ if and only if $v \xrightarrow{* \Pi} u$.

3. Give a semi-Thue process Π such that $1^{[x]} \xrightarrow[\Pi]{} 1^{[y]}$ if and only if $|x - y|$ is even.
4. Let $A = \{1, 2, b_1^1, b_2^1, b_1^2, b_2^2, c_1, c_2, d_1, d_2\}$. Give a semi-Thue process Π such that $b_{j_1}^{i_1} \cdots b_{j_n}^{i_n} \xrightarrow[\Pi]{} w \in \{1, 2\}^*$, for all words $b_{j_1}^{i_1} \cdots b_{j_n}^{i_n}$, where $i_1 \cdots i_n, j_1 \cdots j_n$ are binary representations of numbers and $i_1 \cdots i_n + j_1 \cdots j_n = w$. [Hint: The symbols c_1, c_2 are used to remember the need to carry 1, and d_1, d_2 are used to remember the need to carry 2.]

2. Simulation of Nondeterministic Turing Machines by Semi-Thue Processes

Let us begin with a nondeterministic Turing machine \mathcal{M} with alphabet $\{s_1, \dots, s_K\}$, and states q_1, q_2, \dots, q_n . We shall show how to simulate \mathcal{M} by a semi-Thue process $\Sigma(\mathcal{M})$ on the alphabet

$$s_0, s_1, \dots, s_K, q_0, q_1, q_2, \dots, q_n, q_{n+1}, h.$$

Each stage in a computation by \mathcal{M} is specified completely by the current configuration. We shall code each such stage by a word on the alphabet of $\Sigma(\mathcal{M})$. For example, the configuration

$$\begin{array}{ccccccccc} s_1 & s_1 & s_3 & s_2 & s_0 & s_1 & s_2 \\ & & & & \uparrow & & \\ & & & & q_4 & & \end{array}$$

will be represented by the single word

$$hs_1s_1s_3q_4s_2s_0s_1s_2h. \quad (2.1)$$

Note that h is used as a *beginning* and *end* marker, and the symbol q_4 indicates the state of \mathcal{M} and is placed immediately to the left of the scanned square. A word like (2.1) will be called a *Post word*. Of course, the same configuration can be represented by infinitely many Post words because any number of additional blanks may be shown on the left or right. For example,

$$hs_0s_0s_1s_1s_3q_4s_2s_0s_1s_2s_0h$$

is a Post word representing the same configuration that (2.1) does.

In general, a word huq_ivh , where $0 \leq i \leq n + 1$, is called a *Post word* if u and v are words on the subalphabet $\{s_0, s_1, \dots, s_K\}$. We shall show how to associate suitable semi-Thue productions with each quadruple of \mathcal{M} ; the productions simulate the effect of that quadruple on Post words.

- For each quadruple of \mathcal{M} of the form $q_i s_j s_k q_l$ we place in $\Sigma(\mathcal{M})$ the production

$$q_i s_j \rightarrow q_l s_k.$$

- For each quadruple of \mathcal{M} of the form $q_i s_j R q_l$ we place in $\Sigma(\mathcal{M})$ the productions

$$q_i s_j s_k \rightarrow s_j q_l s_k, \quad k = 0, 1, \dots, K,$$

$$q_i s_j h \rightarrow s_j q_l s_0 h.$$

- For each quadruple of \mathcal{M} of the form $q_i s_j L q_l$ we place in $\Sigma(\mathcal{M})$ the productions

$$s_k q_i s_j \rightarrow q_l s_k s_j, \quad k = 0, 1, \dots, K,$$

$$h q_i s_j \rightarrow h q_l s_0 s_j.$$

To see how these productions simulate the behavior of \mathcal{M} , suppose \mathcal{M} is in configuration

$$\begin{array}{cccc} s_2 & s_1 & s_0 & s_3 \\ & \uparrow & & \\ & q_4 & & \end{array}$$

This configuration is represented by the Post word

$$hs_2 q_4 s_1 s_0 s_3 h.$$

Now suppose \mathcal{M} contains the quadruple

$$q_4 \ s_1 \ s_3 \ q_5.$$

Then $\Sigma(\mathcal{M})$ contains the production

$$q_4 s_1 \rightarrow q_5 s_3,$$

so that

$$hs_2 q_4 s_1 s_0 s_3 h \xrightarrow{\Sigma(\mathcal{M})} hs_2 q_5 s_3 s_0 s_3 h.$$

The Post word on the right then corresponds to the configuration immediately following application of the above quadruple. Now suppose that \mathcal{M} contains the quadruple

$$q_4 \ s_1 \ R \ q_3.$$

(Of course, if \mathcal{M} is a *deterministic* Turing machine, it cannot contain both of these quadruples.) Then $\Sigma(\mathcal{M})$ contains the production

$$q_4 s_1 s_0 \rightarrow s_1 q_3 s_0,$$

so that

$$hs_2 q_4 s_1 s_0 s_3 h \xrightarrow{\Sigma(\mathcal{M})} hs_2 s_1 q_3 s_0 s_3 h.$$

Finally if \mathcal{M} contains the quadruple

$$q_4 \ s_1 \ L \ q_2,$$

then $\Sigma(\mathcal{M})$ contains the production

$$s_2 q_4 s_1 \rightarrow q_2 s_2 s_1,$$

so that

$$hs_2 q_4 s_1 s_0 s_3 h \xrightarrow{\Sigma(\mathcal{M})} hq_2 s_2 s_1 s_0 s_3 h.$$

The productions involving h are to take care of cases where motion to the right or left would go past the part of the tape included in the Post word, so that an additional blank must be added. For example, if the configuration is

$$\begin{array}{c} s_2 \ s_3 \ s_1 \\ \uparrow \\ q_4 \end{array}$$

and \mathcal{M} contains the quadruple

$$q_4 \ s_1 \ R \ q_3,$$

then $\Sigma(\mathcal{M})$ contains the production

$$q_4 s_1 h \rightarrow s_1 q_3 s_0 h$$

and we have

$$hs_2 s_3 q_4 s_1 h \xrightarrow{\Sigma(\mathcal{M})} hs_2 s_3 s_1 q_3 s_0 h,$$

so that the needed blank on the right has been inserted. The reader will readily verify that blanks on the left are similarly supplied when needed.

We now complete the specification of $\Sigma(\mathcal{M})$:

4. Whenever $q_i s_j (i = 1, \dots, n; j = 0, 1, \dots, K)$ are *not* the first two symbols of a quadruple of \mathcal{M} , we place in $\Sigma(\mathcal{M})$ the production

$$q_i s_j \rightarrow q_{n+1} s_j.$$

Thus, q_{n+1} serves as a “halt” state.

5. Finally, we place in $\Sigma(\mathcal{M})$ the productions

$$\begin{aligned} q_{n+1}s_i &\rightarrow q_{n+1}, \quad i = 0, 1, \dots, K, \\ q_{n+1}h &\rightarrow q_0h \\ s_iq_0 &\rightarrow q_0, \quad i = 0, 1, \dots, K. \end{aligned}$$

We have

Theorem 2.1. Let \mathcal{M} be a deterministic Turing machine, and let w be a Post word on the alphabet of $\Sigma(\mathcal{M})$. Then

1. there is at most one word z such that $w \xrightarrow{\Sigma(\mathcal{M})} z$, and
2. if there is a word z satisfying (1), then z is a Post word.

Proof. We have $w = huq_ivh$.

If $1 \leq i \leq n$, then

- a. if $v = 0$ no production of $\Sigma(\mathcal{M})$ applies to w ;
- b. if v begins with the symbol s_j and there is a (necessarily unique) quadruple of \mathcal{M} which begins $q_i s_j$, then there is a uniquely applicable production of $\Sigma(\mathcal{M})$ and the result of applying it will be a Post word;
- c. if v begins with the symbol s_j and there is no quadruple of \mathcal{M} which begins $q_i s_j$, then the one applicable production of $\Sigma(\mathcal{M})$ is

$$q_i s_j \rightarrow q_{n+1} s_j,$$

which yields another Post word when applied to w .

If $i = n + 1$, then

- a. if $v = 0$, the only applicable production of $\Sigma(\mathcal{M})$ is

$$q_{n+1}h \rightarrow q_0h,$$

which yields a Post word;

- b. if v begins with the symbol s_j , the only applicable production of $\Sigma(\mathcal{M})$ is

$$q_{n+1}s_j \rightarrow q_{n+1},$$

which again yields a Post word.

Finally, if $i = 0$, then

- a. if $u = 0$, no production of $\Sigma(\mathcal{M})$ can be applied;
- b. if u ends with s_j , the only applicable production of $\Sigma(\mathcal{M})$ is

$$s_j q_0 \rightarrow q_0,$$

which yields a Post word. ■

Our next result makes precise the sense in which $\Sigma(\mathcal{M})$ simulates \mathcal{M} .

Theorem 2.2. Let \mathcal{M} be a nondeterministic Turing machine. Then, for each string u on the alphabet of \mathcal{M} , \mathcal{M} accepts u if and only if

$$hq_1s_0uh \xrightarrow[\Sigma(\mathcal{M})]{*} hq_0h.$$

Proof. Let the alphabet of \mathcal{M} be s_1, \dots, s_K . First let us suppose that \mathcal{M} accepts u . Then, if \mathcal{M} begins in the configuration

$$\begin{array}{c} s_0 \quad u \\ \uparrow \\ q_1 \end{array}$$

it will eventually reach a state q_i scanning a symbol s_k where no quadruple of \mathcal{M} begins $q_i s_k$. Then we will have (for appropriate words v, w on the alphabet of \mathcal{M})

$$\begin{aligned} hq_1s_0uh &\xrightarrow[\Sigma(\mathcal{M})]{*} hvq_is_kwh \xrightarrow[\Sigma(\mathcal{M})]{*} hvq_{n+1}s_kwh \\ &\xrightarrow[\Sigma(\mathcal{M})]{*} hvq_0h \xrightarrow[\Sigma(\mathcal{M})]{*} hq_0h. \end{aligned}$$

Next suppose that \mathcal{M} does not accept u . Then, beginning with configuration

$$\begin{array}{c} s_0 u \\ \uparrow \\ q_1 \end{array}$$

\mathcal{M} will never halt. Let

$$w_1 = hq_1s_0uh,$$

and suppose that

$$w_1 \xrightarrow[\Sigma(\mathcal{M})]{} w_2 \xrightarrow[\Sigma(\mathcal{M})]{} w_3 \xrightarrow[\Sigma(\mathcal{M})]{} \cdots \xrightarrow[\Sigma(\mathcal{M})]{} w_m.$$

Then each w_j , $1 \leq j \leq m$, must contain a symbol q_i with $1 \leq i \leq n$. Hence there can be no derivation of a Post word containing q_0 from w_1 , and so, in particular, there is no derivation of hq_0h from w_1 . ■

Definition. The *inverse* of the production $g \rightarrow \bar{g}$ is the production $\bar{g} \rightarrow g$.

For example, the inverse of the production $ab \rightarrow aa$ is the production $aa \rightarrow ab$.

Let us write $\Omega(\mathcal{M})$ for the semi-Thue process which consists of the inverses of all the productions of $\Sigma(\mathcal{M})$. Then an immediate consequence of Theorem 2.2 is

Theorem 2.3. Let \mathcal{M} be a nondeterministic Turing machine. Then for each string u in the alphabet of \mathcal{M} , \mathcal{M} accepts u if and only if

$$hq_0h \xrightarrow[\Omega(\mathcal{M})]{*} hq_1s_0uh.$$

Exercises

1. (a) Give $\Sigma(\mathcal{M})$, where \mathcal{M} is the Turing machine in Table 1.1 of Chapter 6.
 (b) Give a derivation that shows that $hq_1s_0111h \xrightarrow[\Sigma(\mathcal{M})]{*} hq_0h$.
 (c) Describe $\{u \mid hq_0h \xrightarrow[\Omega(\mathcal{M})]{*} hq_1s_0uh\}$.
2. Give a semi-Thue process Π such that, for all words $u, v \in \{1, 2\}^*$, $hq_1s_0us_0vh \xrightarrow[\Pi]{*} w \in \{1, 2\}^*$, where $u + v = w$ in binary notation.
3. Show that for any partially computable function $f(x)$, there is a semi-Thue process Π such that for all $x \in N$, $1^{[x]} \xrightarrow[\Pi]{*} 1^{[y]}$ if and only if $y = f(x)$.

3. Unsolvable Word Problems

Definition. The *word problem* for a semi-Thue process Π is the problem of determining for any given pair u, v of words on the alphabet of Π whether $u \xrightarrow[\Pi]{*} v$.

We shall prove

Theorem 3.1. There is a Turing machine \mathcal{M} such that the word problem is unsolvable for both the semi-Thue processes $\Sigma(\mathcal{M})$ and $\Omega(\mathcal{M})$.

Proof. By Theorem 3.1 in Chapter 6, there is a Turing machine \mathcal{M} (in fact, deterministic) that accepts a nonrecursive language. Suppose first that the word problem for $\Sigma(\mathcal{M})$ were solvable. Then there would be an algorithm for testing given words v, w to determine whether $v \xrightarrow[\Sigma(\mathcal{M})]{*} w$. By Theorem 2.2, we could use this algorithm to determine whether \mathcal{M} will accept a given word u by testing whether

$$hq_1s_0uh \xrightarrow[\Sigma(\mathcal{M})]{*} hq_0h.$$

We would thus have an algorithm for testing a given word u to see whether \mathcal{M} will accept it. But such an algorithm cannot exist since the language accepted by \mathcal{M} is not a recursive set.

Finally, an algorithm that solved the word problem for $\Omega(\mathcal{M})$ would also solve the word problem for $\Sigma(\mathcal{M})$, since

$$v \xrightarrow[\Sigma(\mathcal{M})]{} w \quad \text{if and only if} \quad w \xrightarrow[\Omega(\mathcal{M})]{} v. \quad \blacksquare$$

Definition. A semi-Thue process is called a *Thue process* if the inverse of each production in the process is also in the process.

The fact that Thue processes are in fact “two-way” processes is a curious coincidence.

We write $g \leftrightarrow \bar{g}$ to combine the production $g \rightarrow \bar{g}$ and its inverse $\bar{g} \rightarrow g$.

For each Turing machine \mathcal{M} , we write

$$\Theta(\mathcal{M}) = \Sigma(\mathcal{M}) \cup \Omega(\mathcal{M}),$$

so that $\Theta(\mathcal{M})$ is a Thue process. We have

Theorem 3.2 (Post’s Lemma). Let \mathcal{M} be a deterministic Turing machine. Let u be a word on the alphabet of \mathcal{M} such that

$$hq_1s_0uh \xrightarrow[\Theta(\mathcal{M})]{} hq_0h.$$

Then

$$hq_1s_0uh \xrightarrow[\Sigma(\mathcal{M})]{} hq_0h.$$

Proof. Let the sequence

$$hq_1s_0uh = w_1, w_2, \dots, w_l = hq_0h$$

be a derivation in $\Theta(\mathcal{M})$. Since w_1 is a Post word, and each production of $\Theta(\mathcal{M})$ transforms Post words into Post words, we can conclude that the entire derivation consists of Post words. We need to show how to eliminate use of productions belonging to $\Omega(\mathcal{M})$ from this derivation. So let us assume that the last time in the derivation that a production of $\Omega(\mathcal{M})$ was used was in getting from w_i to w_{i+1} . That is, we assume

$$w_i \xrightarrow[\Omega(\mathcal{M})]{} w_{i+1}; \quad w_{i+1} \xrightarrow[\Sigma(\mathcal{M})]{} w_{i+2} \xrightarrow[\Sigma(\mathcal{M})]{} w_l = hq_0h.$$

Now, $\Omega(\mathcal{M})$ consists of inverses of productions of $\Sigma(\mathcal{M})$; hence we must have

$$w_{i+1} \xrightarrow[\Sigma(\mathcal{M})]{} w_i.$$

Moreover, we must have $i + 1 < l$ because no production of $\Sigma(\mathcal{M})$ can be applied to $w_l = hq_0h$. Now, w_{i+1} is a Post word and

$$w_{i+1} \xrightarrow[\Sigma(\mathcal{M})]{} w_i, \quad w_{i+1} \xrightarrow[\Sigma(\mathcal{M})]{} w_{i+2}.$$

By Theorem 2.1, we conclude that $w_{i+2} = w_i$. Thus the transition from w_i to w_{i+1} and then back to $w_{i+2} = w_i$ is clearly an unnecessary detour. That is, the sequence

$$w_1, w_2, \dots, w_i, w_{i+3}, \dots, w_l$$

from which w_{i+1}, w_{i+2} have been omitted is a derivation in $\Theta(\mathcal{M})$.

We have shown that any derivation that uses a production belonging to $\Omega(\mathcal{M})$ can be shortened. Continuing this procedure, we eventually obtain a derivation using only productions of $\Sigma(\mathcal{M})$. ■

Theorem 3.3 (Post–Markov). If the deterministic Turing machine \mathcal{M} accepts a nonrecursive set, then the word problem for the Thue process $\Theta(\mathcal{M})$ is unsolvable.

Proof. Let u be a word on the alphabet of \mathcal{M} . Then we have, using Theorems 2.2 and 3.2,

$$\mathcal{M} \text{ accepts } u$$

if and only if

$$hq_1s_0uh \xrightarrow[\Sigma(\mathcal{M})]{*} hq_0h$$

if and only if

$$hq_1s_0uh \xrightarrow[\Theta(\mathcal{M})]{*} hq_0h.$$

Hence, an algorithm for solving the word problem for $\Theta(\mathcal{M})$ could be used to determine whether or not \mathcal{M} will accept u , which is impossible. ■

Now we consider semi-Thue processes on an alphabet of two symbols.

Theorem 3.4. There is a semi-Thue process on the alphabet $\{a, b\}$ whose word problem is unsolvable. Moreover, for each production $g \rightarrow h$ of this semi-Thue process, $g, h \neq 0$.

Proof. Let us begin with a semi-Thue process Π on the alphabet $A = \{a_1, \dots, a_n\}$ and with productions

$$g_i \rightarrow \bar{g}_i, \quad i = 1, 2, \dots, m,$$

whose word problem is unsolvable. We also assume that for each $i = 1, 2, \dots, m$, $g_i \neq 0$ and $\bar{g}_i \neq 0$. This is legitimate because this condition is satisfied by the productions of $\Sigma(\mathcal{M})$.

We write

$$a'_j = ba^{[j]}b, \quad j = 1, 2, \dots, n,$$

where there is a string of a 's of length j between the two b 's. Finally, for any word $w \neq 0$ in A^* ,

$$w = a_{j_1} a_{j_2} \cdots a_{j_k},$$

we write

$$w' = a'_{j_1} a'_{j_2} \cdots a'_{j_k}.$$

In addition we let $0' = 0$. Then, we consider the semi-Thue process Π' on the alphabet $\{a, b\}$ whose productions are

$$g'_i \rightarrow \bar{g}'_i, \quad i = 1, 2, \dots, m.$$

We have

Lemma 1. If $u \xrightarrow{\Pi} v$, then $u' \xrightarrow{\Pi'} v'$.

Proof. We have $u = rg_is$, $v = r\bar{g}_is$. Hence $u' = r'g'_is'$, $v' = r'\bar{g}'_is'$, so that $u' \xrightarrow{\Pi'} v'$. ■

Lemma 2. If $u' \xrightarrow{\Pi'} w$, then for some $v \in A^*$ we have $w = v'$ and $u \xrightarrow{\Pi} v$.

Proof. We have $u' = pg'_iq$, $w = p\bar{g}'_iq$. Now, since $g_i \neq 0$, g'_i begins and ends with the letter b . Hence each of p and q either begins and ends with b or is 0, so that $p = r'$, $q = s'$. Then, $u = rg_is$. Let $v = r\bar{g}_is$. Then $w = v'$ and $u \xrightarrow{\Pi} v$. ■

Lemma 3. $u \xrightarrow{\ast}_{\Pi} v$ if and only if $u' \xrightarrow{\ast}_{\Pi'} v'$.

Proof. If $u = u_1 \xrightarrow{\Pi} u_2 \xrightarrow{\Pi} \cdots \xrightarrow{\Pi} u_n = v$, then by Lemma 1

$$u' = u'_1 \xrightarrow{\ast}_{\Pi'} u'_2 \xrightarrow{\ast}_{\Pi'} \cdots \xrightarrow{\ast}_{\Pi'} u'_n = v'.$$

Conversely, if

$$u' = w_1 \xrightarrow{\Pi'} w_2 \xrightarrow{\Pi'} \cdots \xrightarrow{\Pi'} w_n = v',$$

then by Lemma 2, for each w_i there is a string $u_i \in A^*$ such that $w_i = u'_i$. Thus,

$$u' = u'_1 \xrightarrow{\Pi'} u'_2 \xrightarrow{\Pi'} \cdots \xrightarrow{\Pi'} u'_n = v'.$$

By Lemma 2 once again,

$$u = u_1 \xrightarrow{\Pi} u_2 \xrightarrow{\Pi} \cdots \xrightarrow{\Pi} u_n = v,$$

so that $u \xrightarrow{\Pi^*} v$. ■

Proof of Theorem 3.4 Concluded. By Lemma 3, if the word problem for Π' were solvable, the word problem for Π would also be solvable. Hence, the word problem for Π' is unsolvable. ■

In the preceding proof it is clear that if the semi-Thue process Π with which we begin is actually a Thue process, then Π' will be a Thue process on $\{a, b\}$. We conclude

Theorem 3.5. There is a Thue process on the alphabet $\{a, b\}$ whose word problem is unsolvable. Moreover, for each production $g \rightarrow h$ of this Thue process, $g, h \neq 0$.

Exercises

- Let Π be the semi-Thue process with productions $cde \rightarrow ce$, $d \rightarrow cde$. Use the construction in the proof of Theorem 3.4 to get a semi-Thue process Π' with productions on $\{a, b\}$ such that $u \xrightarrow{\Pi^*} v$ if and only if $u' \xrightarrow{\Pi'} v'$ for all words $u, v \in \{c, d, e\}^*$.
- A semi-Thue system is defined to be a pair (u_0, Π) , where Π is a semi-Thue process and u_0 is a given word on the alphabet of Π . A word w is called a *theorem* of (u_0, Π) if $u_0 \xrightarrow{\Pi^*} w$. Show that there is a semi-Thue system for which no algorithm exists to determine whether a given string is a theorem of the system.
- Let Π be a semi-Thue process containing only one production. Show that Π has a solvable word problem.
- * Give an upper bound on the size of the smallest semi-Thue process with an undecidable word problem. [See Exercise 2.2 in Chapter 6.]

4. Post's Correspondence Problem

The Post correspondence problem first appeared in a paper by Emil Post in 1946. It was only much later that this problem was seen to have important applications in the theory of formal languages.

Our treatment of the Post correspondence problem is a simplification of a proof due to Floyd, itself much simpler than Post's original work.

The correspondence problem may conveniently be thought of as a solitaire game played with special sets of dominoes. Each domino has a word (on some given alphabet) appearing on each half. A typical domino is shown in Fig. 4.1. A *Post correspondence system* is simply a finite set of dominoes of this kind. Figure 4.2 gives a simple example of a Post correspondence system using three dominoes and the alphabet $\{a, b\}$. Each move in the solitaire game defined by a particular Post correspondence system consists of simply placing one of the dominoes of the system to the right of the dominoes laid down on previous moves. The key fact is that the *dominoes are not used up by being played, so that each one can be used any number of times*. The way to “win” the game is to reach a situation where the very same word appears on the top halves as on the bottom halves of the dominoes when we read across from left to right. Figure 4.3 shows how to win the game defined by the dominoes of Fig. 4.2. (Note that one of the dominoes is used twice.) The word $aabbba$ which appears across both the top halves and bottom halves is called a *solution* of the given Post correspondence system. Thus a Post correspondence system possesses a *solution* if and only if it is possible to win the game defined by that system.

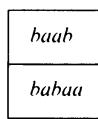


Figure 4.1

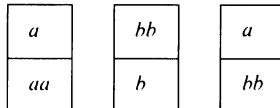


Figure 4.2

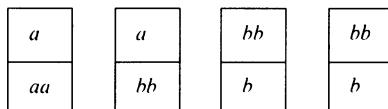


Figure 4.3

We shall prove

Theorem 4.1. There is no algorithm that can test a given Post correspondence system to determine whether it has a solution.

Proof. Using Theorem 3.4, we begin with a semi-Thue process Π on the alphabet $\{a, b\}$ whose word problem is unsolvable. We modify Π in the following trivial way: we add to the productions of Π the two productions

$$a \rightarrow a, \quad b \rightarrow b.$$

Naturally this addition has no effect on whether

$$u \xrightarrow[\Pi]{} v$$

for given words u, v . However, it does guarantee that whenever $u \xrightarrow[\Pi]{} v$, there is a derivation

$$u = u_1 \xrightarrow[\Pi]{} u_2 \xrightarrow[\Pi]{} \cdots \xrightarrow[\Pi]{} u_m = v,$$

where m is an odd number. This is because with the added productions we have

$$u_i \xrightarrow[\Pi]{} u_i$$

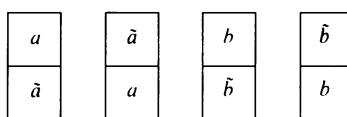
for each i , so that any step in a derivation (e.g., the first) can be repeated if necessary to change the length of the derivation from an even to an odd number.

Let u and v be any given words on the alphabet $\{a, b\}$. We shall construct a Post correspondence system $P_{u,v}$ (which depends on Π as well as on the words u and v) such that $P_{u,v}$ has a solution if and only if $u \xrightarrow[\Pi]{} v$. Once we have obtained this $P_{u,v}$ we are through. For, if there were an algorithm for testing given Post correspondence systems for possessing a solution, this algorithm could be applied in particular to $P_{u,v}$ and therefore to determine whether $u \xrightarrow[\Pi]{} v$; since Π has an unsolvable word problem, this is impossible.

We proceed to show how to construct $P_{u,v}$. Let the productions of Π (including the two we have just added) be $g_i \rightarrow h_i$, $i = 1, 2, \dots, n$. The alphabet of $P_{u,v}$ consists of the eight symbols

$$a \ b \ \tilde{a} \ \tilde{b} \ [\] \ * \ \tilde{*}.$$

For any word w on $\{a, b\}$, we write \tilde{w} for the word on $\{\tilde{a}, \tilde{b}\}$ obtained by placing “~” on top of each symbol of w . $P_{u,v}$ is then to consist of the $2n + 4$ dominoes shown in Fig. 4.4. Note that because Π contains the productions $a \rightarrow a$ and $b \rightarrow b$, $P_{u,v}$ contains the four dominoes



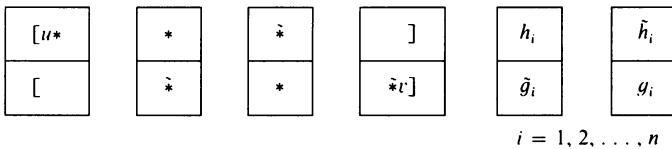
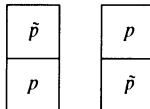


Figure 4.4

Therefore, it is clear that in our play it is legitimate to use dominoes of the form



where p is any word on $\{a, b\}$, since any such dominoes can be assembled by lining up single dominoes selected appropriately from the previous four.

We proceed to show that $P_{u,v}$ has a solution if and only if $u \xrightarrow{\Pi}^* v$.

First suppose that $u \xrightarrow{\Pi}^* v$. Let

$$u = u_1 \xrightarrow{\Pi} u_2 \xrightarrow{\Pi} \cdots \xrightarrow{\Pi} u_m = v,$$

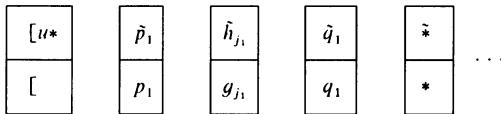
where m is an odd number. Thus, for each i , $1 \leq i < m$, we can write

$$u_i = p_i g_{j_i} q_i, \quad u_{i+1} = p_i h_{j_i} q_i,$$

where the transition from u_i to u_{i+1} is via the j_i th production of Π . Then we claim that the word

$$\left[u_1 * \tilde{u}_2 * u_3 * \cdots * \tilde{u}_{m-1} * u_m \right] \tag{4.1}$$

is a solution of $P_{u,v}$. To see this, let us begin to play by laying down the dominoes:



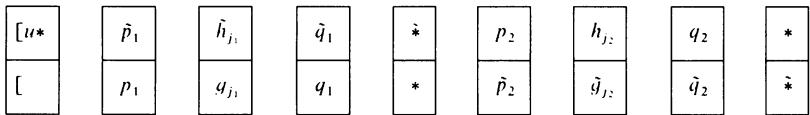
At this stage, the word on top is

$$[u_1 * \tilde{u}_2 *$$

while the word on the bottom is

$$[u_1 * .$$

We can continue to play as follows:



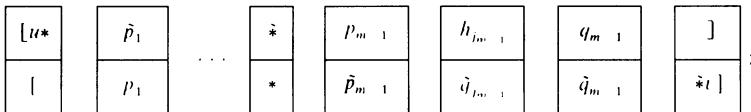
Now the word on top is

$$[u_1 * \tilde{u}_2 * u_3 *]$$

and the word on the bottom is

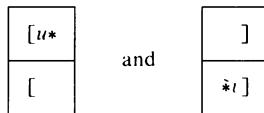
$$[u_1 * \tilde{u}_2 *]$$

Recalling that m is an odd number we see that we can win by continuing as follows:

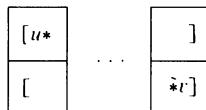


for, at this point the word both on top and on bottom is (4.1).

Conversely suppose that $P_{u,v}$ has a solution w . Examining Fig. 4.4, we see that the only possible way to win involves playing



first and last, respectively. This is because none of the other dominoes in $P_{u,v}$ have tops and bottoms which begin (or end) with the same symbol. Thus, w must begin with [and end with]. Let us write $w = [z]y$, where z contains no]. (Of course it is quite possible that $y = 0$.) Since the only domino containing] contains it on the far right on top and on bottom, we see that [z] itself is already a solution to $P_{u,v}$. We work with this solution. So far we know that the game looks like this:



so that the solution [z] looks like this:

$$[u * \dots * v].$$

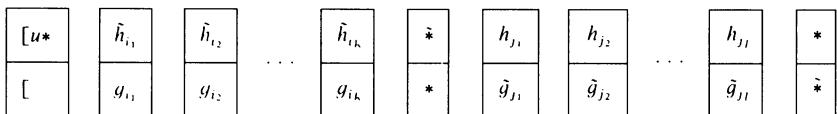
Continuing from the left we see that the play must go



where $g_{i_1}g_{i_2}\cdots g_{i_k} = u$. (This is necessary in order for the bottom to “catch up” with the $u*$ which is already on top.) Writing $u = u_1$ and $u_2 = h_{i_1}h_{i_2}\cdots h_{i_k}$ we see that $u_1 \xrightarrow{*} u_2$ and that the solution has the form

$$[u_1 * \tilde{u}_2 \tilde{*} \cdots \tilde{*} v].$$

Now we see how the play must continue:



where of course $u_2 = g_{j_1}g_{j_2}\cdots g_{j_l}$. Again, writing $u_3 = h_{j_1}h_{j_2}\cdots h_{j_l}$ we have that $u_2 \xrightarrow{*} u_3$ and that the solution has the form

$$[u_1 * \tilde{u}_2 \tilde{*} u_3 * \cdots * v].$$

Continuing, it is clear that the solution can be written

$$[u_1 * \tilde{u}_2 \tilde{*} u_3 * \cdots * \tilde{u}_{m-1} \tilde{*} u_m],$$

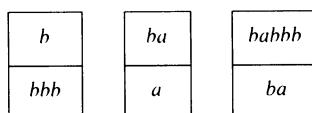
where

$$u = u_1 \xrightarrow{*} u_2 \xrightarrow{*} u_3 \xrightarrow{*} \cdots \xrightarrow{*} u_{m-1} \xrightarrow{*} u_m = v,$$

so that $u \xrightarrow{*} v$. ■

Exercises

- Let Π be the semi-Thue process with productions $aba \rightarrow a$, $b \rightarrow aba$, and let $u = bb$, $v = aaaaaa$. Describe the Post correspondence system $P_{u,v}$ and give a solution to $P_{u,v}$.
- Find a solution to the Post correspondence problem defined by the dominoes



3. Find an algorithm for Post correspondence problems whose alphabet consists of just one symbol.

5. Grammars

A *phrase-structure grammar* or simply a *grammar* is just a semi-Thue process in which the letters of the alphabet are separated into two disjoint sets called the *variables* and the *terminals*, with one of the variables singled out as the *start symbol*. It is customary (but, of course, not necessary) to use lower case letters for terminals, capital letters for variables, and in particular the letter S for the start symbol.

Let Γ be a grammar with start symbol S and let \mathcal{V}, T be the sets of variables and terminals of Γ , respectively. Then we define

$$L(\Gamma) = \{u \in T^* \mid S \xrightarrow{*} u\},$$

and call $L(\Gamma)$ the *language generated by Γ* . Our purpose in this section is to characterize languages which can be generated by grammars.

We first prove

Theorem 5.1. Let U be a language accepted by a nondeterministic Turing machine. Then there is a grammar Γ such that $U = L(\Gamma)$.

Proof. Let $U \subseteq T^*$ and let \mathcal{M} be a nondeterministic Turing machine that accepts U . We will construct Γ by modifying the semi-Thue process $\Omega(\mathcal{M})$ from Section 2. Let \mathcal{M} have the states q_1, \dots, q_n . Then we recall that the alphabet of $\Omega(\mathcal{M})$ [which is the same as that of $\Sigma(\mathcal{M})$] consists of $s_0, q_0, q_1, q_2, \dots, q_n, q_{n+1}, h$ in addition to the letters of the alphabet of \mathcal{M} . We let the *terminals* of Γ be just the letters of T , and the variables of Γ be the symbols from the alphabet of $\Omega(\mathcal{M})$ not in T , together with the two additional symbols S and q . S is to be the start symbol of Γ . The productions of Γ are then the productions of $\Omega(\mathcal{M})$ together with the productions

$$S \rightarrow hq_0h$$

$$hq_1s_0 \rightarrow q$$

$$qs \rightarrow sq \quad \text{for each } s \in T$$

$$qh \rightarrow 0.$$

Now, let \mathcal{M} accept $u \in T^*$. Then, using Theorem 2.3, we have

$$S \xrightarrow{\Gamma} hq_0h \xrightarrow{\Gamma} hq_1s_0uh \xrightarrow{\Gamma} quh \xrightarrow{\Gamma} uqh \xrightarrow{\Gamma} u,$$

so that $u \in L(\Gamma)$.

Conversely, let $u \in L(\Gamma)$. Then $u \in T^*$ and $S \xrightarrow{\Gamma} u$. Examining the list of productions of Γ , we see that we must in fact have

$$S \xrightarrow{\Gamma} hq_0h \xrightarrow{\Gamma} vqhz \xrightarrow{\Gamma} vz = u.$$

Proceeding further, we see that the symbol q could only be introduced using the production

$$hq_1s_0 \rightarrow q.$$

Hence, our derivation must have the form

$$S \xrightarrow{\Gamma} hq_0h \xrightarrow{\Gamma} xhq_1s_0yhz \xrightarrow{\Gamma} xqyhz \xrightarrow{\Gamma} xyqhz \xrightarrow{\Gamma} xyz = u,$$

where of course $xy = v$. Thus, there is a derivation of xhq_1s_0yhz from hq_0h in Γ . Moreover, this must actually be a derivation in $\Omega(\mathcal{M})$ since the added productions are clearly inapplicable. Moreover, the productions of $\Omega(\mathcal{M})$ always lead from Post words to Post words. Hence, xhq_1s_0yhz must be a Post word. That is, $x = z = 0$ and $u = xyz = y$. We conclude that

$$hq_0h \xrightarrow[\Omega(\mathcal{M})]{} hq_1s_0uh.$$

Thus by Theorem 2.3, \mathcal{M} accepts u . ■

Now, let us begin with a grammar Γ and see what we can say about $L(\Gamma)$. Thus, let the alphabet of Γ be

$$\{s_1, \dots, s_n, V_1, \dots, V_k\},$$

where $T = \{s_1, \dots, s_n\}$ is the set of terminals, V_1, \dots, V_k are the variables, and $S = V_1$ is the start symbol. Let us order the alphabet of Γ as shown. Thus strings on this alphabet are notations for integers in the base $n + k$. We have

Lemma 1. The predicate $u \xrightarrow{\Gamma} v$ is primitive recursive.

Proof. Let the productions Γ be $g_i \rightarrow h_i$, $i = 1, 2, \dots, l$. We write, for $i = 1, 2, \dots, l$,

$$\text{PROD}_i(u, v) \Leftrightarrow (\exists r, s)_{\leq_u} [u = \text{CONCAT}(r, g_i, s) \ \& \ v = \text{CONCAT}(r, h_i, s)].$$

Since, by Chapter 5, Section 1, CONCAT is primitive recursive, each of the predicates PROD_i is primitive recursive. But

$$u \xrightarrow[\Gamma]{} v \Leftrightarrow \text{PROD}_1(u, v) \vee \text{PROD}_2(u, v) \vee \cdots \vee \text{PROD}_l(u, v),$$

and the result follows. ■

We write $\text{DERIV}(u, y)$ to mean that for some m , $y = [u_1, \dots, u_m, 1]$, where the sequence u_1, \dots, u_m is a derivation of u from S in Γ . (The “1” has been added to avoid complications in case $u_m = u = 0$.) Then, since the value of S in base $n + k$ is $n + 1$ [because $S = V_1$ is the $(n + 1)$ th symbol in our alphabet], we have

$$\begin{aligned} \text{DERIV}(u, y) \Leftrightarrow & (\exists m)_{\leq y} \left(m + 1 = \text{Lt}(y) \& (y)_1 = n + 1 \right. \\ & \& (y)_m = u \& (y)_{m+1} = 1 \\ & \& \left. \& (\forall j)_{<m} \left\{ j = 0 \vee [(y)_j \xrightarrow[\Gamma]{} (y)_{j+1}] \right\} \right). \end{aligned}$$

Using Lemma 1, we have proved

Lemma 2. $\text{DERIV}(u, y)$ is primitive recursive.

Also, by definition of $\text{DERIV}(u, y)$, we have for every word u on the alphabet of Γ

$$S \xrightarrow[\Gamma]{} u \Leftrightarrow (\exists y) \text{DERIV}(u, y). \quad (5.1)$$

Finally, (5.1) shows that

$$S \xrightarrow[\Gamma]{} u \Leftrightarrow \min_y \text{DERIV}(u, y) \downarrow.$$

Hence, by Lemma 2 and Theorem 7.2 in Chapter 3, we see that $\{u \mid S \xrightarrow[\Gamma]{} u\}$ is r.e. But

$$L(\Gamma) = T^* \cap \{u \mid S \xrightarrow[\Gamma]{} u\} \quad (5.2)$$

(where T is the alphabet of terminals of Γ), so that $L(\Gamma)$ is the intersection of two r.e. sets and hence is r.e. Combining this result with Theorem 5.1 in Chapter 6 and Theorem 5.1 in this chapter, we have

Theorem 5.2. A language U is r.e. if and only if there is a grammar Γ such that $U = L(\Gamma)$.

We now are able to obtain easily the promised converse to Theorem 5.1 in Chapter 6. In fact putting Theorem 3.1 in Chapter 6 and Theorems 5.1 and 5.2 in this chapter all together, we have

Theorem 5.3. Let L be a given language. Then the following conditions are all equivalent:

1. L is r.e.;
2. L is accepted by a deterministic Turing machine;
3. L is accepted by a nondeterministic Turing machine;
4. there is a grammar Γ such that $L = L(\Gamma)$.

Theorem 5.3 involves some of the main concerns of theoretical computer science: on the one hand, the relation between grammars, the languages they generate, and the devices that accept them; on the other hand, the relation, for various devices, between determinism and nondeterminism.

We will conclude this section by obtaining a result that will be needed in Chapter 11, but can easily be proved at this point.

Definition. A grammar Γ is called *context-sensitive* if for each production $g \rightarrow h$ of Γ we have $|g| \leq |h|$.

Lemma 3. If Γ is context-sensitive, then

$$\{u \mid S \xrightarrow[\Gamma]{} u\}$$

is recursive.

Proof. It will suffice to obtain a recursive bound for y in formula (5.1). Since

$$1 = |u_1| \leq |u_2| \leq \dots \leq |u_m| = |u|$$

for any derivation u_1, \dots, u_m of u from S in the context-sensitive grammar Γ , we must have

$$u_1, u_2, \dots, u_m \leq g(u),$$

where $g(u)$ is the smallest number which represents a string of length $|u| + 1$ in base $n + k$. Now, since $g(u)$ is simply the value in base $n + k$ of a string consisting of $|u| + 1$ repetitions of 1, we have

$$g(u) = \sum_{i=0}^{|u|} (n+k)^i,$$

which is primitive recursive because $|u|$ is primitive recursive. Next, note that we may assume that the derivation

$$S = u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_m = u$$

contains no repetitions. This is because given a sequence of steps

$$z = u_i \Rightarrow u_{i+1} \Rightarrow \cdots \Rightarrow u_{i+l} = z,$$

we could simply eliminate the steps u_{i+1}, \dots, u_{i+l} . Hence the length m of the derivation is bounded by the total number of distinct strings of length $\leq |u|$ on our alphabet of $n+k$ symbols. But this number is just $g(u)$. Hence,

$$[u_1, \dots, u_m, 1] = \prod_{i=1}^m p_i^{u_i} \cdot p_{m+1} \leq h(u),$$

where we have written $h(u)$ for the primitive recursive function defined by

$$h(u) = \prod_{i=1}^{g(u)} p_i^{g(u)} \cdot p_{g(u)+1}.$$

Finally, we have

$$S \xrightarrow{*_{\Pi}} u \Leftrightarrow (\exists y)_{\leq h(u)} \text{DERIV}(u, y),$$

which gives the result. ■

Theorem 5.4. If Γ is a context-sensitive grammar, then $L(\Gamma)$ is recursive.

Proof. We will use Lemma 3 and Eq. (5.2). Since T^* is a recursive set, the result follows at once. ■

Exercises

- For each of the following languages L , give a grammar Γ such that $L = L(\Gamma)$.
 - $L = \{a^{[n]}b^{[n]} \mid n \in N\}$.
 - $L = \{a^{[n]}b^{[m]} \mid n \leq m\}$.
 - $L = \{ww^R \mid w \in \{a, b\}^*\}$.
- Use the construction in the proof of Theorem 5.1 to give a grammar Γ such that $L(\Gamma) = \{1^{[m]}B1^{[n]}B1^{[m+n]} \mid m, n \geq 0\}$.
- Write down the proof of Theorem 5.2.

4. (a) Let Γ have the variables S, B, C , the terminals a, b, c and the productions

$$\begin{aligned} S &\rightarrow aSBC, & S &\rightarrow aBC, \\ CB &\rightarrow BC, & bB &\rightarrow bb, \\ aB &\rightarrow ab, & bC &\rightarrow bc, \\ && cC &\rightarrow cc. \end{aligned}$$

Prove that for each $n \neq 0$, $a^{[n]}b^{[n]}c^{[n]} \in L(\Gamma)$.

- (b)* Prove that $L(\Gamma) = \{a^{[n]}b^{[n]}c^{[n]} \mid n \neq 0\}$.

6. Some Unsolvable Problems Concerning Grammars

How much information can we hope to obtain about $L(\Gamma)$ by a computation that uses the grammar Γ as input? Not much at all, as we shall see.

Let \mathcal{M} be a Turing machine and let u be some given word on the alphabet of \mathcal{M} . We shall construct a grammar Γ_u as follows:

The variables of Γ_u are the *entire* alphabet of $\Sigma(\mathcal{M})$ together with S (the start symbol) and V . There is just one terminal, namely, a . The productions of Γ_u are all of the productions of $\Sigma(\mathcal{M})$ together with

$$\begin{aligned} S &\rightarrow hq_1s_0uh \\ hq_0h &\rightarrow V \\ V &\rightarrow aV \\ V &\rightarrow a. \end{aligned}$$

Then it follows at once from Theorems 2.1 and 2.2 that $S \xrightarrow{\Gamma_u} V$ if and only if \mathcal{M} accepts u . Thus we have

Lemma. If \mathcal{M} accepts u , then $L(\Gamma_u) = \{a^{[i]} \mid i \neq 0\}$. If \mathcal{M} does not accept u , then $L(\Gamma_u) = \emptyset$.

Now we can select \mathcal{M} so that the language it accepts is not recursive. Then there is no algorithm for determining for given u whether \mathcal{M} accepts u . But the lemma obviously implies the equivalences

$$\begin{aligned} \mathcal{M} \text{ accepts } u &\Leftrightarrow L(\Gamma_u) \neq \emptyset \\ &\Leftrightarrow L(\Gamma_u) \text{ is infinite} \\ &\Leftrightarrow a \in L(\Gamma_u). \end{aligned}$$

We have obtained

Theorem 6.1. There is no algorithm to determine of a given grammar Γ whether

1. $L(\Gamma) = \emptyset$,
2. $L(\Gamma)$ is infinite, or
3. $v_0 \in L(\Gamma)$ for a fixed word v_0 .

We can also prove

Theorem 6.2. There is no algorithm for determining of a given pair Γ, Δ of grammars whether

1. $L(\Delta) \subseteq L(\Gamma)$,
2. $L(\Delta) = L(\Gamma)$.

Proof. Let Δ be the grammar with the single variable S , the single terminal a , and the productions

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow a. \end{aligned}$$

Then $L(\Delta) = \{a^{[i]} \mid i \neq 0\}$. Thus we have by the previous lemma

$$\mathcal{M} \text{ accepts } u \Leftrightarrow L(\Delta) = L(\Gamma_u) \Leftrightarrow L(\Delta) \subseteq L(\Gamma_u).$$

The result follows at once. ■

Exercise

1. Show that there is no algorithm to determine of a given grammar Γ whether
 - (a) $L(\Gamma)$ contains at least one word with exactly three symbols;
 - (b) v_0 is the shortest word in $L(\Gamma)$ for some given word v_0 ;
 - (c) $L(\Gamma) = A^*$ for some given alphabet A .

*7. Normal Processes

Given a pair of words g and \bar{g} we write

$$gz \rightarrow z\bar{g}$$

to indicate a kind of transformation on strings called a *normal production*. If P is the normal production $gz \rightarrow z\bar{g}$ we write

$$u \xrightarrow{P} v$$

if for some string z we have

$$u = gz, \quad v = z\bar{g}.$$

That is, v can be obtained from u by crossing off g from the left of u and adjoining \bar{g} to the right. A *normal process* is simply a finite set of normal productions. If ν is a normal process, we write

$$u \xrightarrow{\nu} v$$

to mean that

$$u \xrightarrow{P} v$$

for some production P in ν . Finally, we write

$$u \xrightarrow{*} v$$

to mean that there is a sequence (called a *derivation*)

$$u = u_1 \xrightarrow{\nu} u_2 \xrightarrow{\nu} \cdots \xrightarrow{\nu} u_m = v.$$

The *word problem* for ν is the problem of determining of two given words u, v whether $u \xrightarrow{*} v$.

Let Π be a semi-Thue process on the alphabet $\{a, b\}$ with an unsolvable word problem. We shall show how to simulate Π by a normal process ν on the alphabet $\{a, b, \tilde{a}, \tilde{b}\}$. As earlier, if $u \in \{a, b\}^*$, we write \tilde{u} for the word on $\{\tilde{a}, \tilde{b}\}$ obtained by placing \sim above each letter in u . Let the productions of Π be

$$g_i \rightarrow h_i, \quad i = 1, 2, \dots, n.$$

Then the productions of ν will be

$$\begin{aligned} g_i z &\rightarrow z\tilde{h}_i & i = 1, 2, \dots, n \\ az &\rightarrow z\tilde{a} \\ bz &\rightarrow z\tilde{b} \\ \tilde{a}z &\rightarrow za \\ \tilde{b}z &\rightarrow zb. \end{aligned}$$

A word on $\{a, b, \tilde{a}, \tilde{b}\}$ is called *proper* if it can be written in one of the forms $u\tilde{v}$ or $\tilde{u}v$, where u, v are words on $\{a, b\}$. We say that two words are

associates if there is a derivation of one from the other *using only the last four productions* of ν . A word on $\{a, b\}$ of length n has $2n$ associates, all of which are proper. For example, the associates of $baab$ are as follows:

$$baab \Rightarrow aabb \Rightarrow ab\tilde{b}a \Rightarrow b\tilde{b}\tilde{a}a \Rightarrow \tilde{b}\tilde{a}\tilde{a}\tilde{b} \Rightarrow \tilde{a}\tilde{a}\tilde{b}b \Rightarrow \tilde{a}\tilde{b}ba \Rightarrow \tilde{b}baa \Rightarrow baab.$$

Generally for $u, v \in \{a, b\}^*$, the proper words $u\tilde{v}$ and $\tilde{u}v$ are associates of each other and also of the word vu . In fact, vu is the unique word on $\{a, b\}$ which is an associate of $u\tilde{v}$. Thus, a word is proper just in case it is an associate of a word on $\{a, b\}$.

Lemma 1. If $u \xrightarrow{\Pi} v$, then $u \xrightarrow{*} v$.

Proof. We have $u = pg_iq$, $v = ph_iq$ for some i . Then

$$u \xrightarrow{*} g_iq\tilde{p} \xrightarrow{\nu} q\tilde{p}\tilde{h}_i \xrightarrow{*} ph_iq. \quad \blacksquare$$

Lemma 2. If $u \xrightarrow{\star} v$, then $u \xrightarrow{\star} v$.

Proof. Immediate from Lemma 1. \blacksquare

Lemma 3. Let u be proper and let $u \Rightarrow v$. Then there are words r, s on $\{a, b\}$ that are associates of u, v , respectively, such that $r \xrightarrow{\star} s$.

Proof. If v is an associate of u , then u and v are both associates of some word r on $\{a, b\}$, and the result follows because $r \xrightarrow{\star} r$.

If v is not an associate of u , the production used to obtain v from u must be one of the $g_i z \rightarrow \tilde{z}h_i$. Since u is proper, we have $u = g_iq\tilde{p}$, where p, q are words on $\{a, b\}$. Then $v = q\tilde{p}\tilde{h}_i$. Thus, setting

$$r = pg_iq, \quad s = ph_iq,$$

the result follows because $r \Rightarrow s$. \blacksquare

Lemma 4. Let u be proper and let $u \xrightarrow{\star} v$. Then there are words r, s on $\{a, b\}$ that are associates of u, v , respectively, such that $r \xrightarrow{\star} s$.

Proof. By induction on the length of the derivation in ν of v from u . The result is obvious if the derivation has length 1. Suppose the result is known for derivations of length m , and let

$$u = u_1 \xrightarrow{\nu} u_2 \xrightarrow{\nu} \cdots \xrightarrow{\nu} u_m \xrightarrow{\nu} u_{m+1} = v.$$

By the induction hypothesis, there are words r, z on $\{a, b\}$ that are associates of u, u_m , respectively, such that $r \xrightarrow{\star} z$. By Lemma 3, u_{m+1} is an associate of a word s on $\{a, b\}$ such that $z \xrightarrow{\star} s$. Thus, $r \xrightarrow{\star} s$. \blacksquare

Lemma 5. Let u, v be words on $\{a, b\}$. Then $u \xrightarrow{*} v$ if and only if $u \xrightarrow{\Pi}^{*} v$.

Proof. By Lemma 2 we know that $u \xrightarrow{\Pi}^{*} v$ implies $u \xrightarrow{*} v$. Conversely, if $u \xrightarrow{*} v$, by Lemma 4, $r \xrightarrow{\Pi} s$, where r, s are words on $\{a, b\}$ that are associates of u, v , respectively. But since u, v are already words on $\{a, b\}$, we have $r = u, s = v$, so that $u \xrightarrow{\Pi}^{*} v$. ■

Since Π was chosen to have an unsolvable word problem, it is now clear that ν has an unsolvable word problem. For, by Lemma 5, if we had an algorithm for deciding whether $u \xrightarrow{*} v$, we could use it to decide whether $u \xrightarrow{\Pi}^{*} v$.

We have proved

Theorem 7.1. There is a normal process on a four-letter alphabet with an unsolvable word problem.

Exercise

1. Show that there is a normal process with an unsolvable word problem whose alphabet contains only two letters.

8

Classifying Unsolvable Problems

1. Using Oracles

Once one gets used to the fact that there are explicit problems, such as the halting problem, that have no algorithmic solution, one is led to consider questions such as the following.

Suppose we were given a “black box” or, as one says, an *oracle*, which somehow can tell us whether a given Turing machine with given input eventually halts. (Of course, by Church’s thesis, the behavior of such an “oracle” cannot be characterized by an algorithm.) Then it is natural to consider a kind of program that is allowed to ask questions of our oracle and to use the answers in its further computation. Which noncomputable functions will now become computable?

In this chapter we will see how to give a precise answer to such questions. To begin with, we shall have to modify the programming language \mathcal{S} introduced in Chapter 2, to permit the use of “oracles.” Specifically, we change the definition of “statement” (in Chapter 2, Section 3) to allow statements of the form $V \leftarrow O(V)$ instead of $V \leftarrow V$. The modified version of \mathcal{S} thus contains four kinds of statement: increment, decrement, conditional branch, and this new kind of statement which we call an *oracle statement*. The definitions of *instruction*, *program*, *state*, *snapshot*, and *terminal snapshot* remain exactly as in Chapter 2.

We now let G be some partial function on N with values in N , and we shall think of G as an oracle. Let \mathcal{P} be a program of length n and let (i, σ) be a nonterminal snapshot of \mathcal{P} , i.e., $i \leq n$. We define the snapshot (j, τ) to be the G -successor of (i, σ) exactly as in the definition of successor in Chapter 2, Section 3, except that Case 3 is now replaced by

Case 3. The i th instruction of \mathcal{P} is $V \leftarrow O(V)$ and σ contains the equation $V = m$. If $G(m) \downarrow$, then $j = i + 1$ and τ is obtained from σ by replacing the equation $V = m$ by $V = G(m)$. If $G(m) \uparrow$, then (i, σ) has no successor.

Thus, when $G(m) \downarrow$, execution of this oracle statement has the intuitive effect of answering the computer's question " $G(m) = ?$ ". When $G(m) \uparrow$, an "out-of-bounds" condition is recognized, and the computer halts without reaching a terminal snapshot. Of course, when G is total, every nonterminal snapshot has a successor.

A G -computation is defined just like computation except that the word successor is replaced by G -successor. A number m that is replaced by $G(m)$ in the course of a G -computation (under Case 3) is called an oracle query of the G -computation. We define $\psi_{\mathcal{P}, G}^{(m)}(r_1, r_2, \dots, r_m)$ exactly as we defined $\psi_{\mathcal{P}}^{(m)}(r_1, r_2, \dots, r_m)$ in Chapter 2, Section 4, except that the word computation is replaced by G -computation.

Now, let G be a total function. Then, the partial function $\psi_{\mathcal{P}, G}^{(m)}(x_1, \dots, x_m)$ is said to be G -computed by \mathcal{P} . A partial function f is said to be partially G -computable or G -partial recursive if there is a program that G -computes it. A partially G -computable function that is total is called G -computable or G -recursive. Note that we have not defined partially G -computable unless G is a total function.

We have a few almost obvious theorems.

Theorem 1.1. If f is partially computable, then f is partially G -computable for all total functions G .

Proof. Clearly, we can assume that f is computed by a program \mathcal{P} containing no statements of the form¹ $V \leftarrow V$. Now this program \mathcal{P} is also

¹ Unlabeled statements $V \leftarrow V$ can just be deleted, and

$$[L] \quad V \leftarrow V$$

can be replaced by

$$\begin{aligned} [L] \quad &V \leftarrow V + 1 \\ &V \leftarrow V - 1. \end{aligned}$$

a program in the new revised sense; moreover, a computation of \mathcal{P} is the same thing as a G -computation of \mathcal{P} since \mathcal{P} contains no oracle statements. Hence $\psi_{\mathcal{P}, G}^{(m)} = \psi_{\mathcal{P}}^{(m)}$ for all G . ■

We write I for the identity function $I(x) = x$. (Thus, $I = u_1^1$.)

Theorem 1.2. f is partially computable if and only if f is partially I -computable.

Proof. If f is partially computable, then by Theorem 1.1 it is certainly partially I -computable. Conversely, let \mathcal{P} I -compute f . Let \mathcal{P}' be obtained from \mathcal{P} by replacing each oracle statement $V \leftarrow O(V)$ by $V \leftarrow V$. Then, \mathcal{P}' is a program in the original sense and \mathcal{P}' computes f . ■

Theorem 1.3. Let G be a total function. Then G is G -computable.

Proof. The following program² clearly G -computes G :

$$X \leftarrow O(X)$$

$$Y \leftarrow X$$

■

Theorem 1.4. The class of G -computable functions is a PRC class.

Proof. Exactly like the proof of Theorem 3.1 in Chapter 3. ■

This last proof illustrates a situation, which turns out to be quite typical, in which the proof of an earlier theorem can be used virtually intact to prove a theorem relative to an “oracle” G . One speaks of a *relativized* theorem and of *relativizing* a proof. It is a matter of taste how much detail to provide in such a case.

Theorem 1.5. Let F be partially G -computable and let G be H -computable. Then F is partially H -computable.

Proof. Let \mathcal{P} be a program which G -computes F . Let \mathcal{P}' be obtained from \mathcal{P} by replacing each oracle statement $V \leftarrow O(V)$ by a macro expansion obtained from some program which H -computes G . Then clearly, \mathcal{P}' H -computes F . ■

Theorem 1.6. Let G be any computable function. Then a function F is partially computable if and only if it is partially G -computable.

² Of course, we can freely use macro expansions, as explained in Chapter 2.

Proof. Theorem 1.1 gives the result in one direction. For the converse, let F be partially G -computable. By Theorem 1.2, G is I -computable. Hence, by Theorem 1.5, F is partially I -computable and so, by Theorem 1.2 again, F is partially computable. ■

It is useful to be able to work with “oracles” that are functions of more than one variable. We introduce this notion by using a familiar coding device from Chapter 3, Section 8.

Definition. Let f be a total n -ary function on N , $n > 1$. Then we say that g is (*partially*) f -computable to mean that g is (*partially*) G -computable, where

$$G(x) = f((x)_1, \dots, (x)_n). \quad (1.1)$$

Theorem 1.7. Let f be a total n -ary function on N . Then f is f -computable.

Proof. Let G be defined by (1.1). Then

$$f(x_1, \dots, x_n) = G([x_1, \dots, x_n]).$$

Hence the following program G -computes f :

$$Z \leftarrow [X_1, \dots, X_n]$$

$$Z \leftarrow O(Z)$$

$$Y \leftarrow Z$$

■

Since predicates are also total functions we can speak of a function being (*partially*) P -computable, where P is a predicate. Also, we speak of a function being (*partially*) A -computable when $A \subseteq N$; as usual, we simply identify A with the predicate that is its characteristic function.

Exercises

1. Provide a suitable definition of computability by a Post–Turing program relative to an oracle and prove an appropriate equivalence theorem.
2. For a given total function G from N to N , define the class $\text{Rec}(G)$ to be the class of functions obtained from G and the initial functions of Chapter 3 using composition, recursion, and minimalization. Prove that every function in $\text{Rec}(G)$ is partially G -computable.

2. Relativization of Universality

We now proceed to relativize the development in Chapter 4. As in Chapter 4, Section 1, we define an instruction number $\#(I) = \langle a, \langle b, c \rangle \rangle$ for all instructions I . The only difference is that $b = 0$ now indicates an oracle statement instead of one of the form $V \leftarrow V$. For \mathcal{P} a program, we now define $\#(\mathcal{P})$ as before. As indicated in Chapter 4, in order to avoid ambiguity we must not permit a program ending in the instruction whose number is 0. This instruction is now the unlabeled statement $Y \leftarrow O(Y)$. Hence, for complete rigor, if we wish to end a program with $Y \leftarrow O(Y)$, we will have to provide the statement with a spurious label.

We define $\Phi_G^{(n)}(x_1, \dots, x_n, y)$ to be $\psi_{\mathcal{P}, G}^{(n)}(x_1, \dots, x_n)$ where \mathcal{P} is the unique program such that $\#(\mathcal{P}) = y$. We also write $\Phi_G(x, y)$ for $\Phi_G^{(1)}(x, y)$. We have

Theorem 2.1 (Relativized Universality Theorem). Let G be total. Then the function $\Phi_G^{(n)}(x_1, \dots, x_n, y)$ is partially G -computable.

Proof. The proof of this theorem is essentially contained in the program of Fig. 2.1. The daggers (\ddagger) indicate the changes from the unrelativized universal program in Fig. 3.1 in Chapter 4. As in that case, what we have is essentially an interpretative program. The new element is of course the interpretation of oracle statements. This occurs in the following program segment which, not surprisingly, itself contains an oracle statement:

$$\begin{aligned} [O] \quad W &\leftarrow (S)_{r(U)+1} \\ B &\leftarrow W \\ B &\leftarrow O(B) \\ S &\leftarrow [S/P^W] \cdot P^B \end{aligned}$$

The program segment works as follows. First, W and B are both set to the current value of the variable in the oracle statement being interpreted. Then an oracle statement gives B a new value which is G of the old value. Finally, this new value is stored as an exponent on the appropriate prime in the number S . The remainder of the program works exactly as in the unrelativized case. ■

Let G be any *partial* function on N with values in N . Then we define the relativized step-counter predicate by

$$\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t) \Leftrightarrow \text{there is a } G\text{-computation of program number } y \text{ of length } \leq t + 1 \text{ beginning with inputs } x_1, \dots, x_n.$$

	$Z \leftarrow X_{n+1} + 1$	
	$S \leftarrow \prod_{i=1}^n (p_{2i})^{x_i}$	
	$K \leftarrow 1$	
[C]	IF $K = \text{Lt}(Z) + 1 \vee K = 0$ GOTO F	
	$U \leftarrow r((Z)_K)$	
	$P \leftarrow p_{r(U)+1}$	
	IF $l(U) = 0$ GOTO O	(‡)
	IF $l(U) = 1$ GOTO A	
	IF $\sim (P \mid S)$ GOTO N	
	IF $l(U) = 2$ GOTO M	
	$K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$	
	GOTO C	
[O]	$W \leftarrow (S)_{r(U)+1}$	(‡)
	$B \leftarrow W$	(‡)
	$B \leftarrow O(B)$	(‡)
	$S \leftarrow \lfloor S/P^W \rfloor \cdot P^B$	(‡)
	GOTO N	(‡)
[M]	$S \leftarrow \lfloor S/P \rfloor$	
	GOTO N	
[A]	$S \leftarrow S \cdot P$	
[N]	$K \leftarrow K + 1$	
	GOTO C	
[F]	$Y \leftarrow (S)_1$	

Figure 2.1. Program that G -computes $\Phi_G^{(n)}(X_1, \dots, X_n, X_{n+1})$.

As in the unrelativized case, we have

Theorem 2.2 (Relativized Step-Counter Theorem). For any total function G , the predicates $\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t)$ are G -computable.

In Chapter 4 we proved that the unrelativized predicates $\text{STP}^{(n)}$ are primitive recursive, but we do not need such a sharp result here. Instead, we modify the program in Fig. 2.1 by adding a variable Q that functions as a step counter. Then each time through the main loop, Q is increased by 1, so that the program will “know” when a given number of steps has been

	$Z \leftarrow X_{n+1} + 1$	
	$S \leftarrow \prod_{i=1}^n (p_{2i})^{X_i}$	
	$K \leftarrow 1$	
[C]	$Q = Q + 1$	(*)
	IF $Q > X_{n+2} + 1$ GOTO E	(*)
	IF $K = \text{Lt}(Z) + 1 \vee K = 0$ GOTO F	
	$U \leftarrow r((Z)_K)$	
	$P \leftarrow p_{r(U)+1}$	
	IF $l(U) = 0$ GOTO O	(‡)
	IF $l(U) = 1$ GOTO A	
	IF $\sim(P S)$ GOTO N	
	IF $l(U) = 2$ GOTO M	
	$K \leftarrow \min_{i \leq \text{Lt}(Z)} [l((Z)_i) + 2 = l(U)]$	
	GOTO C	
[O]	$W \leftarrow (S)_{r(U)+1}$	(‡)
	$B \leftarrow W$	(‡)
	$B \leftarrow O(B)$	(‡)
	$S \leftarrow \lfloor S/P^W \rfloor \cdot P^B$	(‡)
	GOTO N	(‡)
[M]	$S \leftarrow \lfloor S/P \rfloor$	
	GOTO N	
[A]	$S \leftarrow S \cdot P$	
[N]	$K \leftarrow K + 1$	
	GOTO C	
[F]	$Y \leftarrow 1$	(*)

Figure 2.2. Program that G -computes $\text{STP}_G^{(n)}(X_1, \dots, X_n, X_{n+1}, X_{n+2})$.

exceeded. The program is given in Fig. 2.2. The asterisks (*) indicate changes from the relativized universal program and the daggers (‡), as before, indicate the changes made in relativizing.

We shall now consider certain partial functions with finite domains, and use numbers as codes for them. For every $u \in N$ we define

$$\{u\}(i) = \begin{cases} (r(u))_{i+1} & \text{for } i < l(u) \\ \uparrow & \text{for } i \geq l(u). \end{cases} \quad (2.1)$$

Thus, if $l(u) = 0$, then $\{u\} = \emptyset$, the nowhere defined function. Also, if

$$u = \langle k, [a_0, a_1, \dots, a_{k-1}] \rangle,$$

then $\{u\}(x) = a_x$ for $x = 0, 1, \dots, k - 1$ and $\{u\}(x) \uparrow$ for $x \geq k$.

Theorem 2.3. The predicate

$$P(x_1, \dots, x_n, y, t, u) \Leftrightarrow \text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)$$

is computable.

Proof. We will transform the program in Fig. 2.2 into one that computes $P(x_1, \dots, x_n, x_{n+1}, x_{n+2}, x_{n+3})$. We need only replace the single oracle statement $B \leftarrow O(B)$ by instructions that operate on x_{n+3} to obtain the required information about $\{x_{n+3}\}$. This involves first testing for $\{x_{n+3}\}(b) \downarrow$, where b is the value of the variable B . If $\{x_{n+3}\}(b) \uparrow$, computation should halt with output 0, because there is no computation in this case. Otherwise B should be given the value $\{x_{n+3}\}(b)$. Thus, by (2.1), it suffices to replace the oracle statement $B \leftarrow O(B)$ in the program in Fig. 2.2 by the following pair of instructions:

```
IF  $l(X_{n+3}) \leq B$  GOTO E
     $B \leftarrow (r(X_{n+3}))_{B+1}$ 
```

■

Now, let G be a total function. Then, we define

$$u \prec G$$

to mean that $\{u\}(i) = G(i)$ for $0 \leq i < l(u)$. [Of course, by (2.1), $\{u\}(i) \uparrow$ for $i \geq l(u)$.] For a number u such that $u \prec G$, values of G can be retrieved by using the equations

$$G(i) = (r(u))_{i+1}, \quad i = 0, 1, \dots, l(u) - 1.$$

We can use the predicate $\text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)$ to obtain an important result that isolates the noncomputability of the relativized step-counter predicate in a way that will prove helpful. The simple observation on which this result capitalizes is that any G -computation can contain only finitely many oracle queries.

Theorem 2.4 (Finiteness Theorem). Let G be a total function. Then, we have

$$\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t) \Leftrightarrow (\exists u)[u \prec G \ \& \ \text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)].$$

Proof. First suppose that $\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t)$ is true for some given values of x_1, \dots, x_n, y, t , and let \mathcal{P} be the program with $\#(\mathcal{P}) = y$. Let s_1, s_2, \dots, s_k be a G -computation of \mathcal{P} where s_1 is the initial snapshot corresponding to the input values x_1, x_2, \dots, x_n and where $k \leq t + 1$. Let M be the largest value of an oracle query of this G -computation, and let $u = \langle M + 1, [G(0), G(1), \dots, G(M)] \rangle$. Thus, $u \prec G$ and $\{u\}(m) = G(m)$ for all $m \leq M$. Hence, s_1, s_2, \dots, s_k is likewise a $\{u\}$ -computation of \mathcal{P} . Since $k \leq t + 1$, $\text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)$ is true.

Conversely, let us be given $u \prec G$ such that $\text{STP}_{\{u\}}^{(n)}(x_1, \dots, x_n, y, t)$ is true, and let $\#(\mathcal{P}) = y$. Let s_1, s_2, \dots, s_k be a $\{u\}$ -computation of \mathcal{P} where s_1 is the initial snapshot corresponding to the input values x_1, x_2, \dots, x_n and where $k \leq t + 1$. For each m that is an oracle query of this $\{u\}$ -computation, we must have $\{u\}(m) \downarrow$, since otherwise one of the snapshots in this $\{u\}$ -computation would be nonterminal and yet not have a successor. Since $u \prec G$, we must have $\{u\}(m) = G(m)$ for all such m . Hence s_1, s_2, \dots, s_k is likewise a G -computation of \mathcal{P} . Since $k \leq t + 1$, $\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t)$ is true. ■

To conclude this section we turn to the parameter theorem (Theorem 5.1 in Chapter 4).

Theorem 2.5 (Relativized and Strengthened Parameter Theorem). For each $n, m > 0$, there is a primitive recursive function $S_m^n(u_1, \dots, u_n, y)$ such that for every total function G :

$$\Phi_G^{(m+n)}(x_1, \dots, x_m, u_1, \dots, u_n, y) = \Phi_G^{(m)}(x_1, \dots, x_m, S_m^n(u_1, \dots, u_n, y)). \quad (2.2)$$

Moreover, the functions S_m^n have the property:

$$S_m^n(u_1, \dots, u_n, y) = S_m^n(\bar{u}_1, \dots, \bar{u}_n, y) \quad \text{implies} \quad u_1 = \bar{u}_1, \dots, u_n = \bar{u}_n.$$

Proof. The functions S_m^n are defined exactly as in the proof of Theorem 5.1 in Chapter 4. We briefly give the proof again in a slightly different way. Thus, let $\#(\mathcal{P}) = y$; then the function $S_m^1(u, y)$ is defined to be the number of the program $\bar{\mathcal{P}}$ obtained from \mathcal{P} by preceding it by the statement

$$X_{m+1} \leftarrow X_{m+1} + 1$$

repeated u times. Since $\bar{\mathcal{P}}$ on inputs x_1, \dots, x_m will do exactly what \mathcal{P} would have done on inputs x_1, \dots, x_m, u we have

$$\Phi_G^{(m+1)}(x_1, \dots, x_m, u, y) = \Phi_G^{(m)}(x_1, \dots, x_m, S_m^1(u, y)),$$

the desired result for $n = 1$. To complete the proof, we define S_m^n for $n > 1$ by the recursion

$$S_m^{k+1}(u_1, \dots, u_k, u_{k+1}, y) = S_m^k(u_1, \dots, u_k, S_{m+k}^1(u_{k+1}, y)).$$

It is now easy to prove by induction on n that if $\#(\mathcal{P}) = y$, then $S_m^n(u_1, \dots, u_n, y) = \#(\bar{\mathcal{P}})$, where $\bar{\mathcal{P}}$ is obtained from \mathcal{P} by preceding it by the following program consisting of $u_n + \dots + u_1$ statements.

$$\left. \begin{array}{l} X_{m+1} \leftarrow X_{m+1} + 1 \\ \vdots \\ X_{m+1} \leftarrow X_{m+1} + 1 \\ \vdots \\ X_{m+n} \leftarrow X_{m+n} + 1 \\ \vdots \\ X_{m+n} \leftarrow X_{m+n} + 1 \end{array} \right\} u_1$$

Hence, $\tilde{\mathcal{P}}$ on inputs x_1, \dots, x_m will do exactly what \mathcal{P} would have done on inputs $x_1, \dots, x_m, u_1, \dots, u_n$. Thus, we obtain (2.2).

Finally, let

$$S_m^n(u_1, \dots, u_n, y) = S_m^n(\bar{u}_1, \dots, \bar{u}_n, y) = \#(\bar{\mathcal{P}}),$$

and let $y = \#(\mathcal{P})$. Then, $\bar{\mathcal{P}}$ consists of a list of increment statements followed by \mathcal{P} , and for $1 \leq i \leq n$, u_i and \bar{u}_i are both simply the number of times the statement

$$X_{m+i} \leftarrow X_{m+i} + 1$$

occurs in $\bar{\mathcal{P}}$ preceding \mathcal{P} . Thus, $u_i = \bar{u}_i$.

Exercises

1. (a) Show that the functions S_m^n do *not* have the property:

$$S_m^n(u_1, \dots, u_n, y) = S_m^n(\bar{u}_1, \dots, \bar{u}_n, \bar{y})$$

implies $u_1 = \bar{u}_1, \dots, u_n = \bar{u}_n, y = \bar{y}$.

- (b) Can the definition of S_m^n be modified so the parameter theorem continues to hold, but so the condition of (a) holds as well? How?

2. Prove the converse of Exercise 1.2.

3. Reducibility

If A and B are sets such that A is B -recursive, we also say that A is *Turing-reducible to B* and we write $A \leq_t B$. We have

Theorem 3.1. $A \leq_t A$. If $A \leq_t B$ and $B \leq_t C$, then $A \leq_t C$.

Proof. The first statement follows at once from Theorem 1.3 and the second from Theorem 1.5. ■

Any relation on the subsets of N for which Theorem 3.1 is true is called a *reducibility*. Many reducibilities have been studied. For example, we introduced many-one reducibility in Chapter 4. We can also define a restricted form of many-one reducibility.

Definition. We write $A \leq_1 B$ and say that A is *one-one reducible to B* if there is a one-one recursive function f (i.e., $f(x) = f(y)$ implies $x = y$) such that

$$A = \{x \in N \mid f(x) \in B\}.$$

Theorem 3.2. $A \leq_1 B$ implies $A \leq_m B$ implies $A \leq_t B$.

Proof. The first implication is immediate. For the second implication, let $A = \{x \in N \mid f(x) \in B\}$, where f is recursive. Then the following program B -computes A :

$$\begin{aligned} X &\leftarrow f(X) \\ X &\leftarrow O(X) \\ Y &\leftarrow X \end{aligned}$$

■

Theorem 3.3. \leq_1 and \leq_m are both reducibilities.

Proof. Clearly $A = \{x \in N \mid I(x) \in A\}$, where I is the identity function. Hence $A \leq_1 A$ and therefore $A \leq_m A$.

Let $A \leq_m B$ and $B \leq_m C$, and let

$$A = \{x \in N \mid f(x) \in B\}, \quad B = \{x \in N \mid g(x) \in C\},$$

where f, g are recursive. Then

$$A = \{x \in N \mid g(f(x)) \in C\},$$

so that $A \leq_m C$. If, moreover, f and g are one-one and $h(x) = g(f(x))$, then h is also one-one, because

$$\begin{aligned} h(x) = h(y) &\text{ implies } g(f(x)) = g(f(y)) \\ &\text{implies } f(x) = f(y) \\ &\text{implies } x = y. \end{aligned}$$

■

Thus, we have three examples, \leq_1 , \leq_m , and \leq_t , of reducibilities. Polynomial-time reducibility, \leq_p , which we will study in Chapter 15, is another example. (In fact, historically, polynomial-time reducibility was suggested by many-one reducibility.) There are a number of simple properties that all reducibilities share. To work some of these out, let us write \leq_Q to represent an arbitrary reducibility. By replacing Q by 1, m, t (or even p) we specialize to the particular reducibilities we have been studying. We write $A \not\leq_Q B$ to indicate that it is not the case that $A \leq_Q B$.

Definition. $A \equiv_Q B$ means that $A \leq_Q B$ and $B \leq_Q A$.

Theorem 3.4. For any reducibility \leq_Q :

$$\begin{aligned} A &\equiv_Q A, \\ A \equiv_Q B &\text{ implies } B \equiv_Q A, \\ A \equiv_Q B \text{ and } B \equiv_Q C &\text{ implies } A \equiv_Q C. \end{aligned}$$

Proof. Immediate from the definition. ■

Definition. Let \mathbf{W} be a collection of subsets of N and let \leq_Q be a reducibility. \mathbf{W} is called *Q-closed* if it has the property

$$A \in \mathbf{W} \text{ and } B \leq_Q A \text{ implies } B \in \mathbf{W}.$$

Also, a set $A \in \mathbf{W}$ is called *Q-complete for \mathbf{W}* if for every $B \in \mathbf{W}$ we have $B \leq_Q A$.

NP-completeness, which will be studied in Chapter 15, is, in the present terminology, *polynomial-time completeness for NP*. Completeness of a set A is often proved by showing that a set already known to be complete can be reduced to A .

Theorem 3.5. Let A be *Q*-complete for \mathbf{W} , let $B \in \mathbf{W}$, and let $A \leq_Q B$. Then B is *Q*-complete for \mathbf{W} .

Proof. Let $C \in \mathbf{W}$. Then $C \leq_Q A$. Hence $C \leq_Q B$. ■

If \mathbf{W} is a collection of subsets of N , we write

$$\text{co-}\mathbf{W} = \{A \subseteq N \mid \bar{A} \in \mathbf{W}\}.$$

Theorem 3.6. Let $\text{co-}\mathbf{W}$ be *Q*-closed, let A be *Q*-complete for \mathbf{W} , and let $A \in \text{co-}\mathbf{W}$. Then we have $\mathbf{W} = \text{co-}\mathbf{W}$.

Proof. Let $B \in \mathbf{W}$. Then, since A is Q -complete for \mathbf{W} , $B \leq_Q A$. Since $A \in \text{co-}\mathbf{W}$ and $\text{co-}\mathbf{W}$ is Q -closed, $B \in \text{co-}\mathbf{W}$. This proves that $\mathbf{W} \subseteq \text{co-}\mathbf{W}$.

Next let $B \in \text{co-}\mathbf{W}$. Then $\bar{B} \in \mathbf{W}$. By what has already been shown, $\bar{B} \in \text{co-}\mathbf{W}$. Hence $B \in \mathbf{W}$. This proves that $\text{co-}\mathbf{W} \subseteq \mathbf{W}$. ■

As we shall see, Theorem 3.6 is quite useful. Our applications will be to the case of one-one and many-one reducibility. For this purpose, it is useful to note

Theorem 3.7. If $A \leq_m B$, then $\bar{A} \leq_m \bar{B}$. Likewise if $A \leq_1 B$, then $\bar{A} \leq_1 \bar{B}$.

Proof. If $A = \{x \in N | f(x) \in B\}$, then clearly $\bar{A} = \{x \in N | f(x) \in \bar{B}\}$. ■

Corollary 3.8. If \mathbf{W} is m-closed or 1-closed, then so is $\text{co-}\mathbf{W}$.

Proof. Let $B \in \text{co-}\mathbf{W}$, $A \leq_m B$. By the theorem, $\bar{A} \leq_m \bar{B}$. Since $\bar{B} \in \mathbf{W}$ and \mathbf{W} is m-closed, $\bar{A} \in \mathbf{W}$. Hence $A \in \text{co-}\mathbf{W}$. Similarly for \leq_1 . ■

For a concrete example, we may take \mathbf{W} to be the collection of r.e. subsets of N . (For notation, the reader should review Chapter 4, Section 4.) We have

Theorem 3.9. K is 1-complete for the class of r.e. sets.

Proof. Let A be any r.e. set. We must show that $A \leq_1 K$. Since A is r.e., we have

$$A = \{x \in N | f(x) \downarrow\},$$

where f is a partially computable function. Let $g(t, x) = f(x)$ for all t, x . Thus, g is also partially computable. Using the (unrelativized) universality and parameter theorems, we have for a suitable number e :

$$g(t, x) = \Phi^{(2)}(t, x, e) = \Phi(t, S_1^1(x, e)).$$

Hence,

$$\begin{aligned} A &= \{x \in N | f(x) \downarrow\} \\ &= \{x \in N | g(S_1^1(x, e), x) \downarrow\} \\ &= \{x \in N | \Phi(S_1^1(x, e), S_1^1(x, e)) \downarrow\} \\ &= \{x \in N | S_1^1(x, e) \in K\}. \end{aligned}$$

Thus, $A \leq_m K$. But, by the strengthened version of the parameter theorem (Theorem 2.5), $S_1^1(x, e)$ is actually one-one. Hence, $A \leq_1 K$. ■

The class of r.e. sets is easily seen to be m-closed. Thus, let f be partially computable, let $A = \{x \in N \mid f(x) \downarrow\}$, and let $B = \{x \in N \mid g(x) \in A\}$, where g is computable. Then

$$B = \{x \in N \mid f(g(x)) \downarrow\},$$

so that B is r.e. Applying Theorems 3.2, 3.6, and 3.9 and Corollary 3.8, we obtain the not very interesting conclusion:

If \bar{K} is r.e., then the complement of every r.e. set is r.e.

Since we know that \bar{K} is in fact not r.e., this does us no good. However, Corollary 3.8 and Theorem 3.6 together with the fact that there is an r.e. set (e.g., K) whose complement is not r.e. permits us to conclude

Theorem 3.10. If A is m-complete for the class of r.e. sets, then \bar{A} is not r.e., so that A is not recursive.

We conclude this section with a simple but important construction. For $A, B \subseteq N$ we write

$$A \oplus B = \{2x \mid x \in A\} \cup \{2x + 1 \mid x \in B\}.$$

Intuitively, $A \oplus B$ contains the information in both A and B and nothing else. This suggests the truth of the following simple result.

Theorem 3.11. $A \leq_t A \oplus B$, $B \leq_t A \oplus B$. If $A \leq_t C$ and $B \leq_t C$, then $A \oplus B \leq_t C$.

Proof. The following program ($A \oplus B$)-computes A :

```
X ← 2X
X ← O(X)
Y ← X
```

If the first instruction is replaced by $X \leftarrow 2X + 1$, the program ($A \oplus B$)-computes B .

Finally, let C_A, C_B be the characteristic functions of A and B , respectively. Assuming that A and B are both C -computable, there must be programs that C -compute the functions C_A and C_B , respectively. Hence, we may use macros

$$Y \leftarrow C_A(X) \quad \text{and} \quad Y \leftarrow C_B(X)$$

in programs that have C available as oracle. Thus, the following program C -computes $A \oplus B$:

```

IF  $2 \mid X$  GOTO D
 $X \leftarrow \lfloor (X - 1)/2 \rfloor$ 
 $Y \leftarrow C_B(X)$ 
GOTO E
[D]       $X \leftarrow \lfloor X/2 \rfloor$ 
           $Y \leftarrow C_A(X)$ 

```

■

Exercises

- Let $U = \{x \in N \mid l(x) \in W_{r(x)}\}$. Show that U is 1-complete for the class of r.e. sets.
- Let $K \leq_t A$ and let

$$C = \{x \in K \mid \Phi_x(x) \notin A \oplus \bar{A}\}.$$

- Prove that $A \leq_1 C$, $C \leq_t A$, but $C \not\leq_m A$.
- Prove that Theorem 3.11 holds with \leq_t replaced by \leq_m .
 - Let $\text{FIN} = \{x \in N \mid W_x \text{ is finite}\}$. Prove that $K \leq_1 \text{FIN}$.
 - Prove that if $B, \bar{B} \neq \emptyset$, then for every recursive set A , $A \leq_m B$.

4. Sets r.e. Relative to an Oracle

If G is a total function (of one or more arguments) we say that a set $B \subseteq N$ is *G-recursively enumerable* (abbreviated *G-r.e.*) if there is a partially G -computable function g such that

$$B = \{x \in N \mid g(x)\downarrow\}.$$

By Theorem 1.6, r.e. sets are then simply sets that are *G-r.e.* for some computable function G .

It is easy to relativize the proofs in Chapter 4, Section 4, using, in particular, the relativized step-counter theorem. We give some of the results and leave the details to the reader.

Theorem 4.1. If B is a *G*-recursive set, then B is *G-r.e.*

Theorem 4.2. The set B is G -recursive if and only if B and \bar{B} are both G -r.e.

Theorem 4.3. If B and C are G -r.e. sets, so are $B \cup C$ and $B \cap C$.

Next, we obtain

Theorem 4.4. The set A is G -r.e. if and only if there is a G -computable predicate $Q(x, t)$ such that

$$A = \{x \in N \mid (\exists t)Q(x, t)\}. \quad (4.1)$$

Proof. First let A be G -r.e. Then, there is a partially G -computable function h such that

$$A = \{x \in N \mid h(x) \downarrow\}.$$

Writing $h(x) = \Phi_G(x, z_0)$, we have

$$A = \{x \in N \mid (\exists t)\text{STP}_G^{(1)}(x, z_0, t)\},$$

which gives the result in one direction.

Conversely, let (4.1) hold, where Q is a G -computable predicate. Let $h(x)$ be the partial function which is G -computed by the following program:

[B]	$Z \leftarrow Q(X, Y)$
	$Y \leftarrow Y + 1$
	IF $Z = 0$ GOTO B

Then clearly,

$$A = \{x \in N \mid h(x) \downarrow\},$$

so that A is G -r.e. ■

Corollary 4.5. The set A is G -r.e. if and only if there is a G -recursive set B such that

$$A = \{x \in N \mid (\exists y)(\langle x, y \rangle \in B)\}.$$

Proof. If B is G -recursive, then the predicate $\langle x, y \rangle \in B$ is G -computable (by Theorem 1.4) and hence, by the theorem, A is G -r.e.

Conversely, if A is G -r.e., we have a G -computable predicate Q such that (4.1) holds. Letting $B = \{z \in N \mid Q(l(z), r(z))\}$, B is (again by Theorem 1.4) G -recursive and

$$A = \{x \in N \mid (\exists y)(\langle x, y \rangle \in B)\}. \quad \blacksquare$$

For any unary function G , we write

$$W_n^G = \{x \in N \mid \Phi_G(x, n) \downarrow\}.$$

(Thus $W_n = W_n^I$.) For the remainder of this section, G will be a unary total function. We have at once

Theorem 4.6 (Relativized Enumeration Theorem). A set B is G -r.e. if and only if there is an n for which $B = W_n^G$.

We define

$$G' = \{n \in N \mid n \in W_n^G\}.$$

(Thus, $K = I'$.) G' is called the *jump* of G . We have

Theorem 4.7. G' is G -r.e. but not G -recursive.

This is just the relativization of Theorem 4.7, in Chapter 4, and the proof of that theorem relativizes easily. However, we include the details because of the importance of the result.

Proof of Theorem 4.7. Since

$$G' = \{n \in N \mid \Phi_G(n, n) \downarrow\},$$

the relativized universality theorem shows that G' is G -r.e. If $\overline{G'}$ were also G -r.e., we would have $\overline{G'} = W_i^G$ for some $i \in N$. Then

$$i \in G' \Leftrightarrow i \in W_i^G \Leftrightarrow i \in \overline{G'},$$

a contradiction. ■

Our next result is essentially a relativization of Theorem 3.9.

Theorem 4.8. The following assertions are all equivalent:

- a. $A \leq_1 G'$;
- b. $A \leq_m G'$;
- c. A is G -r.e.

Proof. It is obvious that assertion a implies b. To see that b implies c, let h be a recursive function such that

$$x \in A \quad \text{if and only if} \quad h(x) \in G'.$$

Then

$$x \in A \quad \text{if and only if} \quad \Phi_G(h(x), h(x)) \downarrow,$$

so that A is G -r.e.

Finally, to see that c implies a , let A be G -r.e., so that we can write

$$A = \{x \in N \mid f(x) \downarrow\},$$

where f is partially G -computable. Let $g(t, x) = f(x)$ for all t, x . By the relativized universality and parameter theorems, we have, for some number e ,

$$g(t, x) = \Phi_G^{(2)}(t, x, e) = \Phi_G(t, S_1^1(x, e)).$$

Hence,

$$\begin{aligned} A &= \{x \in N \mid f(x) \downarrow\} \\ &= \{x \in N \mid g(S_1^1(x, e), x) \downarrow\} \\ &= \{x \in N \mid \Phi_G(S_1^1(x, e), S_1^1(x, e)) \downarrow\} \\ &= \{x \in N \mid S_1^1(x, e) \in G'\}. \end{aligned}$$

Since, by Theorem 2.5, $S_1^1(x, e)$ is one-one, we have $A \leq_1 G'$. ■

Theorem 4.9. If F and G are total unary functions and F is G -recursive, then $F' \leq_1 G'$.

Proof. By Theorem 4.7, F' is F -r.e. That is, we can write

$$F' = \{x \in N \mid f(x) \downarrow\},$$

where f is partially F -computable. By Theorem 1.5, f is also partially G -computable. Hence F' is G -r.e. By Theorem 4.8, $F' \leq_1 G'$. ■

By iterating the jump operation, we can obtain a hierarchy of problems each of which is “more unsolvable” than the preceding one.

We write $G^{(n)}$ for the jump iterated n times. That is, we define

$$\begin{aligned} G^{(0)} &= G, \\ G^{(n+1)} &= (G^{(n)})'. \end{aligned}$$

We have

Theorem 4.10. $\emptyset^{(n+1)}$ is $\emptyset^{(n)}$ -r.e. but not $\emptyset^{(n)}$ -recursive.

Proof. Immediate from Theorem 4.7. ■

It should be noted that, by Theorem 4.9, $K \equiv_1 \emptyset'$, since I and \emptyset are both recursive and $K = I'$. Later we shall see that much more can be said along these lines.

Exercise

1. Show that there are sets A, B, C such that A is B -r.e. and B is C -r.e., but A is not C -r.e.

5. The Arithmetic Hierarchy

The arithmetic hierarchy, which we will study in this section, is one of the principle tools used in classifying unsolvable problems.

Definition. Σ_0 is the class of recursive sets. For each $n \in N$, Σ_{n+1} is the class of sets which are A -r.e. for some set A that belongs to Σ_n . For all n , $\Pi_n = \text{co-}\Sigma_n$, $\Delta_n = \Sigma_n \cap \Pi_n$.

Note that Σ_1 is the class of r.e. sets and that $\Sigma_0 = \Pi_0 = \Delta_0 = \Delta_1$ is the class of recursive sets.

Theorem 5.1. $\Sigma_n \subseteq \Sigma_{n+1}$, $\Pi_n \subseteq \Pi_{n+1}$.

Proof. For any set $A \in \Sigma_n$, A is A -r.e. and hence $A \in \Sigma_{n+1}$. The rest follows by taking complements. ■

Theorem 5.2. $\emptyset^{(n)} \in \Sigma_n$.

Proof. By induction. For $n = 0$ the result is obvious. The inductive step follows at once from Theorem 4.10. ■

Theorem 5.3. $A \in \Sigma_{n+1}$ if and only if A is $\emptyset^{(n)}$ -r.e.

Proof. If A is $\emptyset^{(n)}$ -r.e., it follows at once from Theorem 5.2 that $A \in \Sigma_{n+1}$.

We prove the converse by induction. If $A \in \Sigma_1$, then A is r.e., so, of course, A is \emptyset -r.e. Assume the result known for $n = k$ and let $A \in \Sigma_{k+2}$. Then A is B -r.e. for some $B \in \Sigma_{k+1}$. By the induction hypothesis, B is $\emptyset^{(k)}$ -r.e. By Theorem 4.8, $A \leq_1 B'$ and $B \leq_1 \emptyset^{(k+1)}$. By Theorem 4.9, $B' \leq_1 \emptyset^{(k+2)}$. Hence $A \leq_1 \emptyset^{(k+2)}$, and by Theorem 4.8 again, A is $\emptyset^{(k+1)}$ -r.e. ■

Corollary 5.4. For $n \geq 1$ the following are all equivalent:

$$A \leq_1 \emptyset^{(n)};$$

$$A \leq_m \emptyset^{(n)};$$

$$A \in \Sigma_n.$$

Proof. This follows at once from Theorems 4.8 and 5.3. ■

Corollary 5.5. For $n \geq 1$, $\emptyset^{(n)}$ is 1-complete for Σ_n .

Proof. Immediate from Theorem 5.2 and Corollary 5.4. ■

Corollary 5.6. For $n \geq 1$, Σ_n and Π_n are both m-closed and hence 1-closed.

Proof. Let $A \in \Sigma_n$, $B \leq_m A$. Then using Corollary 5.4 twice, $B \leq_m \emptyset^{(n)}$, and hence $B \in \Sigma_n$. This proves that Σ_n is m-closed. The result for Π_n is now immediate from Corollary 3.8. ■

Theorem 5.7. $A \in \Delta_{n+1}$ if and only if $A \leq_t \emptyset^{(n)}$.

Proof. Immediate from Theorems 4.2 and 5.3. ■

In particular, since $K \equiv_t \emptyset'$ (actually $K \equiv_1 \emptyset'$), Δ_2 consists of all sets that are K -recursive, that is, sets for which there are algorithms that can decide membership by making use of an oracle for the halting problem.

Theorem 5.8. $\Sigma_n \cup \Pi_n \subseteq \Delta_{n+1}$.

Proof. For $n = 0$, the inclusion becomes an equality, so we assume $n \geq 1$. If $A \in \Sigma_n$, then by Corollary 5.4, $A \leq_1 \emptyset^{(n)}$, so by Theorem 5.7, $A \in \Delta_{n+1}$. If $A \in \Pi_n$, then $\overline{A} \leq_1 \emptyset^{(n)}$. But clearly $A \leq_t \overline{A}$ (for example, by Theorem 1.4). Hence $A \leq_t \emptyset^{(n)}$ and by Theorem 5.7, $A \in \Delta_{n+1}$. ■

Theorem 5.9. For $n \geq 1$, $\emptyset^{(n)} \in \Sigma_n - \Delta_n$.

Proof. By Theorem 4.10, $\emptyset^{(n)}$ is not $\emptyset^{(n-1)}$ -recursive. ■

Theorem 5.10 (Kleene's Hierarchy Theorem). We have for $n \geq 1$

1. $\Delta_n \subset \Sigma_n$, $\Delta_n \subset \Pi_n$;
2. $\Sigma_n \subset \Sigma_{n+1}$, $\Pi_n \subset \Pi_{n+1}$;
3. $\Sigma_n \cup \Pi_n \subset \Delta_{n+1}$.

Proof.

1. By definition $\Delta_n \subseteq \Sigma_n$, $\Delta_n \subseteq \Pi_n$. By Theorem 5.9, $\emptyset^{(n)} \in \Sigma_n - \Delta_n$, and so $\emptyset^{(n)} \in \Pi_n - \Delta_n$. Thus the inclusions are proper.
2. By Theorem 5.1 we need show only that the inclusions are proper. But $\emptyset^{(n+1)} \in \Sigma_{n+1}$. If $\emptyset^{(n+1)} \in \Sigma_n$, by Theorem 5.8, $\emptyset^{(n+1)} \in \Delta_{n+1}$, contradicting Theorem 5.9. Likewise $\emptyset^{(n+1)} \in \Pi_{n+1} - \Pi_n$.

3. By Theorem 5.8, we need show only that the inclusion is proper. Let $A_n = \overline{\emptyset^{(n)}} \oplus \overline{\emptyset^{(n)}}$. We shall show that $A_n \in \Delta_{n+1} - (\Sigma_n \cup \Pi_n)$. By Theorem 3.11 (with $C = \emptyset^{(n)}$), we have $A_n \leq_t \overline{\emptyset^{(n)}}$. Hence $A_n \in \Delta_{n+1}$. Also,

$$\overline{\emptyset^{(n)}} = \{x \in N \mid 2x \in A_n\},$$

$$\overline{\overline{\emptyset^{(n)}}} = \{x \in N \mid 2x + 1 \in A_n\}.$$

Hence $\overline{\emptyset^{(n)}} \leq_1 A_n$, $\overline{\overline{\emptyset^{(n)}}} \leq_1 A_n$. Suppose that $A_n \in \Sigma_n$. Then, by Corollary 5.6, $\overline{\emptyset^{(n)}} \in \Sigma_n$, so that $\overline{\emptyset^{(n)}} \in \Delta_n$, contradicting Theorem 5.9. Likewise if $A_n \in \Pi_n$, then $\overline{\emptyset^{(n)}} \in \Pi_n$ and hence $\overline{\emptyset^{(n)}} \in \Delta_n$. ■

Since we have now seen that for all $n \geq 1$, $\Sigma_n \neq \text{co-}\Sigma_n$, and since we know that for $n \geq 1$, Σ_n and Π_n are each m-closed, we may apply Theorem 3.6 to obtain the following extremely useful result.

Theorem 5.11. If A is m-complete for Σ_n , then $A \notin \Pi_n$. Likewise, if A is m-complete for Π_n , then $A \notin \Sigma_n$.

6. Post's Theorem

In order to make use of the arithmetic hierarchy, we will employ an alternative characterization of the classes Σ_n , Π_n involving strings of quantifiers. This alternative formulation is most naturally expressed in terms of predicates rather than sets. Hence we will use the following terminology.

We first associate with each predicate $P(x_1, \dots, x_s)$ the set

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\}.$$

Then we say that P is Σ_n or that P is a Σ_n predicate to mean that $A \in \Sigma_n$. Likewise, we say that P is Π_n or Δ_n if $A \in \Pi_n$ or $A \in \Delta_n$, respectively. Notice that we continue to regard Σ_n and Π_n as consisting of subsets of N , and we will not speak of a predicate as being a member of Σ_n or Π_n .

Our terminology involves a slight anomaly for unary predicates. We have just defined $P(x)$ to be Σ_n (or Π_n) if the set $A = \{x \in N \mid P((x)_1)\}$ belongs to Σ_n (or Π_n), whereas it would be more natural to speak of $P(x)$ as being Σ_n (or Π_n) depending on whether $B = \{x \in N \mid P(x)\}$ belongs to Σ_n (or Π_n). Fortunately, there is really no conflict, for we have

Theorem 6.1. Let $B = \{x \in N \mid P(x)\}$. Then $P(x)$ is Σ_n if and only if $B \in \Sigma_n$. Likewise for Π_n, Δ_n .

Proof. For $n = 0$, the result is obvious, so assume that $n \geq 1$. $P(x)$ is Σ_n (or Π_n , or Δ_n) if and only if the set $A = \{x \in N \mid P((x)_1)\}$ belongs to Σ_n (or Π_n or Δ_n). Now,

$$A = \{x \in N \mid (x)_1 \in B\},$$

and

$$B = \{x \in N \mid 2^x \in A\}.$$

Thus $A \equiv_m B$. By Corollary 5.6, this gives the result. ■

Theorem 6.2. Let $P(x_1, \dots, x_s)$ be a Σ_n predicate and let

$$Q(t_1, \dots, t_k) \Leftrightarrow P(f_1(t_1, \dots, t_k), \dots, f_s(t_1, \dots, t_k)),$$

where f_1, \dots, f_s are computable functions. Then Q is also Σ_n . Likewise for Π_n .

Proof. Let

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\},$$

$$B = \{t \in N \mid Q((t)_1, \dots, (t)_k)\}.$$

We shall prove that $B \leq_m A$. It will thus follow that if $A \in \Sigma_n$ (or Π_n), then $B \in \Sigma_n$ (or Π_n), giving the desired result.

We have

$$\begin{aligned} t \in B &\Leftrightarrow Q((t)_1, \dots, (t)_k) \\ &\Leftrightarrow P(f_1((t)_1, \dots, (t)_k), \dots, f_s((t)_1, \dots, (t)_k)) \\ &\Leftrightarrow [f_1((t)_1, \dots, (t)_k), \dots, f_s((t)_1, \dots, (t)_k)] \in A, \end{aligned}$$

so that $B \leq_m A$. ■

Theorem 6.3. A predicate P is Σ_n (or Π_n) if and only if $\sim P$ is Π_n (or Σ_n).

Proof. $A = \{x \in N \mid P((x)_1, \dots, (x)_s)\}$ implies

$$\bar{A} = \{x \in N \mid \sim P((x)_1, \dots, (x)_s)\}.$$

Theorem 6.4. Let $P(x_1, \dots, x_s), Q(x_1, \dots, x_s)$ be Σ_n (or Π_n). Then the predicates $P \& Q$ and $P \vee Q$ are likewise Σ_n (or Π_n).

Proof. For $n = 0$, the result is obvious. Assume that $n \geq 1$ and let

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\},$$

$$B = \{x \in N \mid Q((x)_1, \dots, (x)_s)\},$$

$$C = \{x \in N \mid P((x)_1, \dots, (x)_s) \& Q((x)_1, \dots, (x)_s)\},$$

$$D = \{x \in N \mid P((x)_1, \dots, (x)_s) \vee Q((x)_1, \dots, (x)_s)\}.$$

Thus, $C = A \cap B$ and $D = A \cup B$. If P and Q are Σ_n , then $A, B \in \Sigma_n$. Thus, by Theorem 5.3, A and B are both $\emptyset^{(n-1)}$ -r.e. By Theorem 4.3, C and D are likewise $\emptyset^{(n-1)}$ -r.e., and so $P \& Q$ and $P \vee Q$ are Σ_n .

If P and Q are Π_n , then $A, B \in \Pi_n$ so that $\bar{A}, \bar{B} \in \Sigma_n$. By Theorems 4.3 and 5.3, $\bar{A} \cap \bar{B} = (\bar{A} \cup \bar{B}) \in \Sigma_n$ and $\bar{A} \cup \bar{B} = (\bar{A} \cap \bar{B}) \in \Sigma_n$. Hence $D, C \in \Pi_n$, so that both $P \vee Q$ and $P \& Q$ are Π_n . ■

Theorem 6.5. Let $Q(x_1, \dots, x_s, y)$ be Σ_n , $n \geq 1$, and let

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists y)Q(x_1, \dots, x_s, y).$$

Then P is also Σ_n .

Proof. Let

$$A = \{x \in N \mid Q((x)_1, \dots, (x)_s, (x)_{s+1})\},$$

$$B = \{x \in N \mid P((x)_1, \dots, (x)_s)\}.$$

We are given that $A \in \Sigma_n$, i.e., that A is $\emptyset^{(n-1)}$ -r.e., and we must show that B is likewise $\emptyset^{(n-1)}$ -r.e.

By Theorem 4.4, we may write

$$A = \{x \in N \mid (\exists t)R(x, t)\},$$

where R is $\emptyset^{(n-1)}$ -recursive. Hence,

$$\begin{aligned} Q(x_1, \dots, x_s, y) &\Leftrightarrow [x_1, \dots, x_s, y] \in A \\ &\Leftrightarrow (\exists t)R([x_1, \dots, x_s, y], t). \end{aligned}$$

Thus,

$$\begin{aligned} x \in B &\Leftrightarrow P((x)_1, \dots, (x)_s) \\ &\Leftrightarrow (\exists y)Q((x)_1, \dots, (x)_s, y) \\ &\Leftrightarrow (\exists y)(\exists t)R([(x)_1, \dots, (x)_s, y], t) \\ &\Leftrightarrow (\exists z)R([(x)_1, \dots, (x)_s, l(z)], r(z)). \end{aligned}$$

By Theorems 1.4 and 4.4, B is $\emptyset^{(n-1)}$ -r.e. ■

Theorem 6.6. Let $Q(x_1, \dots, x_s, y)$ be Π_n , $n \geq 1$, and let

$$P(x_1, \dots, x_s) \Leftrightarrow (\forall y)Q(x_1, \dots, x_s, y).$$

Then P is also Π_n .

Proof. $\sim P(x_1, \dots, x_s) \Leftrightarrow (\exists y)\sim Q(x_1, \dots, x_s, y)$. The result follows from Theorems 6.3 and 6.5. ■

The main result of this section is

Theorem 6.7 (Post's Theorem). A predicate $P(x_1, \dots, x_s)$ is Σ_{n+1} if and only if there is a Π_n predicate $Q(x_1, \dots, x_s, y)$ such that

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists y)Q(x_1, \dots, x_s, y). \quad (6.1)$$

Proof. If (6.1) holds, with Q a Π_n predicate, it is easy to see that P must be Σ_{n+1} . By Theorem 5.8, Q is certainly itself Σ_{n+1} , and therefore, by Theorem 6.5, P is Σ_{n+1} .

The converse is somewhat more difficult. Let us temporarily introduce the following terminology: we will say that a predicate $P(x_1, \dots, x_s)$ is \exists^{n+1} if it can be expressed in the form (6.1), where Q is Π_n . Then Post's theorem just says that the Σ_{n+1} and the \exists^{n+1} predicates are the same. We have already seen that all \exists^{n+1} predicates are Σ_{n+1} .

Lemma 1. If a predicate is Σ_n , then it is \exists^{n+1} .

Proof. For $n = 0$, the result is obvious. Let $n \geq 1$, and let $P(x_1, \dots, x_s)$ be Σ_n . Let

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\}.$$

Then A is $\emptyset^{(n-1)}$ -r.e., so by Theorem 4.4,

$$A = \{x \in N \mid (\exists t)R(x, t)\},$$

where R is $\emptyset^{(n-1)}$ -recursive. Thus

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists t)R([x_1, \dots, x_s], t).$$

It remains to show that $R([x_1, \dots, x_s], t)$ is Π_n . But in fact, by Theorem 1.4, $R([x_1, \dots, x_s], t)$ is \emptyset^{n-1} -recursive, so that it is actually Δ_n and hence certainly Π_n . ■

Lemma 2. If a predicate is Π_n , then it is \exists^{n+1} .

Proof. If $P(x_1, \dots, x_s)$ is Π_n , we need only set

$$Q(x_1, \dots, x_s, y) \Leftrightarrow P(x_1, \dots, x_s),$$

so that, of course,

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists y)Q(x_1, \dots, x_s, y).$$

Since

$$\{x \in N \mid Q((x)_1, \dots, (x)_s, (x)_{s+1})\} = \{x \in N \mid P((x)_1, \dots, (x)_s)\},$$

the predicate Q is also Π_n , which gives the result. ■

Lemma 3. If $P(x_1, \dots, x_s, z)$ is \exists^{n+1} and

$$Q(x_1, \dots, x_s) \Leftrightarrow (\exists z)P(x_1, \dots, x_s, z),$$

then Q is \exists^{n+1} .

Proof. We may write

$$P(x_1, \dots, x_s, z) \Leftrightarrow (\exists y)R(x_1, \dots, x_s, z, y),$$

where R is Π_n . Then

$$\begin{aligned} Q(x_1, \dots, x_s) &\Leftrightarrow (\exists z)(\exists y)R(x_1, \dots, x_s, z, y) \\ &\Leftrightarrow (\exists t)R(x_1, \dots, x_s, l(t), r(t)), \end{aligned}$$

which is \exists^{n+1} by Theorem 6.2. ■

Lemma 4. If P and Q are \exists^{n+1} , then so are $P \& Q$ and $P \vee Q$.

Proof. Let us write

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists y)R(x_1, \dots, x_s, y),$$

$$Q(x_1, \dots, x_s) \Leftrightarrow (\exists z)S(x_1, \dots, x_s, z),$$

where R and S are Π_n . Then

$$\begin{aligned} P(x_1, \dots, x_s) \& Q(x_1, \dots, x_s) &\Leftrightarrow (\exists y)(\exists z)[R(x_1, \dots, x_s, y) \\ &\quad \& S(x_1, \dots, x_s, z)] \end{aligned}$$

and

$$\begin{aligned} P(x_1, \dots, x_s) \vee Q(x_1, \dots, x_s) &\Leftrightarrow (\exists y)(\exists z)[R(x_1, \dots, x_s, y) \\ &\quad \vee S(x_1, \dots, x_s, z)]. \end{aligned}$$

The result follows from Theorem 6.4 and Lemmas 2 and 3. ■

Lemma 5. If $P(x_1, \dots, x_s, t)$ is \exists^{n+1} and

$$Q(x_1, \dots, x_s, y) \Leftrightarrow (\forall t)_{\leq y} P(x_1, \dots, x_s, t),$$

then Q is \exists^{n+1} .

Proof. Let

$$P(x_1, \dots, x_s, t) \Leftrightarrow (\exists z)R(x_1, \dots, x_s, t, z),$$

where R is Π_n . Thus,

$$\begin{aligned} Q(x_1, \dots, x_s, y) &\Leftrightarrow (\forall t)_{\leq y}(\exists z)R(x_1, \dots, x_s, t, z) \\ &\Leftrightarrow (\exists u)(\forall t)_{\leq y}R(x_1, \dots, x_s, t, (u)_{t+1}), \end{aligned}$$

where we are using the Gödel number $u = [z_0, z_1, \dots, z_y]$ to encode the sequence of values of z corresponding to $t = 0, 1, \dots, y$. Thus,

$$\begin{aligned} Q(x_1, \dots, x_s, y) &\Leftrightarrow (\exists u)(\forall t)[t > y \vee R(x_1, \dots, x_s, t, (u)_{t+1})] \\ &\Leftrightarrow (\exists u)S(x_1, \dots, x_s, y, u), \end{aligned}$$

where S is Π_n . For $n = 0$, we have used Theorem 6.3 from Chapter 3; and for $n > 0$, we have used the fact that the predicate $t > y$ is recursive (and hence certainly Π_n), and Theorems 6.2, 6.4, and 6.6. ■

We now recall from Section 2 that $u \prec G$ means that

$$\{u\}(i) = G(i) \quad \text{for } 0 \leq i < l(u).$$

Lemma 6. Let $R(x)$ be Σ_n . Then the predicate $u \prec R$ is \exists^{n+1} .

Proof. We have

$$\begin{aligned} u \prec R &\Leftrightarrow (\forall i)_{< l(u)}\{[(r(u))_{i+1} = 1 \& R(i)] \vee [(r(u))_{i+1} = 0 \& \sim R(i)]\} \\ &\Leftrightarrow l(u) = 0 \vee (\exists z)(z + 1 = l(u) \& (\forall i)_{\leq z}\{[(r(u))_{i+1} = 1 \& R(i)] \\ &\quad \vee [(r(u))_{i+1} = 0 \& \sim R(i)]\}). \end{aligned}$$

Thus, using Lemmas 1–5 and the fact that the predicate $\sim R(i)$ is Π_n , we have the result. ■

Proof of Theorem 6.7 (Post's Theorem) Concluded. Let $P(x_1, \dots, x_s)$ be any Σ_{n+1} predicate. Let

$$A = \{x \in N \mid P((x)_1, \dots, (x)_s)\}.$$

Then $A \in \Sigma_{n+1}$, which means that A is B -r.e. for some set $B \in \Sigma_n$. Let $R(x)$ be the characteristic function of B , so that by Theorem 6.1, R is Σ_n . Since A is B -r.e., we are able to write

$$A = \{x \in N \mid f(x) \downarrow\},$$

where f is partially B -computable. Let f be B -computed by a program with number y_0 . Then, using Theorem 2.4 (the finiteness theorem), we have

$$\begin{aligned} x \in A &\Leftrightarrow (\exists t) \text{STP}_R^{(1)}(x, y_0, t) \\ &\Leftrightarrow (\exists t)(\exists u)\{u \prec R \text{ & } \text{STP}_{\{u\}}^{(1)}(x, y_0, t)\}. \end{aligned}$$

Thus,

$$P(x_1, \dots, x_s) \Leftrightarrow (\exists t)(\exists u)\{u \prec R \text{ & } \text{STP}_{\{u\}}^{(1)}([x_1, \dots, x_s], y_0, t)\}.$$

Therefore by Theorem 2.3 and Lemmas 3, 4, and 6, P is \exists^{n+1} . ■

Now that we know that being Σ_{n+1} and \exists^{n+1} are the same, we may rewrite Lemma 5 as

Corollary 6.8. If $P(x_1, \dots, x_s, t)$ is Σ_n and

$$Q(x_1, \dots, x_s, y) \Leftrightarrow (\forall t)_{\leq y} P(x_1, \dots, x_s, t),$$

then Q is also Σ_n .

Also, we can easily obtain the following results.

Corollary 6.9. A predicate $P(x_1, \dots, x_s)$ is Π_{n+1} if and only if there is a Σ_n predicate $Q(x_1, \dots, x_s, y)$ such that

$$P(x_1, \dots, x_s) \Leftrightarrow (\forall y)Q(x_1, \dots, x_s, y).$$

Proof. Immediate from Post's theorem and Theorem 6.3. ■

Corollary 6.10. If $P(x_1, \dots, x_s, t)$ is Π_n , and

$$Q(x_1, \dots, x_s, y) \Leftrightarrow (\exists t)_{\leq y} P(x_1, \dots, x_s, t),$$

then Q is also Π_n .

Proof. Immediate from Corollary 6.8 and Theorem 6.3. ■

We are now in a position to survey the situation. We call a predicate $P(x_1, \dots, x_s)$ *arithmetic* if there is a recursive predicate $R(x_1, \dots, x_s, y_1, \dots, y_n)$ such that

$$P(x_1, \dots, x_s) \Leftrightarrow (Q_1 y_1)(Q_2 y_2) \cdots (Q_n y_n) R(x_1, \dots, x_s, y_1, \dots, y_n), \quad (6.2)$$

where each of Q_1, \dots, Q_n is either the symbol \exists or the symbol \forall . We say that the Q_i are *alternating* if for $1 \leq i < n$ when Q_i is \exists , then Q_{i+1} is \forall and vice versa. Then we have

Theorem 6.11.

- Every predicate that is Σ_n or Π_n for any n is arithmetic.
- Every arithmetic predicate is Σ_n for some n (and also Π_n for some n).
- A predicate is Σ_n (or Π_n) if and only if it can be represented in the form (6.2) with $Q_1 = \exists$ (or $Q_1 = \forall$) and the Q_i alternating.

Proof. Since Σ_0 and Π_0 predicates are just recursive, they are arithmetic. Proceeding by induction, if we know, for some particular n , that all Σ_n and Π_n predicates are arithmetic, then Theorem 6.7 and Corollary 6.9 show that the same is true for Σ_{n+1} and Π_{n+1} predicates. This proves a.

For b we proceed by induction on n , the number of quantifiers. For $n = 0$, we have a Σ_0 (and a Π_0) predicate. If the result is known for $n = k$, then it follows for $n = k + 1$ using Theorems 6.5–6.7 and Corollary 6.9.

Finally, c is easily proved by mathematical induction using Theorem 6.7 and Corollary 6.9. ■

7. Classifying Some Unsolvable Problems

We will now see how to apply the arithmetic hierarchy. We begin with the set

$$\text{TOT} = \{z \in N \mid (\forall x)\Phi(x, z) \downarrow\},$$

which consists of all numbers of programs which compute total functions. This set was discussed in Chapter 4, Section 6, where it was shown that TOT is not r.e. Without relying on this previous discussion, we shall obtain much sharper information about TOT.

We begin by observing that

$$\text{TOT} = \{z \in N \mid (\forall x)(\exists t)\text{STP}^{(1)}(x, z, t)\},$$

so that $\text{TOT} \in \Pi_2$. We shall prove

Theorem 7.1. TOT is 1-complete for Π_2 . Therefore, $\text{TOT} \notin \Sigma_2$.

Proof. The second assertion follows from the first by Theorem 5.11.

Since we know that $\text{TOT} \in \Pi_2$, it remains to show that for any $A \in \Pi_2$, we have $A \leq_1 \text{TOT}$. For $A \in \Pi_2$, we can write

$$A = \{w \in N \mid (\forall x)(\exists y)R(x, y, w)\},$$

where R is recursive. Let

$$h(x, w) = \min_y R(x, y, w),$$

so that h is partially computable. Let h be computed by a program with number e . Thus,

$$(\exists y)R(x, y, w) \Leftrightarrow h(x, w) \downarrow \Leftrightarrow \Phi^{(2)}(x, w, e) \downarrow \Leftrightarrow \Phi(x, S_1^1(w, e)) \downarrow,$$

where we have used the parameter theorem. Hence,

$$\begin{aligned} w \in A &\Leftrightarrow (\forall x)(\exists y)R(x, y, w) \\ &\Leftrightarrow (\forall x)[\Phi(x, S_1^1(w, e)) \downarrow] \\ &\Leftrightarrow S_1^1(w, e) \in \text{TOT}. \end{aligned}$$

Since, by Theorem 2.5, $S_1^1(w, e)$ is one-one, we can conclude that

$$A \leq_1 \text{TOT}. \quad \blacksquare$$

As a second simple example we consider

$$\text{INF} = \{z \in N \mid W_z \text{ is infinite}\}.$$

We have

$$z \in \text{INF} \Leftrightarrow (\forall x)(\exists y)(y > x \ \& \ y \in W_z).$$

Now

$$y \in W_z \Leftrightarrow (\exists t)\text{STP}^{(1)}(y, z, t),$$

and hence the predicate $y \in W_z$ is Σ_1 . Using Theorems 6.4 and 6.5, $(\exists y)(y > x \ \& \ y \in W_z)$ is also Σ_1 , and finally $\text{INF} \in \Pi_2$. We shall show that INF is also 1-complete for Π_2 . By Theorem 3.5, it suffices to show that $\text{TOT} \leq_1 \text{INF}$ since we already know that TOT is 1-complete for Π_2 .

To do this we shall obtain a recursive one-one function $f(x)$ such that

$$W_x = N \text{ implies } W_{f(x)} = N$$

and

$$W_x \neq N \text{ implies } W_{f(x)} \text{ is finite.} \quad (7.1)$$

Having done this we will be through since we will have

$$x \in \text{TOT} \Leftrightarrow f(x) \in \text{INF},$$

and therefore,

$$\text{TOT} \leq_1 \text{INF}.$$

The intuitive idea behind the construction of f is that program number $f(x)$ will “accept” a given input z if and only if program number x

“accepts” successively inputs $0, 1, \dots, z$. We can write this intuitive idea in the form of an equation as follows:

$$W_{f(x)} = \{z \in N \mid (\forall k)_{\leq z} (k \in W_x)\}.$$

Now it is a routine matter to use the parameter theorem to obtain f . We first note that, by Corollary 6.8, the predicate $(\forall k)_{\leq z} (k \in W_x)$ is Σ_1 . Hence, as earlier, there is a number e such that

$$\begin{aligned} (\forall k)_{\leq z} (k \in W_x) &\Leftrightarrow \Phi^{(2)}(z, x, e) \downarrow \\ &\Leftrightarrow \Phi(z, S_1^1(x, e)) \downarrow \\ &\Leftrightarrow z \in W_{S_1^1(x, e)}. \end{aligned}$$

Thus the desired function $f(x)$ is simply $S_1^1(x, e)$, which is one-one, as we know from Theorem 2.5.

This completes the proof that INF is 1-complete for Π_2 . Hence also, $\text{INF} \notin \Sigma_2$.

The following notation will be useful.

Definition. Let $A, B, C \subseteq N$. Then we write $A \leq_m (B, C)$ to mean that there is a recursive function f such that

$$x \in A \text{ implies } f(x) \in B$$

and

$$x \in \bar{A} \text{ implies } f(x) \in C.$$

If f is one-one we write $A \leq_1 (B, C)$.

Thus $A \leq_1 B$ is simply the assertion: $A \leq_1 (B, \bar{B})$.

It will be useful to note that by (7.1), we have actually proved

$$\text{TOT} \leq_1 (\text{TOT}, \overline{\text{INF}}). \quad (7.2)$$

Now, we have

Theorem 7.2. If $A \leq_1 (B, C)$, $B \subseteq D$, and $C \cap D = \emptyset$, then $A \leq_1 D$.

Proof. We have a recursive one-one function f such that

$$x \in A \text{ implies } f(x) \in B \text{ implies } f(x) \in D$$

and

$$x \in \bar{A} \text{ implies } f(x) \in C \text{ implies } f(x) \in \bar{D}. \quad \blacksquare$$

Our final example will classify a Σ_3 set, and is considerably more difficult than either of those considered so far.

Theorem 7.3. Let

$$\text{COF} = \{x \in N \mid \overline{W}_x \text{ is finite}\}.$$

Then COF is 1-complete for Σ_3 .

Lemma 1. $\text{COF} \in \Sigma_3$.

Proof.

$$\text{COF} = \{x \in N \mid (\exists n)(\forall k)(k \leq n \vee k \in W_x)\}.$$

Since the predicate in parentheses is Σ_1 , the result follows from Theorem 6.11. \blacksquare

We introduce the notation

$${}_n W_x = \{m \in N \mid \text{STP}^{(1)}(m, x, n)\}.$$

Intuitively, ${}_n W_x$ is the set of numbers that program number x “accepts” in $\leq n$ steps. Clearly,

$$W_x = \bigcup_{n \in N} ({}_{n+1} W_x).$$

We also define

$${}_n W_x^r = \{m < r \mid m \in {}_n W_x\}.$$

We write $L(n, x)$ to mean that

$${}_{n+1} W_x^n = {}_n W_x^n.$$

Clearly $L(n, x)$ is a recursive predicate. We write

$$R(x, n) \Leftrightarrow (\forall r)_{\leq n}(r \in W_x) \vee [L(n, x) \& (\exists k)_{\leq n}(k \notin {}_n W_x)].$$

Since $R(x, n)$ is Σ_1 we can use the parameter theorem, as in the previous example, to find a recursive one-one function $g(x)$ such that

$$W_{g(x)} = \{n \mid R(x, n)\}.$$

Lemma 2. If $x \in \text{TOT}$, then $g(x) \in \text{TOT}$. If $x \notin \text{INF}$, then $g(x) \in \text{COF} - \text{TOT}$.

Proof. If $x \in \text{TOT}$, then $W_x = N$, so that $(\forall r)_{\leq n}(r \in W_x)$ is true for all n . Hence $R(x, n)$ is true for all n , i.e., $W_{g(x)} = N$ and $g(x) \in \text{TOT}$.

Now let $x \notin \text{INF}$, i.e., W_x is finite. Therefore, there is a number n_0 such that for all $n > n_0$, we have

$$W_x = {}_n W_x^n$$

and

$$(\exists k)_{< n} (k \notin W_x).$$

Thus, for $n > n_0$,

$${}_{n+1} W_x^n = {}_n W_x^n,$$

i.e., $L(n, x)$ is true. Thus, $n > n_0$ implies that $R(x, n)$ is true, i.e., that $n \in W_{g(x)}$. We have shown that all sufficiently large integers belong to $W_{g(x)}$. Hence $g(x) \in \text{COF}$. It remains to show that $g(x) \notin \text{TOT}$.

Let s be the least number not in W_x . We consider two cases.

Case 1. $s \notin W_{g(x)}$. Then surely $g(x) \notin \text{TOT}$.

Case 2. $s \in W_{g(x)}$. That is, $R(x, s)$ is true. But $(\forall r)_{\leq s} (r \in W_x)$ must be false because $s \notin W_x$. Hence $L(s, x)$ must be true and $(\exists k)_{< s} (k \notin {}_s W_x)$. Now this number k is less than s , which is the least number not in W_x . Hence $k \in W_x$. Since $k \notin {}_s W_x$,

$$(\exists n)_{n \geq s} [k \notin {}_n W_x \ \& \ k \in {}_{n+1} W_x]. \quad (7.3)$$

Now we claim that this number $n \notin W_{g(x)}$, which will show that in this case also $g(x) \notin \text{TOT}$. Thus, suppose that $n \in W_{g(x)}$, i.e., that $R(x, n)$ is true. Since $s \notin W_x$ and $n \geq s$, the condition $(\forall r)_{\leq n} (r \in W_x)$ must be false. Thus we would have to have $L(n, x)$, i.e., ${}_{n+1} W_x^n = {}_n W_x^n$. But by (7.3), $k < s \leq n$, $k \in {}_{n+1} W_x$, and $k \notin {}_n W_x$. This is a contradiction. ■

Lemma 3. $\text{TOT} \leq_1 (\text{TOT}, \text{COF} - \text{TOT})$.

Proof. Let f be the recursive one-one function satisfying (7.1) and let g be as above. Let $h(x) = g(f(x))$. Then using Lemma 2 and (7.1), we have

$$x \in \text{TOT} \implies f(x) \in \text{TOT} \implies h(x) \in \text{TOT},$$

$$x \notin \text{TOT} \implies f(x) \notin \text{INF} \implies h(x) \in \text{COF} - \text{TOT}. \quad \blacksquare$$

Now let $A \in \Sigma_3$. We wish to show that $A \leq_1 \text{COF}$. By Post's theorem, we can write

$$x \in A \Leftrightarrow (\exists n) B(x, n),$$

where B is Π_2 . Using the pairing functions, let

$$C = \{t \in N \mid (\exists n)_{\leq l(t)} B(r(t), n)\}.$$

Thus, $C \in \Pi_2$. Theorem 7.1, $C \leq_1 \text{TOT}$. Hence, using Lemma 3, $C \leq_1 (\text{TOT}, \text{COF} - \text{TOT})$. Let θ be a recursive one-one function such that

$$\begin{aligned} t \in C &\text{ implies } \theta(t) \in \text{TOT}, \\ t \notin C &\text{ implies } \theta(t) \in \text{COF} - \text{TOT}. \end{aligned} \quad (7.4)$$

Consider the Σ_1 predicate $r(z) \in W_{\theta(\langle l(z), x \rangle)}$. Using the parameter theorem as usual, we can write this in the form $z \in W_{\psi(x)}$, where ψ is a one-one recursive function. Thus,

$$W_{\psi(x)} = \{\langle k, m \rangle \mid m \in W_{\theta(\langle k, x \rangle)}\}. \quad (7.5)$$

The theorem then follows at once from

Lemma 4. $x \in A$ if and only if $\psi(x) \in \text{COF}$.

Proof. Let $x \in A$. Then $B(x, n)$ is true for some least value of n . Hence, for all $k \geq n$, we have $\langle k, x \rangle \in C$. By (7.4), $\theta(\langle k, x \rangle) \in \text{TOT}$ for all $k \geq n$. Since n is the least value for which $B(x, n)$ is true, $B(x, k)$ is false for $k < n$. Hence, for $k < n$, $\langle k, x \rangle \notin C$. Thus, by (7.4), $\theta(\langle k, x \rangle) \in \text{COF} - \text{TOT}$. To recapitulate,

$$k \geq n \text{ implies } \theta(\langle k, x \rangle) \in \text{TOT},$$

and

$$k < n \text{ implies } \theta(\langle k, x \rangle) \in \text{COF} - \text{TOT}. \quad (7.6)$$

Thus, by (7.5) we see that for $k \geq n$, $\langle k, m \rangle \in W_{\psi(x)}$ for all m . For each $k < n$, $W_{\theta(\langle k, x \rangle)}$ contains all but a finite set of m . Thus, altogether, $W_{\psi(x)}$ can omit at most finitely many integers, i.e., $\psi(x) \in \text{COF}$.

Now, let $x \notin A$. Then, $B(x, n)$ is false for all n . Therefore, $\langle k, x \rangle \notin C$ for all k . By (7.4),

$$\theta(\langle k, x \rangle) \in \text{COF} - \text{TOT} \text{ for all } k \in N,$$

and thus certainly,

$$\theta(\langle k, x \rangle) \notin \text{TOT} \text{ for all } k \in N.$$

That is, for every $k \in N$, there exists m such that $m \notin W_{\theta(\langle k, x \rangle)}$, i.e., by (7.5), such that $\langle k, m \rangle \notin W_{\psi(x)}$. Thus, $\overline{W}_{\psi(x)}$ is infinite, and hence $\psi(x) \notin \text{COF}$. ■

Exercises

1. Show that the following sets belong to Σ_3 .

- (a) $\{x \in N \mid \text{there is a recursive function } f \text{ such that } \Phi_x \subseteq f\}$.
- (b) $\{\langle x, y \rangle \mid x \in N \& y \in N \& W_x - W_y \text{ is finite}\}$.

2. (a) Prove that for each m, n there is a predicate $U(x_1, \dots, x_m, y)$ which is Σ_n , such that for every Σ_n predicate $P(x_1, \dots, x_m)$ there is a number y_0 with

$$P(x_1, \dots, x_m) \Leftrightarrow U(x_1, \dots, x_m, y_0).$$

- (b) State and prove a similar result for Π_n .
3. Use the previous exercise to prove that for each n , $\Pi_n - \Sigma_n \neq \emptyset$.

8. Rice's Theorem Revisited

In Chapter 4, we gave a proof of Rice's theorem (Theorem 7.1) using the original parameter theorem. We get a somewhat stronger result using the strengthened form of the parameter theorem.

Definition. Let Γ be a set of partially computable functions of one variable. As in Chapter 4, Section 7, we write

$$R_\Gamma = \{t \in N \mid \Phi_t \in \Gamma\}.$$

We call Γ *nontrivial* if $\Gamma \neq \emptyset$ and there is at least one partially computable function $g(x)$ such that $g \notin \Gamma$.

Theorem 8.1 (Strengthened Form of Rice's Theorem). Let Γ be a nontrivial collection of partially computable functions of one variable. Then, $K \leq_1 R_\Gamma$ or $\bar{K} \leq_1 R_\Gamma$, so that R_Γ is not recursive.

Thus not only is R_Γ nonrecursive, but the halting problem can be “solved” using R_Γ as an oracle. Actually, the first proof of Rice's theorem already shows that either $K \leq_m R_\Gamma$ or $\bar{K} \leq_m R_\Gamma$. We give essentially the same proof here, using the strengthened form of the parameter theorem to upgrade the result to one-one reducibility.

Proof. We recall (Chapter 1, Section 2) that \emptyset is a partially computable function, namely, the nowhere defined function.

Case 1. $\emptyset \notin \Gamma$. Since Γ is nontrivial, it contains at least one function, say f . Since $f \in \Gamma$ and $\emptyset \notin \Gamma$, $f \neq \emptyset$; f must be defined for at least one value. Let

$$\Omega(x, t) = \begin{cases} f(t) & \text{if } x \in K \\ \uparrow & \text{if } x \notin K. \end{cases}$$

Since

$$x \in K \Leftrightarrow \Phi(x, x) \downarrow,$$

it is clear that Ω is partially computable. Using the parameter theorem in its strengthened form, we can write

$$\Omega(x, t) = \Phi_{g(x)}(t),$$

where g is a one-one recursive function. Then we have

$$x \in K \text{ implies } \Phi_{g(x)} = f \text{ implies } g(x) \in R_\Gamma;$$

$$x \notin K \text{ implies } \Phi_{g(x)} = \emptyset \text{ implies } g(x) \notin R_\Gamma.$$

Thus, $K \leq_1 R_\Gamma$.

Case 2. $\emptyset \in \Gamma$. Now let Δ be the class of all partially computable functions not in Γ . Thus, $R_\Gamma = \bar{R}_\Delta$ and $\emptyset \notin \Delta$. By Case 1, $K \leq_1 R_\Delta$, and hence by Theorem 3.7, $\bar{K} \leq_1 R_\Gamma$. ■

Exercises

1. State and prove a relativized version of Rice's theorem.
2. (a) Develop a code for partial functions from N to N with finite domains, writing f_n for the n th such function.
 (b) Prove the Rice–Shapiro theorem: R_Γ is r.e. if and only if $\Gamma = \emptyset$ or there is a recursive function $t(x)$ such that

$$\Gamma = \{g \mid (\exists x)(g \supseteq f_{t(x)})\}.$$

9. Recursive Permutations

Definition. A one-one recursive function f whose domain and range are both N is called a *recursive permutation*.

With each recursive permutation f we may associate its *inverse* f^{-1} :

$$f^{-1}(t) = \min_x (t = f(x)).$$

Then, f^{-1} is clearly likewise a recursive permutation.

Definition. Let $A, B \subseteq N$. Then A and B are said to be *recursively isomorphic*, written $A \equiv B$, if there is a recursive permutation f such that $x \in A$ if and only if $f(x) \in B$.

Since a recursive permutation provides what is essentially a mere change of notation, recursively isomorphic sets may be thought of as containing the same “information” presented in different notation.

It is obvious that $A \equiv B$ implies $A \equiv_1 B$. Remarkably, the converse statement is also true.

Theorem 9.1 (Myhill). If $A \equiv_1 B$, then $A \equiv B$.

In our proof of this theorem we shall need to code sequences of ordered pairs of numbers. We shall speak of *the code* of the sequence

$$(a_1, b_1), \dots, (a_n, b_n) \quad (9.1)$$

of pairs of elements of N meaning the number

$$u = \langle n, [\langle a_1, b_1 \rangle, \dots, \langle a_n, b_n \rangle] \rangle.$$

Thus, the numbers a_i, b_i can be retrieved from the code u by using the relations

$$\left. \begin{array}{l} a_i = l((r(u))_i) \\ b_i = r((r(u))_i) \end{array} \right\} \quad i = 1, 2, \dots, l(u).$$

Note that every natural number is the code of a unique finite (possibly empty) sequence of ordered pairs.

We say that the finite sequence (9.1) *associates* A and B , where $A, B \subseteq N$, if

1. $a_i \neq a_j$ for $1 \leq i < j \leq n$;
2. $b_i \neq b_j$ for $1 \leq i < j \leq n$;
3. for each i , $1 \leq i \leq n$, either $a_i \in A$ and $b_i \in B$ or $a_i \notin A$ and $b_i \notin B$.

We shall prove the

Lemma. Let $A \leq_1 B$. Then there is a computable function $k(u, v)$ such that if u codes the sequence (9.1) that associates A and B and $a \notin \{a_1, a_2, \dots, a_n\}$, then there is a b such that $k(u, a)$ codes the sequence

$$(a_1, b_1), \dots, (a_n, b_n), (a, b) \quad (9.2)$$

that also associates A and B .

Proof. Let f be a recursive one-one function such that

$$x \in A \quad \text{if and only if} \quad f(x) \in B. \quad (9.3)$$

We provide an algorithm for computing b from u and a . $k(u, a)$ can then be set equal to the code of (9.2), i.e.

$$k(u, a) = \langle l(u) + 1, r(u) \cdot p_{l(u)+1}^{\langle a, b \rangle} \rangle.$$

The numbers $f(a_1), f(a_2), \dots, f(a_n), f(a)$ are all distinct, because f is one-one. Hence, at least one of these $n + 1$ numbers does not belong to the set $\{b_1, b_2, \dots, b_n\}$. Our algorithm for obtaining b begins by computing $f(a)$. If $f(a) \notin \{b_1, b_2, \dots, b_n\}$, we set $b = f(a)$. Otherwise, $f(a) = b_i$ for some i and we try $f(a_i)$, because

$$a \in A \Leftrightarrow f(a) = b_i \in B \Leftrightarrow a_i \in a_i \in A \Leftrightarrow f(a_i) \in B.$$

If $f(a_i) \notin \{b_1, b_2, \dots, b_n\}$, we set $b = f(a_i)$. Otherwise, if $f(a_i) = b_j$, we continue the process, trying $f(a_j)$. By 1 and 2, none of the a_i and b_i obtained in this way duplicate previous ones. Thus, by our earlier remark the process must terminate in a value b . Using (9.3), we see that either $a \in A$ and $b \in B$ or $a \notin A$ and $b \notin B$. ■

Proof of Theorem 9.1. Since $A \leq_1 B$, by the Lemma there is a computable function $k(u, v)$ such that if u codes (9.1) that associates A and B and $a \notin \{a_1, a_2, \dots, a_n\}$, then for some b , $k(u, a)$ codes the sequence (9.2) that also associates A and B . But since $B \leq_1 A$, we can also apply the Lemma to obtain a computable function $\bar{k}(u, v)$ such that if u codes (9.1) that associates A and B and $b \notin \{b_1, b_2, \dots, b_n\}$, then for some a , $\bar{k}(u, b)$ codes the sequence (9.2) that likewise associates A and B .

We let $\nu(0) = 0$, which codes the empty sequence. (Note that the empty sequence does associate A and B .) We let

$$\begin{aligned} \nu(2x + 1) &= \begin{cases} \nu(2x) & \text{if } x \text{ is one of the left components} \\ & \text{of the sequence coded by } \nu(2x) \\ k(\nu(2x), x) & \text{otherwise;} \end{cases} \\ \nu(2x + 2) &= \begin{cases} \nu(2x + 1) & \text{if } x \text{ is one of the right components} \\ & \text{of the sequence coded by } \nu(2x + 1) \\ \bar{k}(\nu(2x + 1), x) & \text{otherwise.} \end{cases} \end{aligned}$$

Thus, we have

1. ν is a computable function.
2. For each x , $\nu(x)$ codes a sequence that associates A and B .
3. The sequence coded by $\nu(x + 1)$ is identical to, or is an extension of, the sequence coded by $\nu(x)$.
4. For each $a \in N$, there is an x such that a pair (a, b) occurs in the sequence coded by $\nu(x)$. (In fact, we can take $x = 2a + 1$.)
5. For each $b \in N$, there is an x such that a pair (a, b) occurs in the sequence coded by $\nu(x)$. (In fact, we can take $x = 2b + 2$.)

We now define the function f by setting $f(a)$ to be the number b such that the pair (a, b) appears in the sequence coded by some $\nu(x)$. b is uniquely determined because all the $\nu(x)$ code sequences that associate A and B , f is clearly computable. In fact,

$$f(a) = \min_b (\exists i)_{\leq l(\nu(2a+1))} [(r(\nu(2a+1)))_i = \langle a, b \rangle].$$

By 5, the range of f is N ; thus f is a recursive permutation and hence, $A \equiv B$. ■

Exercises

1. Prove that $K \equiv U$, where U is defined in Exercise 3.1.
2. Prove that

$$A \oplus \overline{A} \equiv \overline{A \oplus \overline{A}}.$$

Part 2

*Grammars and
Automata*

9

Regular Languages

1. Finite Automata

Computability theory, discussed in Part 1, is the theory of computation obtained when limitations of space and time are deliberately ignored. In automata theory, which we study in this chapter, computation is studied in a context in which bounds on space and time are entirely relevant. The point of view of computability theory is exemplified in the behavior of a Turing machine (Chapter 6) in which a read–write head moves back and forth on an infinite tape, with no preset limit on the number of steps required to reach termination.¹ At the opposite pole, one can imagine a device which moves from left to right on a finite input tape, and it is just such devices, the so-called *finite automata*, that we will now study. Since a finite automaton will have only one opportunity to scan each square in its motion from left to right, nothing is to be gained by permitting the device to “print” new symbols on its tape.

Unlike modern computers, whose action is controlled in part by an internally stored list of instructions called a *program*, the computing

¹ The present chapter does not depend on familiarity with the material in Chapters 2–8. Any exercises that refer to earlier material are marked with an *.

Table 1.1

δ	a	b
q_1	q_2	q_4
q_2	q_2	q_3
q_3	q_4	q_3
q_4	q_4	q_4

devices we will consider in this chapter have no such programs and no internal memory for storing either programs or partial results. In addition, since, as we just indicated, a finite automaton is permitted only a single pass over the tape, there is no external memory available. Instead, there are internal *states* that control the automaton's behavior and also function as memory in the sense of being able to retain some information about what has been read from the input tape up to a given point.

Thus, a finite automaton can be thought of as a very limited computing device which, after reading a string of symbols on the input tape, either accepts the input or rejects it, depending upon the state the machine is in when it has finished reading the tape.

The machine begins by reading the leftmost symbol on the tape, in a specified state called the *initial state* (the automaton is in this state whenever it is initially “turned on”). If at a given time, the machine is in a state q_i reading a given symbol s_j on the input tape, the device moves one square to the right on the tape and enters a state q_k . The current state of the automaton plus the symbol on the tape being read completely determine the automaton's next state.

Definition. A *finite automaton* \mathcal{M} on the alphabet² $A = \{s_1, \dots, s_n\}$ with states $Q = \{q_1, \dots, q_m\}$ is given by a function δ that maps each pair (q_i, s_j) , $1 \leq i \leq m$, $1 \leq j \leq n$, into a state q_k , together with a set $F \subseteq Q$. One of the states, usually q_1 , is singled out and called the *initial state*. The states belonging to the set F are called the *final* or *accepting* states, δ is called the *transition function*.

We can represent the function δ using a state versus symbol table. An example is given in Table 1.1, where the alphabet is $\{a, b\}$, $F = \{q_3\}$, and q_1

² For an introduction to alphabets and strings, see Chapter 1, Section 3.

is the initial state. It is easy to check that for the tapes

a	a	b	b	b
b	a	b	a	
a	a	b	a	
a	b	b	b	

the automaton will terminate in states q_3 , q_4 , q_4 , and q_3 , respectively. We shall say that the automaton *accepts* the strings $aabb$ and $abbb$ (because $q_3 \in F$), while it *rejects* the strings $baba$ and $aaba$ (because $q_4 \notin F$), i.e., that it *accepts* the first and fourth of the preceding tapes and *rejects* the second and third.

To proceed more formally, let \mathcal{M} be a finite automaton with transition function δ , initial state q_1 , and accepting states F . If q_i is any state of \mathcal{M} and $u \in A^*$, where A is the alphabet of \mathcal{M} , we shall write $\delta^*(q_i, u)$ for the state which \mathcal{M} will enter if it begins in state q_i at the left end of the string u and moves across u until the entire string has been processed. A formal definition by recursion is

$$\delta^*(q_i, 0) = q_i,$$

$$\delta^*(q_i, us_j) = \delta(\delta^*(q_i, u), s_j).$$

Obviously, $\delta^*(q_i, s_j) = \delta(q_i, s_j)$. Then we say that \mathcal{M} *accepts* a word u provided that $\delta^*(q_1, u) \in F$. \mathcal{M} *rejects* u means that $\delta^*(q_1, u) \in Q - F$. Finally, the *language accepted by \mathcal{M}* , written $L(\mathcal{M})$, is the set of all $u \in A^*$ accepted by \mathcal{M} :

$$L(\mathcal{M}) = \{u \in A^* \mid \delta^*(q_1, u) \in F\}.$$

A language is called *regular* if there exists a finite automaton that accepts it.

It is important to realize that the notion of regular language does not depend on the particular alphabet. That is, if $L \subseteq A^*$ and $A \subseteq B$, then there is an automaton on the alphabet A that accepts L if and only if there is one on the alphabet B that accepts L . That is, an automaton with alphabet B can be contracted to one on the alphabet A by simply restricting the transition function δ to A ; clearly this will have no effect

on which elements of A^* are accepted. Likewise, an automaton \mathcal{M} with alphabet A can be expanded to one with alphabet B by introducing a new “trap” state q and decreeing

$$\delta(q_i, b) = q \text{ for all states } q_i \text{ of } \mathcal{M} \text{ and all } b \in B - A,$$

$$\delta(q, b) = q \text{ for all } b \in B.$$

Leaving the set of accepting states unchanged (so that q is not an accepting state), we see that the expanded automaton accepts the same language as \mathcal{M} .

Returning to the automaton given by Table 1.1 with $F = \{q_3\}$, it is easy to see that the language it accepts is

$$\{a^{[n]}b^{[m]} \mid n, m > 0\}. \quad (1.1)$$

Thus we have shown that (1.1) is a regular language.

We conclude this section by mentioning another way to represent the transition function δ . We can draw a graph in which each state is represented by a vertex. Then, the fact that $\delta(q_i, s_j) = q_k$ is represented by drawing an arrow from vertex q_i to vertex q_k and labeling it s_j . The diagram thus obtained is called the *state transition diagram* for the given automaton. The state transition diagram for the transition function of Table 1.1 is shown in Fig. 1.1.

Exercises

1. In each of the following examples, an alphabet A and a language L are indicated with $L \subseteq A^*$. In each case show that L is regular by constructing a finite automaton \mathcal{M} that accepts L .

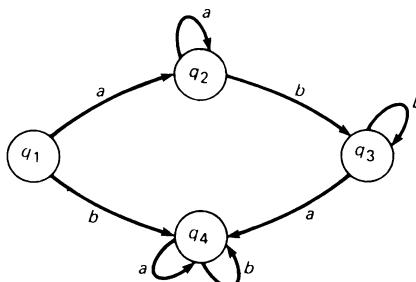


Figure 1.1

- (a) $A = \{1\}; L = \{1^{[6k]}1 \mid k \geq 0\}$.
- (b) $A = \{a, b\}$; L consists of all words whose final four symbols form the string $bbab$.
- (c) $A = \{a, b\}$; L consists of all words whose final five symbols include two a 's and three b 's.
- (d) $A = \{0, 1\}$; L consists of all strings that, when considered as binary numbers, have a value which is an integral multiple of 5.
- (e) $A = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\}$;

\mathcal{M} is to be a binary addition checker in the sense that it accepts strings of binary triples

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix} \cdots \begin{pmatrix} a_n \\ b_n \\ c_n \end{pmatrix}$$

such that $c_1c_2 \cdots c_n$ is the sum of $a_1a_2 \cdots a_n$ and $b_1b_2 \cdots b_n$ when each is treated as a binary number.

- (f) $A = \{a, b, c\}$. A *palindrome* is a word such that $w = w^R$. That is, it reads the same backward and forward. L consists of all palindromes of length less than or equal to 6.
 - (g) $A = \{a, b\}$; L consists of all strings $s_1s_2 \cdots s_n$ such that $s_{n-2} = b$. (Note that L contains no strings of length less than 3.)
 - (h) $A = \{a, b\}$; L consists of all words in which three a 's occur consecutively.
 - (i) $A = \{a, b\}$; L consists of all words in which three a 's do not occur consecutively.
2. (a) Suppose that the variable names in your favorite programming language are words w on the alphabet $\{A, \dots, Z, 0, \dots, 9\}$ such that $1 \leq |w| \leq 8$ and such that the first symbol of w belongs to $\{A, \dots, Z\}$. Give a finite automaton that accepts the language consisting of these variable names.
- (b) Now, remove the restriction $|w| \leq 8$ and give a finite automaton that accepts this extended language.
3. Describe the language accepted by each of the following finite automata. In each case the initial state is q_1 .

(a)

δ_1	a	b	c	
q_1	q_2	q_3	q_4	
q_2	q_2	q_4	q_5	
q_3	q_4	q_3	q_5	
q_4	q_4	q_4	q_4	
q_5	q_4	q_4	q_5	

$$F_1 = \{q_5\}.$$

(b) $\delta_2 = \delta_1$, $F_2 = \{q_4\}$.

(c)

δ_3	a	b	c	
q_1	q_2	q_2	q_1	
q_2	q_3	q_2	q_1	
q_3	q_1	q_3	q_2	

$$F_3 = \{q_2\}.$$

4. Let $A = \{s_1, \dots, s_n\}$. How many finite automata are there on A with exactly m states, $m > 0$?
5. Show that there is a regular language that is not accepted by any finite automaton with just one accepting state.
6. For any regular language L , define $\text{rank}(L) =$ the least number n such that L is accepted by some finite automaton with n states. Prove that for every $n > 0$ there is a regular language L with $\text{rank}(L) = n$.
7. Prove or disprove the following: If L_1, L_2 are regular languages such that $L_1 \subseteq L_2$, then $\text{rank}(L_1) \leq \text{rank}(L_2)$.
- 8.* Let \mathcal{M} be a finite automaton on the alphabet $A = \{s_1, \dots, s_n\}$ with states $Q = \{q_1, \dots, q_m\}$, transition function δ , initial state q_1 , and accepting states F . Give a Turing machine \mathcal{M}' that accepts $L(\mathcal{M})$.

2. Nondeterministic Finite Automata

Next we modify the definition of a finite automaton to permit transitions at each stage to either zero, one, or more than one states. Formally, we accomplish this by altering the definition of a finite automaton in the previous section by making the values of the transition function δ be *sets of states*, i.e., *sets of elements of Q* (rather than members of Q). The devices

Table 2.1

δ	a	b
q_1	$\{q_1, q_2\}$	$\{q_1, q_3\}$
q_2	$\{q_4\}$	\emptyset
q_3	\emptyset	$\{q_4\}$
q_4	$\{q_4\}$	$\{q_4\}$

so obtained are called *nondeterministic finite automata* (*ndfa*), and sometimes ordinary finite automata are then called *deterministic finite automata* (*dfa*). An ndfa on a given alphabet A with set of states Q is specified by giving such a transition function δ [which maps each pair (q_i, s_j) into a possibly empty subset of Q] and a fixed subset F of Q . For an ndfa, we define

$$\delta^*(q_i, 0) = \{q_i\},$$

$$\delta^*(q_i, us_j) = \bigcup_{q \in \delta^*(q_i, u)} \delta(q, s_j).$$

Thus, in calculating $\delta^*(q_i, u)$, one accumulates *all* states that the automaton can enter when it reaches the right end of u , beginning at the left end of u in state q_i . An ndfa \mathcal{M} with initial state q_1 accepts $u \in A^*$ if $\delta^*(q_1, u) \cap F \neq \emptyset$, i.e., if at least one of the states at which \mathcal{M} ultimately arrives belongs to F . Finally, $L(\mathcal{M})$, the language accepted by \mathcal{M} , is the set of all strings accepted by \mathcal{M} .

An example is given in Table 2.1 and Figure 2.1. Here $F = \{q_4\}$. It is not difficult to see that this ndfa accepts a string on the alphabet $\{a, b\}$ just in case at least one of the symbols has two successive occurrences in the string.

In state q_1 , if the next character read is an a , then there are two possibilities. It might be that this a is the first of the desired pair of a 's. In that case we would want to remember that we had found one a and hence

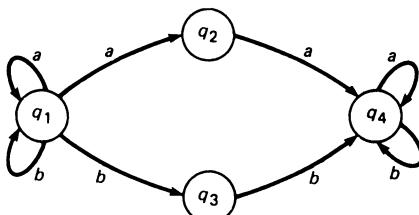


Figure 2.1

enter state q_2 to record that fact. On the other hand, it might be that the symbol following this a will be a b . Then this a is of no help in attaining the desired goal and hence we would remain in q_1 . Since we are not able to look ahead in the string, we cannot at this point determine which role the current a is playing and so the automaton “simultaneously” hypothesizes both possibilities. If the next character read is b , then since there is no transition from q_2 reading b , the choice has been resolved and the automaton will be in state q_1 . If instead, the character following the first a is another a , then since $q_2 \in \delta(q_1, a)$ and $q_4 \in \delta(q_2, a)$, and on any input the automaton once in state q_4 remains in q_4 , the input string will be accepted because q_4 is an accepting state. A similar analysis can be made if a b is read when the automaton is in state q_1 .

Strictly speaking, a dfa is *not* just a special kind of ndfa, although it is frequently thought of as such. This is because for a dfa, $\delta(q, s)$ is a state, whereas for an ndfa it is a set of states. But it is natural to identify the dfa \mathcal{M} with transition function δ , with the closely related ndfa $\tilde{\mathcal{M}}$ whose transition function $\bar{\delta}$ is given by

$$\bar{\delta}(q, s) = \{\delta(q, s)\},$$

and which has the same final states as \mathcal{M} . Obviously $L(\mathcal{M}) = L(\tilde{\mathcal{M}})$.

The main theorem on nondeterministic finite automata is

Theorem 2.1. A language is accepted by an ndfa if and only if it is regular. Equivalently, a language is accepted by an ndfa if and only if it is accepted by a dfa.

Proof. As we have just seen, a language accepted by a dfa is also accepted by an ndfa. Conversely, let $L = L(\mathcal{M})$, where \mathcal{M} is an ndfa with transition function δ , set of states $Q = \{q_1, \dots, q_m\}$, and set of final states F . We will construct a dfa $\tilde{\mathcal{M}}$ such that $L(\tilde{\mathcal{M}}) = L(\mathcal{M}) = L$. The idea of the construction is that the individual states of $\tilde{\mathcal{M}}$ will be sets of states of \mathcal{M} .

Thus, we proceed to specify the dfa $\tilde{\mathcal{M}}$ on the same alphabet as \mathcal{M} . The states of $\tilde{\mathcal{M}}$ are just the 2^m sets of states (including \emptyset) of \mathcal{M} . We write these as $\tilde{Q} = \{\tilde{Q}_1, \tilde{Q}_2, \dots, \tilde{Q}_{2^m}\}$, where in particular $\tilde{Q}_1 = \{q_1\}$ is to be the initial state of $\tilde{\mathcal{M}}$. The set \tilde{F} of final states of $\tilde{\mathcal{M}}$ is given by

$$\tilde{F} = \{\tilde{Q}_i \mid Q_i \cap F \neq \emptyset\}.$$

The transition function $\tilde{\delta}$ of $\tilde{\mathcal{M}}$ is then defined by

$$\tilde{\delta}(\tilde{Q}_i, s) = \bigcup_{q \in Q_i} \delta(q, s).$$

Now, we have

Lemma 1. Let $R \subseteq \tilde{Q}$. Then

$$\tilde{\delta}\left(\bigcup_{Q_i \in R} Q_i, s\right) = \bigcup_{Q_i \in R} \tilde{\delta}(Q_i, s).$$

Proof. Let $\bigcup_{Q_i \in R} Q_i = Q$. Then by definition,

$$\begin{aligned} \tilde{\delta}(Q, s) &= \bigcup_{q \in Q} \delta(q, s) \\ &= \bigcup_{Q_i \in R} \bigcup_{q \in Q_i} \delta(q, s) \\ &= \bigcup_{Q_i \in R} \tilde{\delta}(Q_i, s). \end{aligned} \quad \blacksquare$$

Lemma 2. For any string u ,

$$\tilde{\delta}^*(Q_i, u) = \bigcup_{q \in Q_i} \delta^*(q, u).$$

Proof. The proof is by induction on $|u|$. If $|u| = 0$, then $u = 0$ and

$$\tilde{\delta}^*(Q_i, 0) = Q_i = \bigcup_{q \in Q_i} \{q\} = \bigcup_{q \in Q_i} \delta^*(q, 0).$$

If $|u| = l + 1$ and the result is known for $|u| = l$, we write $u = vs$, where $|v| = l$, and observe that, using Lemma 1 and the induction hypothesis,

$$\begin{aligned} \tilde{\delta}^*(Q_i, u) &= \tilde{\delta}^*(Q_i, vs) \\ &= \tilde{\delta}\left(\tilde{\delta}^*(Q_i, v), s\right) \\ &= \tilde{\delta}\left(\bigcup_{q \in Q_i} \delta^*(q, v), s\right) \\ &= \bigcup_{q \in Q_i} \tilde{\delta}(\delta^*(q, v), s) \\ &= \bigcup_{q \in Q_i} \bigcup_{r \in \delta^*(q, v)} \delta(r, s) \\ &= \bigcup_{q \in Q_i} \delta^*(q, vs) \\ &= \bigcup_{q \in Q_i} \delta^*(q, u). \end{aligned} \quad \blacksquare$$

Lemma 3. $L(\mathcal{M}) = L(\tilde{\mathcal{M}})$.

Proof. $u \in L(\tilde{\mathcal{M}})$ if and only if $\tilde{\delta}^*(Q_1, u) \in \mathcal{F}$. But, by Lemma 2,

$$\tilde{\delta}^*(Q_1, u) = \tilde{\delta}^*(\{q_1\}, u) = \delta^*(q_1, u).$$

Hence,

$$\begin{aligned} u \in L(\tilde{\mathcal{M}}) &\quad \text{if and only if } \delta^*(q_1, u) \in \mathcal{F} \\ &\quad \text{if and only if } \delta^*(q_1, u) \cap F \neq \emptyset \\ &\quad \text{if and only if } u \in L(\mathcal{M}). \end{aligned}$$
■

Proof of Theorem 2.1 Concluded. Theorem 2.1 is an immediate consequence of Lemma 3. ■

Note that this proof is constructive. Not only have we shown that if a language is accepted by some ndfa, it is also accepted by some dfa, but we have also provided, within the proof, an algorithm for carrying out the conversion. This is important because, although it is frequently easier to design an ndfa than a dfa to accept a particular language, actual machines that are built are deterministic.

Exercises

1. Describe the language accepted by each of the following ndfas. In each case the initial state is q_1 .

(a)

δ_1	a	b	c	
q_1	$\{q_1, q_2, q_3\}$	\emptyset	\emptyset	
q_2	\emptyset	$\{q_4\}$	\emptyset	$F_1 = \{q_4\}$
q_3	\emptyset	\emptyset	$\{q_4\}$	
q_4	\emptyset	\emptyset	\emptyset	

(b) $\delta_2 = \delta_1$, $F_2 = \{q_1, q_2, q_3\}$.

(c)

δ_3	a	b	
q_1	$\{q_2\}$	\emptyset	
q_2	\emptyset	$\{q_1, q_3\}$	$F_3 = \{q_2\}$
q_3	$\{q_1, q_3\}$	\emptyset	

2. For each dfa \mathcal{M} in Exercise 1.3, transform \mathcal{M} into an ndfa \mathcal{M}' which accepts $L(\mathcal{M})$. Then transform \mathcal{M}' into a dfa \mathcal{M}'' by way of the construction in the proof of Theorem 2.1.
3. Let \mathcal{M} be a dfa with a single accepting state. Consider the ndfa \mathcal{M}' formed by reversing the roles of the initial and accepting states and reversing the direction of the arrows of all transitions in the transition diagram. Describe $L(\mathcal{M}')$ in terms of $L(\mathcal{M})$.
4. Prove that, given any ndfa \mathcal{M}_1 , there exists an ndfa \mathcal{M}_2 with exactly one accepting state such that

$$L(\mathcal{M}_2) = L(\mathcal{M}_1) \quad \text{or} \quad L(\mathcal{M}_2) = L(\mathcal{M}_1) - \{0\}.$$

5. (a) The construction in the proof of Theorem 2.1 shows that any regular language accepted by an ndfa with n states is accepted by some dfa with 2^n states. Show that there is a regular language that is accepted by an ndfa with two states, not accepted by any ndfa with fewer than two states, and accepted by a dfa with two states.
- (b) Show that there is a regular language that is accepted by an ndfa with two states and not accepted by any dfa with fewer than four states.
- (c) Show that there is a regular language that is accepted by an ndfa with three states and not accepted by any dfa with fewer than eight states.

3. Additional Examples

We first give two simple examples of finite automata and their associated regular languages.

For our first example we consider a unary even parity checker. That is, we want to design a finite automaton over the alphabet $\{1\}$ such that the machine terminates in an accepting state if and only if the input string contains an even number of ones. Intuitively then, the machine must contain two states which “remember” whether an even or an odd number of ones have been encountered so far. When the automaton begins, no ones, and hence an even number of ones, have been read; hence the initial state q_1 will represent the even parity state, and q_2 , the odd parity state. Furthermore, since we want to accept words containing an even number of ones, q_1 will be an accepting state.

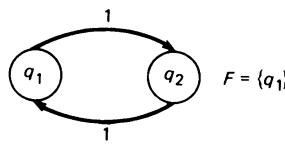


Figure 3.1

Thus the finite automaton to perform the required task is as shown in Fig. 3.1, and the language it accepts is

$$\{(11)^{[n]} \mid n \geq 0\}.$$

We next consider a slightly more complicated example. Suppose we wish to design a finite automaton that will function as a 25¢ candy vending machine. The alphabet consists of the three symbols n , d , and q (representing nickel, dime, and quarter, respectively—no pennies, please!). If more than 25¢ is deposited, no change is returned and no credit is given for the overage. Intuitively, the states keep track of the amount of money deposited so far. The automaton is exhibited in Fig. 3.2, with each state labeled to indicate its role. The state labeled 0 is the initial state. Note that the state labeled d is a “dead” state; i.e., once that state is entered it may never be left. Whenever sufficient money has been inserted so that the automaton has entered the 25¢ (accepting) state, any additional coins will send the machine into this dead state, which may be thought of as a coin return state. Presumably when in the accepting state, a button can be pressed to select your candy and the machine is reset to 0.

Unlike the previous example, the language accepted by this finite automaton is a finite set. It consists of the following combinations of nickels, dimes, and quarters: $\{nnnnn, nnnnd, nnnnq, nnnd, nnnq, nndn, nndd, nndq, nnq, ndnn, ndnd, ndnq, ndd, ndq, nq, dnnn, dnnd, dnnq, dnd, dnq, ddn, ddd, ddq, dq, q\}$.

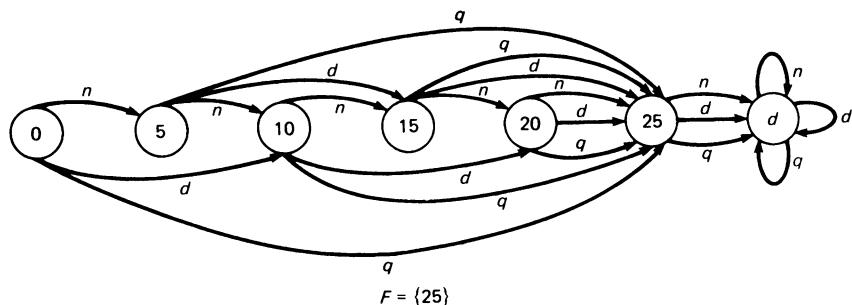


Figure 3.2

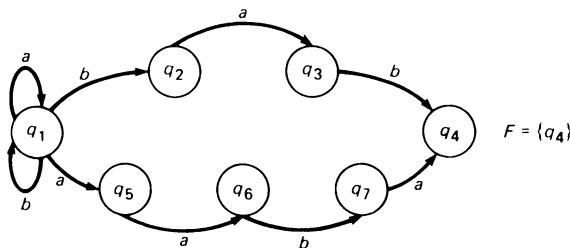


Figure 3.3

Suppose we wish to design an automaton on the alphabet $\{a, b\}$ that accepts all and only strings which end in bab or $aaba$. A real-world analog of this problem might arise in a demographic study in which people of certain ethnic groups are to be identified by checking to see if their family name ends in certain strings of letters.

It is easy to design the desired ndfa: see Fig. 3.3.

As our final example, we discuss a slightly more complicated version of the first example considered in Section 1:

$$L = \{a^{[n_1]}b^{[m_1]} \dots a^{[n_k]}b^{[m_k]} \mid n_1, m_1, \dots, n_k, m_k > 0\}.$$

An ndfa \mathcal{M} such that $L(\mathcal{M}) = L$ is shown in Fig. 3.4.

These two examples of ndfas illustrate an important characteristic of such machines: not only is it permissible to have many alternative transitions for a given state–symbol pair, but frequently there are no transitions for a given pair. In a sense, this means that whereas for a dfa one has to describe what happens for *any* string whether or not that string is a word in the language, for an ndfa one need only describe the behavior of the automaton for words in the language.

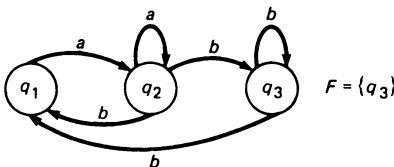


Figure 3.4

4. Closure Properties

We will be able to prove that the class of regular languages is closed under a large number of operations. It will be helpful that, by the equivalence

theorems of the previous two sections, we can use deterministic or non-deterministic finite automata to suit our convenience.

Definition. A dfa is called *nonrestarting* if there is no pair q, s for which

$$\delta(q, s) = q_1,$$

where q_1 is the initial state.

Theorem 4.1. There is an algorithm that will transform a given dfa \mathcal{M} into a nonrestarting dfa $\tilde{\mathcal{M}}$ such that $L(\tilde{\mathcal{M}}) = L(\mathcal{M})$.

Proof. Let $Q = \{q_1, q_2, \dots, q_n\}$ be the set of states of \mathcal{M} , q_1 the initial state, F the set of accepting states, and δ the transition function. We construct $\tilde{\mathcal{M}}$ with the set of states $\tilde{Q} = Q \cup \{q_{n+1}\}$, initial state q_1 , and transition function $\tilde{\delta}$ defined by

$$\tilde{\delta}(q, s) = \begin{cases} \delta(q, s) & \text{if } q \in Q \text{ and } \delta(q, s) \neq q_1 \\ q_{n+1} & \text{if } q \in Q \text{ and } \delta(q, s) = q_1, \end{cases}$$

$$\tilde{\delta}(q_{n+1}, s) = \tilde{\delta}(q_1, s).$$

Thus, there is no transition into state q_1 for $\tilde{\mathcal{M}}$. The set of accepting states \tilde{F} of $\tilde{\mathcal{M}}$ is defined by

$$\tilde{F} = \begin{cases} F & \text{if } q_1 \notin F \\ F \cup \{q_{n+1}\} & \text{if } q_1 \in F. \end{cases}$$

To see that $L(\mathcal{M}) = L(\tilde{\mathcal{M}})$ as required, one need only observe that $\tilde{\mathcal{M}}$ follows the same transitions as \mathcal{M} except that whenever \mathcal{M} reenters q_1 , $\tilde{\mathcal{M}}$ enters q_{n+1} . ■

Theorem 4.2. If L and \tilde{L} are regular languages, then so is $L \cup \tilde{L}$.

Proof. Without loss of generality, by Theorem 4.1, let $\mathcal{M}, \tilde{\mathcal{M}}$ be non-restarting dfas that accept L and \tilde{L} , respectively, with Q, q_1, F, δ and $\tilde{Q}, \tilde{q}_1, \tilde{F}, \tilde{\delta}$ the set of states, initial state, set of accepting states, and transition function of \mathcal{M} and $\tilde{\mathcal{M}}$, respectively. We also assume that \mathcal{M} and $\tilde{\mathcal{M}}$ have no states in common, i.e., $Q \cap \tilde{Q} = \emptyset$. Furthermore, by the discussion in Section 1, we can assume that the alphabets of L and \tilde{L} are the same, say, A . We define the *ndfa* $\check{\mathcal{M}}$ with states \check{Q} , initial state \check{q}_1 , set of accepting states \check{F} , and transition function $\check{\delta}$ as follows:

$$\check{Q} = Q \cup \tilde{Q} \cup \{\check{q}_1\} - \{q_1, \tilde{q}_1\}.$$

(That is, $\check{\mathcal{M}}$ contains a new initial state \check{q}_1 and all states of \mathcal{M} and $\tilde{\mathcal{M}}$ except their initial states.)

$$\check{F} = \begin{cases} F \cup \tilde{F} \cup \{\check{q}_1\} - \{q_1, \tilde{q}_1\} & \text{if } q_1 \in F \text{ or } \tilde{q}_1 \in \tilde{F} \\ F \cup \tilde{F} & \text{otherwise.} \end{cases}$$

The transition function of $\check{\mathcal{M}}$ is defined as follows for $s \in A$:

$$\check{\delta}(q, s) = \begin{cases} \{\delta(q, s)\} & \text{if } q \in Q - \{q_1\} \\ \{\tilde{\delta}(q, s)\} & \text{if } q \in \tilde{Q} - \{\tilde{q}_1\} \end{cases}$$

$$\check{\delta}(\check{q}_1, s) = \{\delta(q_1, s)\} \cup \{\tilde{\delta}(\tilde{q}_1, s)\}.$$

Thus, since $Q \cap \tilde{Q} = \emptyset$ and \mathcal{M} and $\tilde{\mathcal{M}}$ are nonrestarting, once a first transition has been selected, the automaton $\check{\mathcal{M}}$ is locked into one of the two automata \mathcal{M} and $\tilde{\mathcal{M}}$. Hence $L(\check{\mathcal{M}}) = L \cup \tilde{L}$. ■

Theorem 4.3. Let $L \subseteq A^*$ be a regular language. Then $A^* - L$ is regular.

Proof. Let \mathcal{M} be a dfa that accepts L . Let \mathcal{M} have alphabet A , set of states Q , and set of accepting states F . Let $\bar{\mathcal{M}}$ be exactly like \mathcal{M} except that it accepts precisely when \mathcal{M} rejects. That is, the set of accepting states of $\bar{\mathcal{M}}$ is $Q - F$. Then $\bar{\mathcal{M}}$ clearly accepts $A^* - L$. ■

Theorem 4.4. If L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$.

Proof. Let $L_1, L_2 \subseteq A^*$. Then we have the De Morgan identity:

$$L_1 \cap L_2 = A^* - ((A^* - L_1) \cup (A^* - L_2)).$$

Theorems 4.2 and 4.3 then give the result. ■

Theorem 4.5. \emptyset and $\{0\}$ are regular languages.

Proof. \emptyset is clearly the language accepted by any automaton whose set of accepting states is empty. Next, the automaton with states q_1, q_2 , alphabet $\{a\}$, accepting states $F = \{q_1\}$, and transition function $\delta(q_1, a) = \delta(q_2, a) = q_2$ clearly accepts $\{0\}$, as does any nonrestarting dfa on any alphabet provided $F = \{q_1\}$. ■

Theorem 4.6. Let $u \in A^*$. Then $\{u\}$ is a regular language.

Proof. For $u = 0$, we already know this from Theorem 4.5. Otherwise let $u = a_1a_2 \cdots a_la_{l+1}$, where $a_1, a_2, \dots, a_l, a_{l+1} \in A$. Let \mathcal{M} be the ndfa

with states q_1, q_2, \dots, q_{l+2} , initial state q_1 , accepting state q_{l+2} , and transition function δ given by

$$\begin{aligned}\delta(q_i, a_i) &= \{q_{i+1}\}, & i = 1, \dots, l + 1, \\ \delta(q_i, a) &= \emptyset & \text{for } a \in A - \{a_i\}.\end{aligned}$$

Then $L(\mathcal{M}) = \{u\}$. ■

Corollary 4.7. Every finite subset of A^* is regular.

Proof. We have already seen that \emptyset is regular. If $L = \{u_1, \dots, u_n\}$, where $u_1, \dots, u_n \in A^*$, we note that

$$L = \{u_1\} \cup \{u_2\} \cup \dots \cup \{u_n\},$$

and apply Theorems 4.2 and 4.6. ■

Exercises

1. Let $A = \{a, b\}$, let $L_1 \subseteq A^*$ consist of all words with at least two occurrences of a , and let $L_2 \subseteq A^*$ consist of all words with at least two occurrences of b . For each of the following languages L , give an ndfa that accepts L .
 - (a) $L = L_1 \cup L_2$.
 - (b) $L = A^* - L_1$.
 - (c) $L = A^* - L_2$.
 - (d) $L = L_1 \cap L_2$.
2. Use the constructions in the proofs of Theorem 4.6 and Corollary 4.7 to give an ndfa that accepts the language $\{ab, ac, ad\}$.
3. (a) Let L, L' be regular languages. Prove that $L - L'$ is regular.
 (b) Let L, L' be languages such that L is regular, $L \cup L'$ is regular, and $L \cap L' = \emptyset$. Prove that L' is regular.
4. Let L_1, L_2 be regular languages with $\text{rank}(L_1) = n_1$ and $\text{rank}(L_2) = n_2$. [See Exercise 1.6 for the definition of *rank*.]
 - (a) Use Theorems 4.1, 4.2, and 2.1 to give an upper bound on $\text{rank}(L_1 \cup L_2)$.
 - (b) Use Theorems 4.1, 4.2, 4.3, 4.4, and 2.1 to give an upper bound on $\text{rank}(L_1 \cap L_2)$.
- 5.* Let A_1, A_2 be alphabets, and let f be a function from A_1^* to subsets of A_2^* . f is a *substitution* on A_1 if $f(0) = \{0\}$ and, for all nonnull words

$a_1 \cdots a_n \in A_1^*$, where $a_1, \dots, a_n \in A_1$, $f(a_1 \cdots a_n) = f(a_1) \cdots f(a_n) = \{u_1, \dots, u_n \mid u_i \in f(a_i), 1 \leq i \leq n\}$. For $L \subseteq A_1^*$, $f(L) = \bigcup_{w \in L} f(w)$.

- (a) Let $A_1 = \{a, b\}$, $A_2 = \{c, d, e\}$, let f be the substitution on A_1 such that $f(a) = \{cc, 0\}$ and $f(b) = \{w \in A_2^* \mid w \text{ ends in } e\}$, and let $L = \{a^{[m]}b^{[n]} \mid m, n \geq 0\}$. What is $f(L)$?
- (b) Let A_1, A_2 be alphabets, let f be a substitution on A_1 such that $f(a) \subseteq A_2^*$ is a regular language for all $a \in A_1$, and let L be a regular language on A_1 . Prove that $f(L)$ is a regular language on A_2 .
- (c) Let A_1, A_2 be alphabets, and let g be a function from A_1^* to A_2^* . g is a *homomorphism* on A_1 if $g(0) = 0$ and, for all nonnull words $a_1 \cdots a_n \in A_1^*$, where $a_1, \dots, a_n \in A_1$, $g(a_1 \cdots a_n) = g(a_1) \cdots g(a_n)$. For $L \subseteq A_1^*$, $g(L) = \{g(w) \mid w \in L\}$. Use (b) to show that if g is a homomorphism on A_1 and $L \subseteq A_1^*$ is regular, then $g(L)$ is regular.

5. Kleene's Theorem

In this section we will see how the class of regular languages can be characterized as the class of all languages obtained from finite languages using a few operations.

Definition. Let $L_1, L_2 \subseteq A^*$. Then, we write

$$L_1 \cdot L_2 = L_1 L_2 = \{uv \mid u \in L_1 \text{ and } v \in L_2\}.$$

Definition. Let $L \subseteq A^*$. Then we write

$$L^* = \{u_1 u_2 \cdots u_n \mid n \geq 0, u_1, \dots, u_n \in L\}.$$

With respect to this last definition, note that

1. $0 \in L^*$ automatically because $n = 0$ is allowed;
2. for A^* the present notation is consistent with what we have been using.

Theorem 5.1. If L, \tilde{L} are regular languages, then $L \cdot \tilde{L}$ is a regular language.

Proof. Let \mathcal{M} and $\tilde{\mathcal{M}}$ be dfas that accept L and \tilde{L} , respectively, with Q, q_1, F, δ and $\tilde{Q}, \tilde{q}_1, \tilde{F}, \tilde{\delta}$ the set of states, initial state, set of accepting states, and transition function, respectively. Assume that \mathcal{M} and $\tilde{\mathcal{M}}$ have

no states in common, i.e., $Q \cap \tilde{Q} = \emptyset$. By our discussion in Section 1, we can assume without loss of generality that the alphabets of L and \tilde{L} are the same. Consider the ndfa $\dot{\mathcal{M}}$ formed by “gluing together” \mathcal{M} and $\tilde{\mathcal{M}}$ in the following way. The set \dot{Q} of states of $\dot{\mathcal{M}}$ is $Q \cup \tilde{Q}$, and the initial state is q_1 . We will define the transition function $\dot{\delta}$ of $\dot{\mathcal{M}}$ in such a way that the transitions of $\dot{\mathcal{M}}$ will contain all transitions of \mathcal{M} and $\tilde{\mathcal{M}}$. In addition $\dot{\delta}(q, s)$ will contain $\tilde{\delta}(\tilde{q}_1, s)$ for every $q \in F$. Thus, any time a symbol of the input string causes \mathcal{M} to enter an accepting state, $\dot{\mathcal{M}}$ can either continue by treating the next symbol of the input as being from the word of L or as the first symbol of the word of \tilde{L} . Formally we define $\dot{\delta}$ as follows:

$$\dot{\delta}(q, s) = \begin{cases} \{\delta(q, s)\} & \text{for } q \in Q - F \\ \{\delta(q, s)\} \cup \{\tilde{\delta}(\tilde{q}_1, s)\} & \text{for } q \in F \\ \{\tilde{\delta}(q, s)\} & \text{for } q \in \tilde{Q}. \end{cases}$$

Thus, $\dot{\mathcal{M}}$ begins by behaving exactly like \mathcal{M} . However, just when \mathcal{M} has accepted a word and would make a transition from an accepting state, $\dot{\mathcal{M}}$ may proceed as if it were $\tilde{\mathcal{M}}$ making a transition from \tilde{q}_1 .

Finally, if $0 \in \tilde{L}$ we set $\dot{F} = F \cup \tilde{F}$, and if $0 \notin \tilde{L}$ we set $\dot{F} = \tilde{F}$. Clearly, $L \cdot \tilde{L} = L(\dot{\mathcal{M}})$, so that $L \cdot \tilde{L}$ is a regular language. ■

Theorem 5.2. If L is a regular language, then so is L^* .

Proof. Let \mathcal{M} be a nonrestarting dfa that accepts L with alphabet A , set of states Q , initial state q_1 , accepting states F , and transition function δ . We construct the ndfa $\dot{\mathcal{M}}$ with the same states and initial state as \mathcal{M} , and accepting state q_1 . The transition function $\bar{\delta}$ is defined as follows:

$$\bar{\delta}(q, s) = \begin{cases} \{\delta(q, s)\} & \text{if } \delta(q, s) \notin F \\ \{\delta(q, s)\} \cup \{q_1\} & \text{if } \delta(q, s) \in F. \end{cases}$$

That is, whenever \mathcal{M} would enter an accepting state, $\dot{\mathcal{M}}$ will enter either the corresponding accepting state or the initial state. Clearly $L^* = L(\dot{\mathcal{M}})$, so that L^* is a regular language. ■

Theorem 5.3 (Kleene’s Theorem). A language is regular if and only if it can be obtained from finite languages by applying the three operators \cup , \cdot , $*$ a finite number of times.

The characterization of regular languages that Kleene’s theorem gives resembles the definition of the primitive recursive functions and the characterization of the partially computable functions of Theorem 3.5 in

Chapter 4. In each case one begins with some initial objects and applies certain operations a finite number of times.

Proof. Every finite language is regular by Corollary 4.7, and if $L = L_1 \cup L_2$ or $L = L_1 \cdot L_2$ or $L = L_1^*$, where L_1 and L_2 are regular, then L is regular by Theorems 4.2, 5.1, and 5.2, respectively. Therefore, by induction on the number of applications of \cup , \cdot , and $*$, any language obtained from finite languages by applying these operators a finite number of times is regular.

On the other hand, let L be a regular language, $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with states q_1, \dots, q_n . As usual, q_1 is the initial state, F is the set of accepting states, δ is the transition function, and $A = \{s_1, \dots, s_K\}$ is the alphabet. We define the sets $R_{i,j}^k$, $i, j > 0, k \geq 0$, as follows:

$$R_{i,j}^k = \{x \in A^* \mid \delta^*(q_i, x) = q_j \text{ and } \mathcal{M} \text{ passes through no state } q_l \text{ with } l > k \text{ as it moves across } x\}.$$

More formally, $R_{i,j}^k$ is the set of words $x = s_{i_1}s_{i_2} \cdots s_{i_r}s_{i_{r+1}}$ such that we can write

$$\delta(q_i, s_{i_1}) = q_{j_1},$$

$$\delta(q_{j_1}, s_{i_2}) = q_{j_2},$$

$$\vdots$$

$$\delta(q_{j_{r-1}}, s_{i_r}) = q_{j_r},$$

$$\delta(q_{j_r}, s_{i_{r+1}}) = q_j,$$

where $j_1, j_2, \dots, j_r \leq k$. Now, we observe that

$$R_{i,j}^0 = \{a \in A \mid \delta(q_i, a) = q_j\},$$

since for a word of length 1, \mathcal{M} passes directly from state q_i into state q_j while in processing any word of length > 1 , \mathcal{M} will pass through some intermediate state q_l , $l \geq 1$. Thus $R_{i,j}^0$ is a finite set. Furthermore, we have

$$R_{i,j}^{k+1} = R_{i,j}^k \cup [R_{i,k+1}^k \cdot (R_{k+1,k+1}^k)^* \cdot R_{k+1,j}^k]. \quad (5.1)$$

This rather imposing formula really states something quite simple: The set $R_{i,j}^{k+1}$ contains all the elements of $R_{i,j}^k$ and in addition contains strings x , such that \mathcal{M} in scanning x passes through the state q_{k+1} (but through none with larger subscript) some finite number of times. Such a string can be decomposed into a left end, which \mathcal{M} enters in state q_i and leaves in

state q_{k+1} (passing only through states with subscripts less than $k+1$ in the process), followed by some finite number of pieces each of which \mathcal{M} enters and leaves in state q_{k+1} (passing only through q_l with $l \leq k$), and a right end which \mathcal{M} enters in state q_{k+1} and leaves in state q_j (again passing only through states with subscript $\leq k$ in between). Now we have

Lemma. Each $R_{i,j}^k$ can be obtained from finite languages by a finite number of applications of the operations $\cup, \cdot, ^*$.

Proof. We prove by induction on k that for all i, j , the set $R_{i,j}^k$ has the desired property. For $k = 0$ this is obvious, since $R_{i,j}^0$ is finite.

Assuming the result known for k , (5.1) yields the result for $k + 1$. ■

Proof of Kleene's Theorem Concluded. We note that

$$L(\mathcal{M}) = \bigcup_{q_j \in F} R_{1,j}^n;$$

thus, the result follows at once from the lemma. ■

Kleene's theorem makes it possible to give names to regular languages in a particularly simple way. Let us begin with an alphabet $A = \{s_1, s_2, \dots, s_k\}$. Then we define the corresponding alphabet:

$$\tilde{A} = \{s_1, s_2, \dots, s_k, \mathbf{0}, \emptyset, \cup, \cdot, ^*, (,)\}.$$

The class of *regular expressions* on the alphabet A is then defined to be the subset of \tilde{A}^* determined by the following:

1. $\emptyset, \mathbf{0}, s_1, \dots, s_k$ are regular expressions.
2. If α and β are regular expressions, then so is $(\alpha \cup \beta)$.
3. If α and β are regular expressions, then so is $(\alpha \cdot \beta)$.
4. If α is a regular expression, then so is α^* .
5. No expression is regular unless it can be generated using a finite number of applications of 1–4.

Here are a few examples of regular expressions on the alphabet $A = \{a, b, c\}$:

$$\begin{aligned} & (a \cdot (b^* \cup c^*)) \\ & (\mathbf{0} \cup (a \cdot b)^*) \\ & (c^* \cdot b^*). \end{aligned}$$

For each regular expression γ , we define a corresponding regular language $\langle \gamma \rangle$ by recursion according to the following “semantic” rules:³

$$\begin{aligned}\langle s_i \rangle &= \{s_i\}, \\ \langle \mathbf{0} \rangle &= \{0\}, \\ \langle \emptyset \rangle &= \emptyset, \\ \langle (\alpha \cup \beta) \rangle &= \langle \alpha \rangle \cup \langle \beta \rangle, \\ \langle (\alpha \cdot \beta) \rangle &= \langle \alpha \rangle \cdot \langle \beta \rangle, \\ \langle \alpha^* \rangle &= \langle \alpha \rangle^*.\end{aligned}$$

When $\langle \gamma \rangle = L$, we say that the regular expression γ represents L . Thus,

$$\begin{aligned}\langle (a \cdot (b^* \cup c^*)) \rangle &= \{ab^{[n]} \mid n \geq 0\} \cup \{ac^{[m]} \mid m \geq 0\}, \\ \langle ((\mathbf{0} \cup (a \cdot b))^*) \rangle &= \langle (a \cdot b)^* \rangle = \{(ab)^{[n]} \mid n \geq 0\}, \\ \langle (c^* \cdot b^*) \rangle &= \{c^{[m]}b^{[n]} \mid m, n \geq 0\}.\end{aligned}$$

We have

Theorem 5.4. For every finite subset L of A^* , there is a regular expression γ on A such that $\langle \gamma \rangle = L$.

Proof. If $L = \emptyset$, then $L = \langle \emptyset \rangle$. If $L = \{0\}$, then $L = \langle \mathbf{0} \rangle$. If $L = \{x\}$, where $x = s_{i_1} s_{i_2} \cdots s_{i_l}$, then

$$L = \langle (s_{i_1} \cdot (s_{i_2} \cdot (s_{i_3} \cdots s_{i_l}) \cdots)) \rangle.$$

This gives the result for languages L consisting of 0 or 1 element. Assuming the result known for languages of k elements, let L have $k + 1$ elements. Then we can write

$$L = L_1 \cup \{x\},$$

where $x \in A^*$ and L_1 contains k elements. By the induction hypothesis, there is a regular expression α such that $\langle \alpha \rangle = L_1$. By the one-element case already considered, there is a regular expression β such that $\langle \beta \rangle = \{x\}$. Then we have

$$\langle (\alpha \cup \beta) \rangle = \langle \alpha \rangle \cup \langle \beta \rangle = L_1 \cup \{x\} = L. \quad \blacksquare$$

³ For more on this subject see Part 5.

Theorem 5.5 (Kleene's Theorem—Second Version). A language $L \subseteq A^*$ is regular if and only if there is a regular expression γ on A such that $\langle \gamma \rangle = L$.

Proof. For any regular expression γ , the regular language $\langle \gamma \rangle$ is built up from finite languages by applying \cup , \cdot , $*$ a finite number of times, so $\langle \gamma \rangle$ is regular by Kleene's theorem.

On the other hand, let L be a regular language. If L is finite then, by Theorem 5.4, there is a regular expression γ such that $\langle \gamma \rangle = L$. Otherwise, by Kleene's theorem, L can be obtained from certain finite language by a finite number of applications of the operations \cup , \cdot , $*$. By beginning with regular expressions representing these finite languages, we can build up a regular expression representing L by simply indicating each use of the operations \cup , \cdot , $*$ by writing \cup , \cdot , $*$, respectively, and punctuating with (and). ■

Exercises

1. (a) For each language L described in Exercise 1.1, give a regular expression α such that $L = \langle \alpha \rangle$.
 (b) For each dfa \mathcal{M} described in Exercise 1.3, give a regular expression α such that $L(\mathcal{M}) = \langle \alpha \rangle$.
 (c) For each ndfa \mathcal{M} described in Exercise 2.1, give a regular expression α such that $L(\mathcal{M}) = \langle \alpha \rangle$.
2. For regular expressions α, β , let us write $\alpha \equiv \beta$ to mean that $\langle \alpha \rangle = \langle \beta \rangle$. For α, β, γ given regular expressions, prove the following identities.
 - (a) $(\alpha \cup \alpha) \equiv \alpha$.
 - (b) $((\alpha \cdot \beta) \cup (\alpha \cdot \gamma)) \equiv (\alpha \cdot (\beta \cup \gamma))$.
 - (c) $((\beta \cdot \alpha) \cup (\gamma \cdot \alpha)) \equiv ((\beta \cup \gamma) \cdot \alpha)$.
 - (d) $(\alpha^* \cdot \alpha^*) \equiv \alpha^*$.
 - (e) $(\alpha \cdot \alpha^*) \equiv (\alpha^* \cdot \alpha)$.
 - (f) $\alpha^{**} \equiv \alpha^*$.
 - (g) $(\emptyset \cup (\alpha \cdot \alpha^*)) \equiv \alpha^*$.
 - (h) $((\alpha \cdot \beta)^* \cdot \alpha) \equiv (\alpha \cdot (\beta \cdot \alpha)^*)$.
 - (i) $(\alpha \cup \beta)^* \equiv (\alpha^* \cdot \beta^*)^* \equiv (\alpha^* \cup \beta^*)^*$.
3. Using the identities of Exercise 2 prove that

$$((abb)^*(ba)^*(b \cup aa)) \equiv (abb)^*((\emptyset \cup (b(ab)^*a))b \cup (ba)^*(aa)).$$

(Note that parentheses and the symbol “.” have been omitted to facilitate reading.)

4. Let α, β be given regular expressions such that $0 \notin \langle \alpha \rangle$. Consider the equation in the “unknown” regular expression ξ :

$$\xi \equiv (\beta \cup (\xi \cdot \alpha^*)).$$

Prove that this equation has the solution

$$\xi \equiv (\beta \cdot \alpha^*)$$

and that the solution is unique in the sense that if ξ' also satisfies the equation, then $\xi \equiv \xi'$.

5. Let $L = \{x \in \{a, b\}^* \mid x \neq 0 \text{ and } bb \text{ is not a substring of } x\}$.
- (a) Show that L is regular by constructing a dfa \mathcal{M} such that $L = L(\mathcal{M})$.
 - (b) Find a regular expression γ such that $L = \langle \gamma \rangle$.
6. Let $L = \langle ((a \cdot a) \cup (a \cdot a \cdot a))^* \rangle$. Find a dfa \mathcal{M} that accepts L .
7. Describe an algorithm that, given any regular expression α , produces an ndfa \mathcal{M} that accepts $\langle \alpha \rangle$.
8. Let L_1, L_2 be regular languages with $\text{rank}(L_1) = n_1$ and $\text{rank}(L_2) = n_2$. [See Exercise 1.6 for the definition of rank.]
- (a) Use Theorem 5.1 to give an upper bound on $\text{rank}(L_1 \cdot L_2)$.
 - (b) Use Theorem 5.2 to give an upper bound on $\text{rank}(L_1^*)$.
9. Let $A = \{s_1, \dots, s_n\}$.
- (a) Give a function b_1 such that $\text{rank}(\langle \alpha \rangle) \leq b_1(\alpha)$ for all regular expressions α on A .
 - (b) Define the *size* of a regular expression on A as follows.

$$\begin{aligned}\text{size}(\emptyset) &= 1 \\ \text{size}(0) &= 1 \\ \text{size}(s_i) &= 1 \quad i = 1, \dots, n \\ \text{size}((\alpha \cup \beta)) &= \text{size}(\alpha) + \text{size}(\beta) + 1 \\ \text{size}((\alpha \cdot \beta)) &= \text{size}(\alpha) + \text{size}(\beta) + 1 \\ \text{size}(\alpha^*) &= \text{size}(\alpha) + 1\end{aligned}$$

Give a numeric function b_2 such that $\text{rank}(\langle \alpha \rangle) \leq b_2(\text{size}(\alpha))$ for all regular expressions α on A .

(c)* Verify that b_2 is primitive recursive.

- 10.* Let $A = \{s_1, \dots, s_n\}$, let α, β be regular expressions on A , and let P_α, P_β be primitive recursive predicates such that for all $w \in A^*$,

$P_\alpha(w) = 1$ if and only if $w \in \langle \alpha \rangle$ and $P_\beta(w) = 1$ if and only if $w \in \langle \beta \rangle$.

- (a) Give a primitive recursive predicate $P_{(\alpha \cup \beta)}$ such that $P_{(\alpha \cup \beta)}(w) = 1$ if and only if $w \in \langle (\alpha \cup \beta) \rangle$.
- (b) Give a primitive recursive predicate $P_{(\alpha \cdot \beta)}$ such that $P_{(\alpha \cdot \beta)}(w) = 1$ if and only if $w \in \langle (\alpha \cdot \beta) \rangle$.
- (c) Give a primitive recursive predicate P_{α^*} such that $P_{\alpha^*}(w) = 1$ if and only if $w \in \langle \alpha^* \rangle$.
- (d) Use parts (a), (b), and (c) to show that for all regular expressions γ on A , there is a primitive recursive predicate P_γ such that $P_\gamma(w) = 1$ if and only if $w \in \langle \gamma \rangle$.

6. The Pumping Lemma and Its Applications

We will make use of the following basic combinatorial fact:

Pigeon-Hole Principle. If $(n + 1)$ objects are distributed among n sets, then at least one of the sets must contain at least two objects.

We will use this pigeon-hole principle to prove the following result.

Theorem 6.1 (Pumping Lemma). Let $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with n states. Let $x \in L$, where $|x| \geq n$. Then we can write $x = uvw$, where $v \neq 0$ and $uv^{[i]}w \in L$ for all $i = 0, 1, 2, 3, \dots$.

Proof. Since x consists of at least n symbols, \mathcal{M} must go through at least n state transitions as it scans x . Including the initial state, this requires at least $n + 1$ (not necessarily distinct) states. But since there are only n states in all, we conclude (here is the pigeon-hole principle!) that \mathcal{M} must be in at least one state more than once. Let q be a state in which \mathcal{M} finds itself at least twice. Then we can write $x = uvw$, where

$$\delta^*(q_1, u) = q,$$

$$\delta^*(q, v) = q,$$

$$\delta^*(q, w) \in F.$$

That is, \mathcal{M} arrives in state q for the first time after scanning the last (right-hand) symbol of u and then again after scanning the last symbol of v . Since this “loop” can be repeated any number of times, it is clear that

$$\delta^*(q_1, uv^{[i]}w) = \delta^*(q_1, uvw) \in F.$$

Hence $uv^{[i]}w \in L$. ■

Theorem 6.2. Let \mathcal{M} be a dfa with n states. Then, if $L(\mathcal{M}) \neq \emptyset$, there is a string $x \in L(\mathcal{M})$ such that $|x| < n$.

Proof. Let x be a string in $L(\mathcal{M})$ of the shortest possible length. Suppose $|x| \geq n$. By the pumping lemma, $x = uvw$, where $v \neq 0$ and $uw \in L(\mathcal{M})$. Since $|uw| < |x|$, this is a contradiction. Thus $|x| < n$. ■

This theorem furnishes an algorithm for testing a given dfa \mathcal{M} to see whether the language it accepts is empty. We need only “run” \mathcal{M} on all strings of length less than the number of states of \mathcal{M} . If none is accepted, we will be able to conclude that $L(\mathcal{M}) = \emptyset$.

Next we turn to infinite regular languages. If $L = L(\mathcal{M})$ is infinite, then L must surely contain words having length greater than the number of states of \mathcal{M} . Hence from the pumping lemma, we can conclude

Theorem 6.3. If L is an infinite regular language, then there are words u, v, w , such that $v \neq 0$ and $uv^{[i]}w \in L$ for $i = 0, 1, 2, 3, \dots$.

This theorem is useful in showing that certain languages are *not* regular. However, for infinite regular languages we can say even more.

Theorem 6.4. Let \mathcal{M} be a dfa with n states. Then $L(\mathcal{M})$ is infinite if and only if $L(\mathcal{M})$ contains a string x such that $n \leq |x| < 2n$.

Proof. First let $x \in L(\mathcal{M})$ with $n \leq |x| < 2n$. By the pumping lemma, we can write $x = uvw$, where $v \neq 0$ and $uv^{[i]}w \in L(\mathcal{M})$ for all i . But then $L(\mathcal{M})$ is infinite.

Conversely, let $L(\mathcal{M})$ be infinite. Then $L(\mathcal{M})$ must contain strings of length $\geq 2n$. Let $x \in L(\mathcal{M})$, where x has the shortest possible length $\geq 2n$. We write $x = x_1x_2$, where $|x_1| = n$. Thus $|x_2| \geq n$. Then using the pigeon-hole principle as in the proof of the pumping lemma, we can write $x_1 = uvw$, where

$$\begin{aligned}\delta^*(q_1, u) &= q, \\ \delta^*(q, v) &= q \quad \text{with} \quad 1 \leq |v| \leq n, \\ \delta^*(q, wx_2) &\in F.\end{aligned}$$

Thus $uwx_2 \in L(\mathcal{M})$. But

$$|uwx_2| \geq |x_2| \geq n,$$

and $|uwx_2| < |x|$, and since x was a shortest word of $L(\mathcal{M})$ with length at least $2n$, we have

$$n \leq |uwx_2| < 2n.$$

This theorem furnishes an algorithm for testing a given dfa \mathcal{M} to determine whether $L(\mathcal{M})$ is finite. We need only run \mathcal{M} on all strings x such that $n \leq |x| < 2n$, where \mathcal{M} has n states. $L(\mathcal{M})$ is infinite just in case \mathcal{M} accepts at least one of these strings.

For another example of an algorithm, let $\mathcal{M}_1, \mathcal{M}_2$ be dfas on the alphabet A and let us seek to determine whether $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$. Using the methods of proof of Theorems 4.2–4.4, we can obtain a dfa \mathcal{M} such that

$$L(\mathcal{M}) = L(\mathcal{M}_1) \cap [A^* - L(\mathcal{M}_2)].$$

Then $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$ if and only if $L(\mathcal{M}) = \emptyset$. Since Theorem 6.2 enables us to test algorithmically whether $L(\mathcal{M}) = \emptyset$, we have an algorithm by means of which we can determine whether $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$. Moreover, since $L(\mathcal{M}_1) = L(\mathcal{M}_2)$ just when $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$ and $L(\mathcal{M}_2) \subseteq L(\mathcal{M}_1)$, we also have an algorithm for testing whether $L(\mathcal{M}_1) = L(\mathcal{M}_2)$.

The pumping lemma also furnishes a technique for showing that given languages are not regular. For example, let $L = \{a^{[n]}b^{[n]} \mid n > 0\}$, and suppose that $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with m states. We get a contradiction by showing that there is a word $x \in L$, with $|x| \geq m$, such that there is no way of writing $x = uvw$, with $v \neq 0$, so that $\{uv^{[i]}w \mid i \geq 0\} \subseteq L$. Let $x = a^{[l]}b^{[l]}$, where $2l \geq m$, and let $a^{[l]}b^{[l]} = uwv$. Then either $v = a^{[l_1]}$ or $v = a^{[l_1]}b^{[l_2]}$ or $v = b^{[l_2]}$, with $l_1, l_2 \leq l$, and in each case $uvw \notin L$, contradicting the pumping lemma, so there can be no such dfa \mathcal{M} , and L is not regular.

This example and the exercises at the end of Section 7 show that finite automata are incapable of doing more than a limited amount of counting.

Exercises

- Given a word w and a dfa \mathcal{M} , a test to determine if $w \in L(\mathcal{M})$ is a *membership test*.
 - Let $\mathcal{M}_1, \mathcal{M}_2$ be arbitrary dfas on alphabet $A = \{s_1, \dots, s_n\}$, where \mathcal{M}_1 has m_1 states and \mathcal{M}_2 has m_2 states. Give an upper bound $f(m_1, m_2)$ on the number of membership tests necessary to determine if $L(\mathcal{M}_1) = L(\mathcal{M}_2)$.
 - * Verify that f is primitive recursive.
- (a) Describe an algorithm that, for any regular expressions α and β , determines if $\langle \alpha \rangle = \langle \beta \rangle$.
 - Give a function $g(x, y)$ such that the algorithm in part (a) requires at most $g(\text{size}(\alpha), \text{size}(\beta))$ membership tests. [See Exercise 5.9 for the definition of $\text{size}(\alpha)$.]
 - * Verify that g is primitive recursive.

7. The Myhill–Nerode Theorem

We conclude this chapter by giving another characterization of the regular languages on an alphabet A . We begin with a pair of definitions.

Definition. Let $L \subseteq A^*$, where A is an alphabet. For strings $x, y \in A^*$, we write $x \equiv_L y$ to mean that for every $w \in A^*$ we have $xw \in L$ if and only if $yw \in L$.

It is obvious that \equiv_L has the following properties.

$$x \equiv_L x.$$

$$\text{If } x \equiv_L y, \text{ then } y \equiv_L x.$$

$$\text{If } x \equiv_L y \text{ and } y \equiv_L z, \text{ then } x \equiv_L z.$$

(Relations having these three properties are known as *equivalence relations*.)

It is also obvious that

$$\text{If } x \equiv_L y, \text{ then for all } w \in A^*, xw \equiv_L yw.$$

Definition. Let $L \subseteq A^*$, where A is an alphabet. Let $S \subseteq A^*$. Then S is called a *spanning set for L* if

1. S is finite, and
2. for every $x \in A^*$, there is a $y \in S$ such that $x \equiv_L y$.

Then we can prove

Theorem 7.1 (Myhill–Nerode). A language is regular if and only if it has a spanning set.

Proof. First let L be regular. Then $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with set of states Q , initial state q_1 , and transition function δ . Let us call a state $q \in Q$ *reachable* if there exists $y \in A^*$ such that

$$\delta^*(q_1, y) = q. \quad (7.1)$$

For each reachable state q , we select one particular string y that satisfies (7.1) and we write it as y_q . Thus,

$$\delta^*(q_1, y_q) = q$$

for every reachable state q . We set

$$S = \{y_q \mid q \text{ is reachable}\}.$$

S is clearly finite. To show that S is a spanning set for L , we let $x \in A^*$ and show how to find $y \in S$ such that $x \equiv_L y$. In fact, let $\delta^*(q_1, x) = q$,

and set $y = y_q$. Thus, $y \in S$ and $\delta^*(q_1, y) = q$. Now for every $w \in A^*$,

$$\delta^*(q_1, xw) = \delta^*(q, w) = \delta^*(q_1, yw).$$

Hence, $\delta^*(q_1, xw) \in F$ if and only if $\delta^*(q_1, yw) \in F$; i.e., $xw \in L$ if and only if $yw \in L$. Thus, $x \equiv_L y$.

Conversely, let $L \subseteq A^*$ and let $S \subseteq A^*$ be a spanning set for L . We show how to construct a dfa \mathcal{M} such that $L(\mathcal{M}) = L$. We define the set of states of \mathcal{M} to be $Q = \{q_x \mid x \in S\}$, where we have associated a state q_x with each element $x \in S$. Since S is a spanning set for L , there is an $x_0 \in S$ such that $0 \equiv_L x_0$; we take q_{x_0} to be the initial state of \mathcal{M} . We let the final states of \mathcal{M} be

$$F = \{q_y \mid y \in L\}.$$

Finally, for $a \in A$, we set $\delta(q_x, a) = q_y$, where $y \in S$ and $xa \equiv_L y$. Then we claim that for all $w \in A^*$,

$$\delta^*(q_x, w) = q_y, \quad \text{where } xw \equiv_L y.$$

We prove this claim by induction on $|w|$. For $|w| = 0$, we have $w = 0$. Moreover, $\delta^*(q_x, 0) = q_x$ and $x0 = x \equiv_L x$. Suppose our claim is known for all words w such that $|w| = k$, and consider $w \in A^*$ with $|w| = k + 1$. Then $w = ua$, where $|u| = k$ and $a \in A$. We have

$$\delta^*(q_x, w) = \delta(\delta^*(q_x, u), a) = \delta(q_y, a) = q_z,$$

where, using the induction hypothesis, $xu \equiv_L y$ and, by definition of δ , $ya \equiv_L z$. Then $xw = xua \equiv_L ya \equiv_L z$, which proves the claim. Now, we have

$$L(\mathcal{M}) = \{w \in A^* \mid \delta^*(q_{x_0}, w) \in F\}.$$

Let $\delta^*(q_{x_0}, w) = q_y$. Then by the way x_0 was defined and our claim,

$$w \equiv_L x_0w \equiv_L y.$$

Thus, $w \in L$ if and only if $y \in L$, which in turn is true if and only if $q_y \in F$, i.e., if and only if $w \in L(\mathcal{M})$. Hence $L = L(\mathcal{M})$. ■

Like the pumping lemma, the Myhill–Nerode theorem furnishes a technique for showing that a given language is not regular. For example, let $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ again, and let n_1, n_2 be distinct numbers > 0 . Then $a^{[n_1]}b^{[n_1]} \in L$ and $a^{[n_2]}b^{[n_1]} \notin L$, so $a^{[n_1]} \not\equiv_L a^{[n_2]}$, and since \equiv_L is an equivalence relation, there can be no word w such that $a^{[n_1]} \equiv_L w$ and $a^{[n_2]} \equiv_L w$. But if there were a spanning set $S = \{w_1, \dots, w_m\}$ for L , then by the pigeon-hole principle, there would have to be at least two distinct

words among $\{a, aa, \dots, a^{[m+1]}\}$, say $a^{[i]}$ and $a^{[j]}$, and some $w_k \in S$ such that $a^{[i]} \equiv_L w_k$ and $a^{[j]} \equiv_L w_k$, which is impossible. Therefore L has no spanning set, and by the Myhill–Nerode theorem, L is not regular.

Exercises

1. (a) For each language L described in Exercise 1.1, give a spanning set for L .
 (b) For each dfa \mathcal{M} described in Exercise 1.3, give a spanning set for $L(\mathcal{M})$.
 (c) For each ndfa \mathcal{M} described in Exercise 2.1, give a spanning set for $L(\mathcal{M})$.
2. Prove that there is no dfa that accepts exactly the set of all words that are palindromes over a given alphabet containing at least two symbols. (For a definition of palindrome, see Exercise 1.1f.)
3. u is called an *initial segment* of a word w if there is a word v such that $w = uv$. Let L be a regular language. Prove that the language consisting of all initial segments of words of L is a regular language.
4. Let L be a regular language and L' the language consisting of all words w such that both w and $w \cdot w$ are words in L . Prove that L' is regular.
5. Prove the following statement, if it is true, or give a counterexample: Every language that is a subset of a regular language is regular.
6. Prove that each of the following is *not* a regular language.
 - (a) The language on the alphabet $\{a, b\}$ consisting of all strings in which the number of occurrences of b is greater than the number of occurrences of a .
 - (b) The language L over the alphabet $\{., 0, 1, \dots, 9\}$, consisting of all strings that are initial segments of the infinite decimal expansion of π . [$L = \{3, 3., 3.1, 3.14, 3.141, 3.1415, \dots\}$.]
 - (c) The language L over the alphabet $\{a, b\}$ consisting of all strings that are initial segments of the infinite string

$$bababaaaabaaaab\dots$$
7. Let $L = \{a^{[i]}b^{[j]} \mid i \neq j\}$. Show that L is not regular.
8. Let $L = \{a^{[n]}b^{[2n]} \mid n > 0\}$. Show that L is not regular.
9. Let $L = \{a^{[n]}b^{[m]} \mid 0 < n \leq m\}$. Show that L is not regular.
10. Let $L = \{a^{[p]} \mid p \text{ is a prime number}\}$. Show that L is not regular.

11. Let \mathcal{M} be a finite automaton with alphabet A , set of states $Q = \{q_1, \dots, q_n\}$, initial state q_1 , and transition function δ . Let a_1, a_2, a_3, \dots be an *infinite sequence* of symbols of A . We can think of these symbols as being “fed” to \mathcal{M} in the given order producing a sequence of states r_1, r_2, r_3, \dots , where r_1 is just the initial state q_1 and $r_{i+1} = \delta(r_i, a_i)$, $i = 1, 2, 3, \dots$. Suppose there are integers p, k such that

$$a_{i+p} = a_i \quad \text{for all } i \geq k.$$

Prove that there are integers l, s such that $s \leq np$ and

$$r_{i+s} = r_i \quad \text{for all } i \geq l.$$

[Hint: Use the pigeon-hole principle.]

12. (a) Let L be a regular language, and let S be a spanning set for L . S is a *minimal* spanning set for L if there is no spanning set for L that has fewer elements than S , and S is *independent* if there is no pair s, s' of distinct elements of S such that $s \equiv_L s'$. Prove that S is minimal if and only if it is independent.
- (b) Let L be a regular language, and let S, S' be spanning sets for L . S and S' are *isomorphic* if there is a one-one function f from S onto S' such that $s \equiv_L f(s)$ for all $s \in S$. Prove that if S and S' are both minimal, then they are isomorphic.
- (c) A dfa \mathcal{M} is a *minimal dfa* for a regular language L if $L = L(\mathcal{M})$ and if there is no dfa \mathcal{M}' with fewer states than \mathcal{M} such that $L(\mathcal{M}') = L(\mathcal{M})$. Let L be a regular language, let \mathcal{M} be a dfa that accepts L , and let S be a spanning set for L constructed from \mathcal{M} as in the proof of Theorem 7.1. Prove that if \mathcal{M} is a minimal dfa for L then S is a minimal spanning set for L . Why is the converse to this statement false?
- (d) Let L be a regular language, let S be a spanning set for L , and let \mathcal{M} be the dfa constructed from S as in the proof of Theorem 7.1. Prove that S is a minimal spanning set for L if and only if \mathcal{M} is a minimal dfa for L .
- (e) Let \mathcal{M} and \mathcal{M}' be dfas on alphabet A with states Q and Q' , initial states q_1 and q'_1 , accepting states F and F' , and transition functions δ and δ' . \mathcal{M} and \mathcal{M}' are *isomorphic* if there is a one-one function g from Q onto Q' such that $g(q_1) = q'_1$, $q \in F$ if and only if $g(q) \in F'$, and $\delta'(g(q), s) = g(\delta(q, s))$ for all $q \in Q$ and $s \in A$. (Informally, \mathcal{M} and \mathcal{M}' are identical but for a renaming of the states.) Prove that, if \mathcal{M} and \mathcal{M}' are both minimal dfas for some regular language L , then they are isomorphic.

13. Let L_1, L_2 be languages on some alphabet A . The *right quotient* of L_1 by L_2 , denoted L_1/L_2 , is $\{x \mid xy \in L_1 \text{ for some } y \in L_2\}$. Prove that if L_1 and L_2 are regular, then L_1/L_2 is regular.
14. Let $L = \{a^{[p]}b^{[m]} \mid p \text{ is a prime number, } m > 0\} \cup \{a^{[n]} \mid n \geq 0\}$.
 - (a) Show that L is not regular. [Hint: See Exercise 4.3 and Exercises 10 and 13 above.]
 - (b) Explain why the pumping lemma alone is not sufficient to show that L is not regular.
 - (c) State and prove a stronger version of the pumping lemma which is sufficient to show that L is not regular.

10

Context-Free Languages

1. Context-Free Grammars and Their Derivation Trees

Let \mathcal{V}, T be a pair of disjoint alphabets. A *context-free production* on \mathcal{V}, T is an expression

$$X \rightarrow h$$

where $X \in \mathcal{V}$ and $h \in (\mathcal{V} \cup T)^*$. The elements of \mathcal{V} are called *variables*, and the elements of T are called *terminals*. If P stands for the production $X \rightarrow h$ and $u, v \in (\mathcal{V} \cup T)^*$, we write

$$u \xrightarrow{P} v$$

to mean that there are words $p, q \in (\mathcal{V} \cup T)^*$ such that $u = pXq$ and $v = phq$. In other words, v results from u by replacing the variable X by the word h . Productions $X \rightarrow 0$ are called *null productions*. A *context-free grammar* Γ with variables \mathcal{V} and terminals T consists of a finite set of context-free productions on \mathcal{V}, T together with a designated symbol $S \in \mathcal{V}$ called the *start symbol*. Collectively, the set $\mathcal{V} \cup T$ is called the *alphabet* of Γ . If none of the productions of Γ is a null production, Γ is called a *positive context-free grammar*.¹

¹ Those who have read Chapter 7 should note that every positive context-free grammar is a context-sensitive grammar in the sense defined there. For the moment we are not assuming familiarity with Chapter 7. However, the threads will all be brought together in the next chapter.

If Γ is a context-free grammar with variables \mathcal{V} and terminals T , and if $u, v \in (\mathcal{V} \cup T)^*$, we write

$$u \xrightarrow{P} v$$

to mean that $u \xrightarrow{P} v$ for some production P of Γ . We write

$$u \xrightarrow{\Gamma} v$$

to mean there is a sequence u_1, \dots, u_m where $u = u_1, u_m = v$, and

$$u_i \xrightarrow{\Gamma} u_{i+1} \quad \text{for } 1 \leq i < m.$$

The sequence u_1, \dots, u_m is called a *derivation of v from u in Γ* . The number m is called the *length* of the derivation.² The symbol Γ below the \Rightarrow may be omitted when no ambiguity results. Finally, we define

$$L(\Gamma) = \{u \in T^* \mid S \xrightarrow{*} u\}.$$

$L(\Gamma)$ is called the language *generated* by Γ . A language $L \subseteq T^*$ is called *context-free* if there is a context-free grammar Γ such that $L = L(\Gamma)$.

A simple example of a context-free grammar Γ is given by $\mathcal{V} = \{S\}$, $T = \{a, b\}$, and the productions

$$S \rightarrow aSb, \quad S \rightarrow ab.$$

Here we clearly have

$$L(\Gamma) = \{a^{[n]}b^{[n]} \mid n > 0\};$$

thus, this language is context-free. We showed in Chapter 9, Section 6, that $L(\Gamma)$ is not regular. Later we shall see that every regular language is context-free. For the meanwhile we have proved

Theorem 1.1. The language $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ is context-free but not regular.

We now wish to discuss the relation between context-free grammars in general and positive context-free grammars. It is obvious that if Γ is a positive context-free grammar, then $0 \notin L(\Gamma)$. We shall show that except for this limitation, everything that can be done using context-free grammars can be done with positive context-free grammars. This will require some messy technicalities, but working out the details now will simplify matters later.

² Some authors use the number $m - 1$ as the length of the derivation.

Definition. We define the *kernel* of a given context-free grammar Γ , written $\ker(\Gamma)$, to be the set of variables V of Γ such that $V \xrightarrow{\Gamma}^* 0$.

As an example consider the context-free grammar Γ_0 with productions

$$S \rightarrow XYYX, \quad S \rightarrow aX, \quad X \rightarrow 0, \quad Y \rightarrow 0.$$

Then $\ker(\Gamma_0) = \{X, Y, S\}$. This example suggests an algorithm for locating the elements of $\ker(\Gamma)$ for a given context-free grammar Γ . We let

$$\mathcal{V}_0 = \{V \mid V \rightarrow 0 \text{ is a production of } \Gamma\},$$

$$\mathcal{V}_{i+1} = \mathcal{V}_i \cup \{V \mid V \rightarrow \alpha \text{ is a production of } \Gamma, \text{ where } \alpha \in \mathcal{V}_i^*\}.$$

Thus for Γ_0 , $\mathcal{V}_0 = \{X, Y\}$, $\mathcal{V}_1 = \{X, Y, S\}$, and $\mathcal{V}_i = \mathcal{V}_1$ for all $i > 1$. S is in \mathcal{V}_1 because $XYYX \in \mathcal{V}_0^*$. In the general case it is clear, because Γ has only finitely many variables, that a stage k will eventually be reached for which $\mathcal{V}_{k+1} = \mathcal{V}_k$ and that then $\mathcal{V}_i = \mathcal{V}_k$ for all $i > k$. We have

Lemma 1. If $\mathcal{V}_k = \mathcal{V}_{k+1}$, then $\ker(\Gamma) = \mathcal{V}_k$.

Proof. It is clear that $\mathcal{V}_i \subseteq \ker(\Gamma)$ for all i . Conversely, we show that if $V \in \ker(\Gamma)$, then $V \in \mathcal{V}_k$. We prove this by induction on the length of a derivation of 0 from V in Γ . If there is such a derivation of length 2, then $V \xrightarrow{\Gamma} 0$, so that $V \rightarrow 0$ is a production of Γ and $V \in \mathcal{V}_0$. Let us assume the result for all derivations of length $< r$ and let $V = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_{r-1} \Rightarrow \alpha_r = 0$ be a derivation of length r in Γ . The words $\alpha_1, \alpha_2, \dots, \alpha_{r-1}$ must consist entirely of variables, since terminals cannot be eliminated by context-free productions. Let $\alpha_2 = V_1 V_2 \dots V_s$. Then we have $V_i \xrightarrow{\Gamma} 0$, $i = 1, 2, \dots, s$, by derivations of length $< r$. By the induction hypothesis, each $V_i \in \mathcal{V}_k$. Since Γ contains the production $V \rightarrow V_1 V_2 \dots V_s$, and $\alpha_2 \in \mathcal{V}_k^*$, we have $V \in \mathcal{V}_{k+1} = \mathcal{V}_k$. ■

Lemma 2. There is an algorithm that will transform a given context-free grammar Γ into a positive context-free grammar $\bar{\Gamma}$ such that $L(\Gamma) = L(\bar{\Gamma})$ or $L(\Gamma) = L(\bar{\Gamma}) \cup \{0\}$.

Proof. We begin by computing $\ker(\Gamma)$. Then we obtain $\bar{\Gamma}$ by first adding all productions that can be obtained from the productions of Γ by deleting from their righthand sides one or more variables belonging to $\ker(\Gamma)$ and by then deleting all productions (old and new) of the form $V \rightarrow 0$. (In our example, $\bar{\Gamma}_0$ would have the productions $S \rightarrow XYYX$, $S \rightarrow aX$, $S \rightarrow a$, $S \rightarrow YYX$, $S \rightarrow XYX$, $S \rightarrow XYY$, $S \rightarrow XY$, $S \rightarrow YY$, $S \rightarrow YX$, $S \rightarrow XX$, $S \rightarrow X$, $S \rightarrow Y$.) We shall show that $L(\Gamma) = L(\bar{\Gamma})$ or $L(\Gamma) = L(\bar{\Gamma}) \cup \{0\}$.

Let $V \rightarrow \beta_1 \beta_2 \cdots \beta_s$ be a production of $\bar{\Gamma}$ that is not a production of Γ , where $\beta_1, \beta_2, \dots, \beta_s \in (\mathcal{V} \cup T)$, and where this production was obtained from a production of Γ of the form

$$V \rightarrow u_0 \beta_1 u_1 \beta_2 \cdots \beta_s u_s,$$

with $u_0, u_1, u_2, \dots, u_s \in (\ker(\Gamma))^*$. [Of course, u_0, u_s might be 0. But since $0 \in (\ker(\Gamma))^*$, this creates no difficulty.] Now,

$$u_i \xrightarrow[\bar{\Gamma}]{} 0, \quad i = 0, 1, 2, \dots, s,$$

so that

$$V \xrightarrow[\bar{\Gamma}]{} u_0 \beta_1 u_1 \beta_2 \cdots u_{s-1} \beta_s u_s \xrightarrow[\bar{\Gamma}]{} \beta_1 \beta_2 \cdots \beta_s.$$

Thus, the effect of this new production of $\bar{\Gamma}$ can be simulated in Γ . This proves that $L(\bar{\Gamma}) \subseteq L(\Gamma)$.

It remains to show that if $v \in L(\Gamma)$ and $v \neq 0$, then $v \in L(\bar{\Gamma})$. Let T be the set of terminals of Γ (and also of $\bar{\Gamma}$). We shall prove by induction the stronger assertion:

For any variable V , if $V \xrightarrow[\bar{\Gamma}]{} w \neq 0$ for $w \in T^$, then $V \xrightarrow[\bar{\Gamma}]{} w$.*

If in fact $V \xrightarrow[\bar{\Gamma}]{} w$, then Γ contains the production $V \rightarrow w$ which is also a production of $\bar{\Gamma}$. Otherwise we may write

$$V \xrightarrow[\bar{\Gamma}]{} w_0 V_1 w_1 V_2 w_2 \cdots V_s w_s \xrightarrow[\bar{\Gamma}]{} w,$$

where V_1, \dots, V_s are variables and $w_0, w_1, w_2, \dots, w_s$ are (possible null) words on the terminals. Then w can be written

$$w = w_0 v_1 w_1 v_2 w_2 \cdots v_s w_s,$$

where

$$V_i \xrightarrow[\bar{\Gamma}]{} v_i, \quad i = 1, 2, \dots, s.$$

Since each v_i must have a shorter derivation from V_i than w has from V , we may proceed inductively by assuming that for each v_i which is not 0, $V_i \xrightarrow[\bar{\Gamma}]{} v_i$. On the other hand, if $v_i = 0$, then $V_i \in \ker(\Gamma)$. We set

$$V_i^0 = \begin{cases} 0 & \text{if } v_i = 0 \\ V_i & \text{otherwise.} \end{cases}$$

Then $V \rightarrow w_0 V_1^0 w_1 V_2^0 w_2 \cdots V_s^0 w_s$ is one of the productions of $\bar{\Gamma}$. Hence we have

$$V \xrightarrow[\bar{\Gamma}]{} w_0 V_1^0 w_1 V_2^0 w_2 \cdots V_s^0 w_s \xrightarrow[\bar{\Gamma}]{} w_0 v_1 w_1 v_2 w_2 \cdots v_s w_s = w. \quad \blacksquare$$

We can now easily prove

Theorem 1.2. A language L is context-free if and only if there is a positive context-free grammar Γ such that

$$L = L(\Gamma) \text{ or } L = L(\Gamma) \cup \{0\}. \quad (1.1)$$

Moreover, there is an algorithm that will transform a context-free grammar Δ for which $L = L(\Delta)$ into a positive context-free grammar Γ that satisfies (1.1).

Proof. If L is context-free with $L = L(\Delta)$ for a context-free grammar Δ , then we can use the algorithm of Lemma 2 to construct a positive context-free grammar Γ such that $L = L(\Gamma)$ or $L = L(\Gamma) \cup \{0\}$.

Conversely, if Γ is a positive context-free grammar and $L = L(\Gamma)$, there is nothing to prove since a positive context-free grammar is already a context-free grammar. If $L = L(\Gamma) \cup \{0\}$, let S be the start symbol of Γ and let $\tilde{\Gamma}$ be the context-free grammar obtained from Γ by introducing \tilde{S} as a new start symbol and adding the productions

$$\tilde{S} \rightarrow S, \quad \tilde{S} \rightarrow 0.$$

Clearly, $L(\tilde{\Gamma}) = L(\Gamma) \cup \{0\}$. ■

Now, let Γ be a *positive* context-free grammar with alphabet $T \cup \mathcal{V}$, where T consists of the terminals and \mathcal{V} is the set of variables. We will make use of *trees* consisting of a finite number of points called *nodes* or *vertices*, each of which is labeled by a letter of the alphabet, i.e., an element of $T \cup \mathcal{V}$. Certain vertices will have other nodes as *immediate successors*, and the immediate successors of a given node are to be in some definite order. It is helpful (though of course not part of the formal development) to think of the immediate successors of a given node as being physically *below* the given node and arranged from left to right in their given order. Nodes are to be connected by line segments to their immediate successors. There is to be exactly one node which is not an immediate successor; this node is called the *root*. Each node other than the root is to be the immediate successor of precisely one node, its *predecessor*. Nodes which have no immediate successors are called *leaves*.

A tree is called a Γ -*tree* if it satisfies the following conditions:

1. the root is labeled by a variable;
2. each vertex which is not a leaf is labeled by a variable;
3. if a vertex is labeled X and its immediate successors are labeled $\alpha_1, \alpha_2, \dots, \alpha_k$ (reading from left to right), then $X \rightarrow \alpha_1 \alpha_2 \dots \alpha_k$ is a production of Γ .

Let \mathcal{T} be a Γ -tree, and let v be a vertex of \mathcal{T} which is labeled by the variable X . Then we shall speak of the *subtree* \mathcal{T}^v of \mathcal{T} determined by v . The vertices of \mathcal{T}^v are v , its immediate successors in \mathcal{T} , their immediate successors, and so on. The vertices of \mathcal{T}^v are labeled exactly as they are in \mathcal{T} . (In particular, the root of \mathcal{T}^v is v which is labeled X .) Clearly, \mathcal{T}^v is itself a Γ -tree.

If \mathcal{T} is a Γ -tree, we write $\langle \mathcal{T} \rangle$ for the word that consists of *the labels of the leaves of \mathcal{T}* reading from left to right (a vertex to the left of a given node is regarded as also being to the left of each of its immediate successors). If the root of \mathcal{T} is labeled by the start symbol S of Γ and if $w = \langle \mathcal{T} \rangle$, then \mathcal{T} is called a *derivation tree for w in Γ* . Thus the tree shown in Fig. 1.1 is a derivation tree for $a^{[4]}b^{[3]}$ in the grammar shown in the same figure.

Theorem 1.3. If Γ is a positive context-free grammar, and $S \xrightarrow{*}_{\Gamma} w$, then there is a derivation tree for w in Γ .

Proof. Our proof is by induction on the length of a derivation of w from S in Γ . If this length is 1, then $w = S$ and the required derivation tree consists of a single vertex labeled S (being both root and leaf).

Now let w have a derivation from S of length $r + 1$, where the result is known for derivations of length r . Then we have $S \xrightarrow{*} v = w$ with $v, w \in (\mathcal{V} \cup T)^*$, where the induction hypothesis applies to the derivation $S \xrightarrow{*} v$. Thus, we may assume that we have a derivation tree for v . Now since $v \Rightarrow w$, we must have $v = xXy$ and $w = x\alpha_1 \cdots \alpha_k y$, where Γ contains the production $X \rightarrow \alpha_1 \cdots \alpha_k$. Then the derivation tree for v can be extended to yield a derivation tree for w simply by giving k immediate successors to the node labeled X , labeled $\alpha_1, \dots, \alpha_k$ from left to right. ■

Before considering the converse of Theorem 1.3, it will be helpful to consider the following derivations of $a^{[4]}b^{[3]}$ from S with respect to the grammar indicated in Fig. 1.1:

1. $S \Rightarrow aXbY \Rightarrow a^{[2]}XbY \Rightarrow a^{[3]}XbY \Rightarrow a^{[4]}bY \Rightarrow a^{[4]}b^{[2]}Y \Rightarrow a^{[4]}b^{[3]}$
2. $S \Rightarrow aXbY \Rightarrow a^{[2]}XbY \Rightarrow a^{[2]}Xb^{[2]}Y \Rightarrow a^{[3]}Xb^{[2]}Y \Rightarrow a^{[3]}Xb^{[3]} \Rightarrow a^{[4]}b^{[3]}$
3. $S \Rightarrow aXbY \Rightarrow aXb^{[2]}Y \Rightarrow aXb^{[3]} \Rightarrow a^{[2]}Xb^{[3]} \Rightarrow a^{[3]}Xb^{[3]} \Rightarrow a^{[4]}b^{[3]}$.

Now, if the proof of Theorem 1.3 is applied to these three derivations, the very same derivation tree is obtained—namely, the one shown in Fig. 1.1. This shows that there does not exist a one-one correspondence between derivations and derivation trees, but that rather, several derivations may give rise to the same tree. Hence, there is no unique derivation which we can hope to be able to read off a given derivation tree.

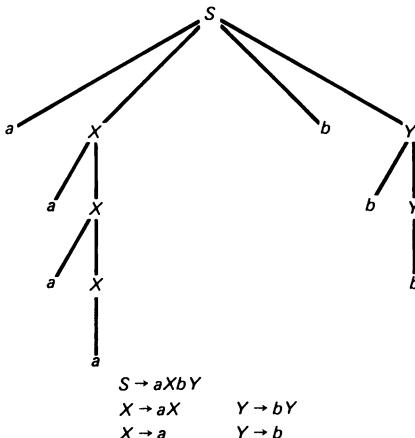


Figure 1.1. A derivation tree for $a^{[4]}b^{[3]}$ in the indicated grammar.

Definition. We write $u \Rightarrow_l v$ (in Γ) if $u = xXy$ and $v = xzy$, where $X \rightarrow z$ is a production of Γ and $x \in T^*$. If, instead, $x \in (T \cup \mathcal{V})^*$ but $y \in T^*$, we write $u \Rightarrow_r v$.

Thus, when $u \Rightarrow_l v$, it is the *leftmost* variable in u for which a substitution is made, whereas when $u \Rightarrow_r v$, it is the *rightmost* variable in u . A derivation

$$u_1 \Rightarrow_l u_2 \Rightarrow_l u_3 \Rightarrow_l \cdots \Rightarrow_l u_n$$

is called a *leftmost* derivation, and then we write $u_1 \stackrel{*}{\Rightarrow}_l u_n$. Similarly, a derivation

$$u_1 \Rightarrow_r u_2 \Rightarrow_r u_3 \Rightarrow_r \cdots \Rightarrow_r u_n$$

is called a *rightmost* derivation, and we write $u_1 \stackrel{*}{\Rightarrow}_r u_n$. In the preceding examples of derivations of $a^{[4]}b^{[3]}$ from S in the grammar of Fig. 1.1, 1 is leftmost, 3 is rightmost, and 2 is neither.

Now we shall see how, given a derivation tree \mathcal{T} for a word $w \in T^*$, we can obtain a *leftmost derivation of w from S* and a *rightmost derivation of w from S* . Let the word which consists of the labels of the immediate successors of the root of \mathcal{T} (reading from left to right) be $v_0X_1v_1X_2 \cdots X_rv_r$, where $v_0, v_1, \dots, v_r \in T^*$, $X_1, X_2, \dots, X_r \in \mathcal{V}$, and X_1, X_2, \dots, X_r label the vertices v_1, \dots, v_r , which are immediate successors of the root of \mathcal{T} . (Of course, some of the v_i may be 0.) Then $S \rightarrow v_0X_1v_1X_2 \cdots X_rv_r$ is one of the productions of Γ . Now it is possible that the immediate successors of the root of \mathcal{T} are all leaves; this is precisely the case where $w = v_0$ and $r = 0$. If this is the case, then we have $S \Rightarrow_l w$ and $S \Rightarrow_r w$, so that we do have a leftmost as well as a rightmost derivation of w from S .

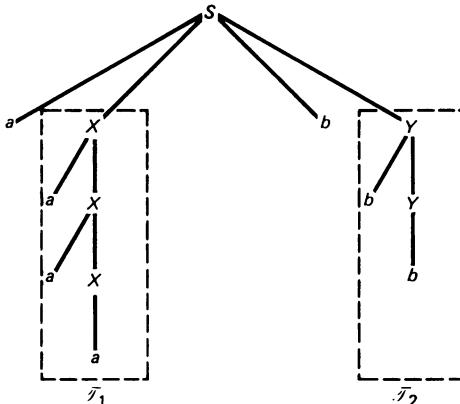


Figure 1.2. Decomposition of the tree of Fig. 1.1 as in the proof of the existence of leftmost and rightmost derivations.

Otherwise, i.e., for $r > 0$, we consider the trees $\mathcal{T}_i = \mathcal{T}^{\nu_i}$, $i = 1, 2, \dots, r$. Here \mathcal{T}_i has its root ν_i labeled X_i and is made up of the part of \mathcal{T} consisting of ν_i , its immediate successors, their immediate successors, etc. (see Fig. 1.2). Let Γ_i be the grammar whose productions and alphabet are the same as for Γ but which has start symbol X_i . Then \mathcal{T}_i is a derivation tree in Γ_i . Let \mathcal{T}_i be a derivation tree for w_i in Γ_i . Then, clearly,

$$w = v_0 w_1 v_1 w_2 v_2 \cdots w_r v_r.$$

Moreover, since each \mathcal{T}_i contains fewer vertices than \mathcal{T} , we may assume inductively that for $i = 1, 2, \dots, r$

$$X_i \xrightarrow{*} w_i \quad \text{and} \quad X_i \xrightarrow{*} w_i.$$

Hence we have

$$\begin{aligned} S &\Rightarrow_l v_0 X_1 v_1 X_2 \cdots X_r v_r \\ &\xrightarrow{*} v_0 w_1 v_1 X_2 \cdots X_r v_r \\ &\xrightarrow{*} v_0 w_1 v_1 w_2 \cdots X_r v_r \\ &\vdots \\ &\xrightarrow{*} v_0 w_1 v_1 w_2 \cdots w_r v_r = w \end{aligned}$$

and

$$\begin{aligned} S &\Rightarrow_r v_0 X_1 v_1 X_2 \cdots X_r v_r \\ &\xrightarrow{*} v_0 X_1 v_1 X_2 \cdots w_r v_r \\ &\vdots \\ &\xrightarrow{*} v_0 X_1 v_1 w_2 \cdots w_r v_r \\ &\xrightarrow{*} v_0 w_1 v_1 w_2 \cdots w_r v_r = w. \end{aligned}$$

So we have shown how to obtain a leftmost and a rightmost derivation of w from S in Γ .

Now, Theorem 1.3 tells us that if $w \in L(\Gamma)$, there is a derivation tree for w in Γ . And we have just seen that if there is a derivation tree for w in Γ , then there are both leftmost and rightmost derivations of w from S in Γ [so that, in particular, $w \in L(\Gamma)$]. Putting all of this information together we have

Theorem 1.4. Let Γ be a positive context-free grammar with start symbol S and terminals T . Let $w \in T^*$. Then the following conditions are equivalent:

1. $w \in L(\Gamma)$;
2. there is a derivation tree for w in Γ ;
3. there is a leftmost derivation of w from S in Γ ;
4. there is a rightmost derivation of w from S in Γ .

Definition. A positive context-free grammar is called *branching* if it has no productions of the form $X \rightarrow Y$, where X and Y are variables.

For a derivation tree in a branching grammar Γ , each vertex that is not a leaf cannot be the *only* immediate successor of its predecessor. Since we shall find it useful to work with branching grammars, we prove

Theorem 1.5. There is an algorithm that transforms a given positive context-free grammar Γ into a branching context-free grammar Δ such that $L(\Delta) = L(\Gamma)$.

Proof. Let \mathcal{V} be the set of variables of Γ . First suppose that Γ contains productions

$$X_1 \rightarrow X_2, \quad X_2 \rightarrow X_3, \quad \dots, \quad X_k \rightarrow X_1, \quad (1.2)$$

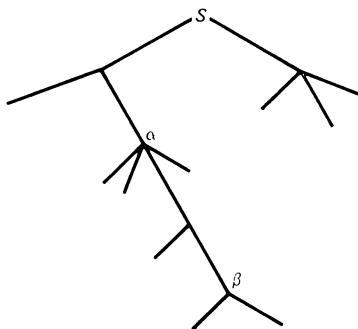
where $k \geq 1$ and $X_1, X_2, \dots, X_k \in \mathcal{V}$. Then, we can eliminate the productions (1.2) and replace each variable X_i in the remaining productions of Γ by the new variable X . (If one of X_1, \dots, X_k is the start symbol, then X must now be the start symbol.) Obviously the language generated is not changed by this transformation.

Thus, we need consider only the case where no “cycles” like (1.2) occur in Γ . If Γ is not branching, it must contain a production $X \rightarrow Y$ such that Γ contains no productions of the form $Y \rightarrow Z$. We eliminate the production $X \rightarrow Y$, but add to Γ productions $X \rightarrow x$ for each word $x \in (\mathcal{V} \cup T)^*$ for which $Y \rightarrow x$ is a production of Γ . Again the language generated is unchanged, but the number of productions that Γ contains of the form

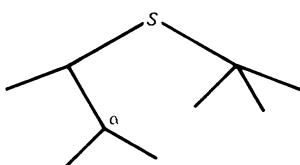
$U \rightarrow V$ has been decreased. Iterating this process we arrive at a grammar Δ containing no productions of the form $U \rightarrow V$, which is therefore of the required form. ■

A *path* in a Γ -tree \mathcal{T} is a sequence $\alpha_1, \alpha_2, \dots, \alpha_k$ of vertices of \mathcal{T} such that α_{i+1} is an immediate successor of α_i for $i = 1, 2, \dots, k - 1$. All of the vertices on the path are called *descendants* of α_1 .

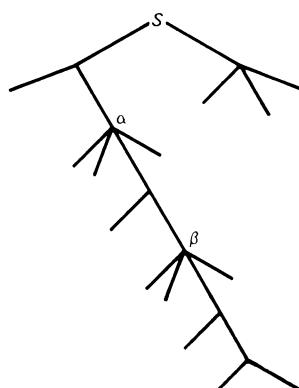
A particularly interesting situation arises when two different vertices α, β lie on the same path in the derivation tree \mathcal{T} and are labeled by the same variable X . In such a case one of the vertices is a descendant of the other, say, β is a descendant of α . \mathcal{T}^β is then not only a subtree of \mathcal{T} but also of \mathcal{T}^α . [In fact, $(\mathcal{T}^\alpha)^\beta = \mathcal{T}^\beta$.] We wish to consider two important



Original tree \mathcal{T}
(α, β are labeled by the same variable)



\mathcal{T} pruned



\mathcal{T} spliced

Figure 1.3

operations on the derivation tree \mathcal{T} which can be performed in this case. The first operation, which we call *pruning*, is to remove the subtree \mathcal{T}^α from the vertex α and to *graft* the subtree \mathcal{T}^β in its place. The second operation, which we call *splicing*, is to remove the subtree \mathcal{T}^β from the vertex β and to *graft* an exact copy of \mathcal{T}^α in its place. (See Fig. 1.3.) *Because α and β are labeled by the same variable, the trees obtained by pruning and splicing are themselves derivation trees.*

Let \mathcal{T}_p and \mathcal{T}_s be trees obtained from a derivation tree \mathcal{T} in a *branching* grammar by pruning and splicing, respectively, where α and β are as before. We have $\langle \mathcal{T} \rangle = r_1 \langle \mathcal{T}^\alpha \rangle r_2$ for words r_1, r_2 and $\langle \mathcal{T}^\alpha \rangle = q_1 \langle \mathcal{T}^\beta \rangle q_2$ for words q_1, q_2 . Since α, β are distinct vertices, and since the grammar is branching, q_1 and q_2 cannot both be 0. (That is, $q_1 q_2 \neq 0$.) Also,

$$\langle \mathcal{T}_p \rangle = r_1 \langle \mathcal{T}^\beta \rangle r_2 \quad \text{and} \quad \langle \mathcal{T}_s \rangle = r_1 q_1^{[2]} \langle \mathcal{T}^\beta \rangle q_2^{[2]} r_2. \quad (1.3)$$

Since $q_1 q_2 \neq 0$, we have $|\langle \mathcal{T}^\beta \rangle| < |\langle \mathcal{T}^\alpha \rangle|$ and hence $|\langle \mathcal{T}_p \rangle| < |\langle \mathcal{T} \rangle|$. From this last inequality and Theorem 1.4, we can easily infer

Theorem 1.6. Let Γ be a branching context-free grammar, let $u \in L(\Gamma)$, and let u have a derivation tree \mathcal{T} in Γ that has two different vertices on the same path labeled by the same variable. Then there is a word $v \in L(\Gamma)$ such that $|v| < |u|$.

Proof. Since $u = \langle \mathcal{T} \rangle$, we need only take $v = \langle \mathcal{T}_p \rangle$. ■

Exercises

- Find a context-free grammar generating the set of arithmetic statements of Pascal (or FORTRAN).
- Consider the grammar Γ with start symbol S and productions

$$\begin{array}{lll} S \rightarrow XXYY & X \rightarrow XX & Y \rightarrow YY \\ X \rightarrow a & & Y \rightarrow b. \end{array}$$

Show that Γ generates the same language as the grammar of Fig. 1.1.

- Show that \emptyset is a context-free language.
- Give three languages that are context-free but not regular. Justify your answer.
- Give a context-free grammar Γ such that $\ker(\Gamma) = \mathcal{V}_4$ and $\mathcal{V}_i \neq \ker(\Gamma)$, $i = 1, 2, 3$.

6. Let Γ be a context-free grammar with productions $X_1 \rightarrow \alpha_1, \dots, X_n \rightarrow \alpha_n$. We define the *width* of Γ as $\sum_{i=1}^n |\alpha_i|$.
- Give a function $f(w)$ such that for any context-free grammar Γ with width w , there is a positive context-free grammar $\bar{\Gamma}$ such that $L(\Gamma) = L(\bar{\Gamma})$ or $L(\Gamma) = L(\bar{\Gamma}) \cup \{0\}$ and $\bar{\Gamma}$ has no more than $f(w)$ productions.
 - Give a grammar Γ with width w for which any such $\bar{\Gamma}$ has at least $f(w)/2$ productions.
7. (a) Let Γ be the grammar in Exercise 2. Give two different derivation trees for $aaabb$. From each tree obtain a leftmost and a rightmost derivation of $aaabb$ from S .
- (b) Let Γ' be the grammar in Fig. 1.1. Prove that for every $w \in L(\Gamma')$, there is a unique derivation tree for w in Γ' .
8. Let Γ be the grammar with productions

$$\begin{array}{llll} S \rightarrow VW & W \rightarrow aW & X \rightarrow S & Y \rightarrow X \\ S \rightarrow W & W \rightarrow X & X \rightarrow W & Y \rightarrow c \\ V \rightarrow bX & W \rightarrow Y & X \rightarrow Z & Z \rightarrow V \\ V \rightarrow b & & & \end{array}$$

and start symbol S . Use the construction in the proof of Theorem 1.5 to give a branching context-free grammar Δ such that $L(\Delta) = L(\Gamma)$. Can any of the resulting productions be eliminated from Δ ?

2. Regular Grammars

We shall now see that regular languages are generated by context-free grammars of an especially simple form.

Definition. A context-free grammar is called *regular* if each of its productions has one of the two forms

$$U \rightarrow aV \quad \text{or} \quad U \rightarrow a,$$

where U, V are variables and a is a terminal.

Then we have

Theorem 2.1. If L is a regular language, then there is a regular grammar Γ such that either $L = L(\Gamma)$ or $L = L(\Gamma) \cup \{0\}$.

Proof. Let $L = L(\mathcal{M})$, where \mathcal{M} is a dfa with states q_1, \dots, q_m , alphabet $\{s_1, \dots, s_n\}$, transition function δ , and set of accepting states F . We

construct a grammar Γ with variables q_1, \dots, q_m , terminals s_1, \dots, s_n , and start symbol q_1 . The productions are

1. $q_i \rightarrow s_r q_j$ whenever $\delta(q_i, s_r) = q_j$, and
2. $q_i \rightarrow s_r$ whenever $\delta(q_i, s_r) \in F$.

Clearly the grammar Γ is regular. We shall show that $L(\Gamma)$ is just $L - \{0\}$.

First, suppose $u \in L$, $u \neq 0$; let $u = s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}}$. Thus, $\delta^*(q_1, u) \in F$, so that we have

$$\delta(q_1, s_{i_1}) = q_{j_1}, \quad \delta(q_{j_1}, s_{i_2}) = q_{j_2}, \quad \dots, \quad \delta(q_{j_l}, s_{i_{l+1}}) = q_{j_{l+1}} \in F. \quad (2.1)$$

Hence, the grammar Γ contains the productions

$$q_1 \rightarrow s_{i_1} q_{j_1}, \quad q_{j_1} \rightarrow s_{i_2} q_{j_2}, \quad \dots, \quad q_{j_{l-1}} \rightarrow s_{i_l} q_{j_l}, \quad q_{j_l} \rightarrow s_{i_{l+1}}. \quad (2.2)$$

Thus, we have in Γ

$$\begin{aligned} q_1 &\Rightarrow s_{i_1} q_{j_1} \\ &\Rightarrow s_{i_1} s_{i_2} q_{j_2} \\ &\vdots \\ &\Rightarrow s_{i_1} s_{i_2} \cdots s_{i_l} q_{j_l} \\ &\Rightarrow s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}} = u, \end{aligned} \quad (2.3)$$

so that $u \in L(\Gamma)$.

Conversely, suppose that $u \in L(\Gamma)$, $u = s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}}$. Then there is a derivation of u from q_1 in Γ , which must be of the form (2.3). Hence, the productions listed in (2.2) must belong to Γ , and finally, the transitions (2.1) must hold in \mathcal{M} . Thus, $u \in L(\mathcal{M})$. ■

Theorem 2.2. Let Γ be a regular grammar. Then $L(\Gamma)$ is a regular language.

Proof. Let Γ have the *variables* V_1, V_2, \dots, V_K , where $S = V_1$ is the start symbol, and the *terminals* s_1, \dots, s_n . Since Γ is assumed to be regular, its productions are of the form $V_i \rightarrow s_r V_j$ and $V_i \rightarrow s_r$. We shall construct an ndfa \mathcal{M} which accepts precisely $L(\Gamma)$.

The states of \mathcal{M} will be V_1, V_2, \dots, V_K and an additional state W . V_1 will be the initial state and W will be the only accepting state, i.e., $F = \{W\}$. Let

$$\delta_1(V_i, s_r) = \{V_j \mid V_i \rightarrow s_r V_j \text{ is a production of } \Gamma\},$$

$$\delta_2(V_i, s_r) = \begin{cases} \{W\} & \text{if } V_i \rightarrow s_r \text{ is a production of } \Gamma \\ \emptyset & \text{otherwise.} \end{cases}$$

Then we take as the transition function δ of \mathcal{M}

$$\delta(V_i, s_r) = \delta_1(V_i, s_r) \cup \delta_2(V_i, s_r).$$

This completes the specification of \mathcal{M} .

Now let $u = s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}} \in L(\Gamma)$. Thus, we must have

$$V_1 \Rightarrow s_{i_1} V_{j_1} \Rightarrow s_{i_1} s_{i_2} V_{j_2} \Rightarrow s_{i_1} s_{i_2} \cdots s_{i_l} V_{j_l} \Rightarrow s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}}, \quad (2.4)$$

where Γ contains the productions

$$\begin{aligned} V_1 &\rightarrow s_{i_1} V_{j_1}, \\ V_{j_1} &\rightarrow s_{i_2} V_{j_2}, \\ &\vdots \\ V_{j_{l-1}} &\rightarrow s_{i_l} V_{j_l}, \\ V_{j_l} &\rightarrow s_{i_{l+1}}. \end{aligned} \quad (2.5)$$

Thus,

$$\begin{aligned} V_{j_1} &\in \delta(V_1, s_{i_1}), \\ V_{j_2} &\in \delta(V_{j_1}, s_{i_2}), \\ &\vdots \\ V_{j_l} &\in \delta(V_{j_{l-1}}, s_{i_l}), \\ W &\in \delta(V_{j_l}, s_{i_{l+1}}). \end{aligned} \quad (2.6)$$

Thus, $W \in \delta^*(V_1, u)$ and $u \in L(\mathcal{M})$.

Conversely, if $u = s_{i_1} s_{i_2} \cdots s_{i_l} s_{i_{l+1}}$ is accepted by \mathcal{M} , then there must be a sequence of transitions of the form (2.6). Hence, the productions of (2.5) must all belong to Γ , so that there is a derivation of the form (2.4) of u from V_1 . ■

In order to combine Theorems 2.1 and 2.2 in a single equivalence, it is necessary to show only that if L is a regular language, then so is $L \cup \{0\}$. But this follows at once from Theorems 4.2 and 4.5 in Chapter 9.

Combining Theorems 2.1 and 2.2 with this discussion, we have

Theorem 2.3. A language L is regular if and only if there is a regular grammar Γ such that either $L = L(\Gamma)$ or $L = L(\Gamma) \cup \{0\}$.

Since regular grammars are context-free grammars, we have

Corollary 2.4. Every regular language is context-free.

The converse of Corollary 2.4 is not true, however, as we have already observed in Theorem 1.1.

There are more extensive classes of context-free grammars which can be shown to generate only regular languages. A particularly important example for us (see Section 7) is the class of *right-linear* grammars.

Definition. A context-free grammar is called *right-linear* if each of its productions has one of the two forms

$$U \rightarrow xV \quad \text{or} \quad U \rightarrow x, \quad (2.7)$$

where U, V are variables and $x \neq 0$ is a word consisting entirely of terminals.

Thus a regular grammar is just a right-linear grammar in which $|x| = 1$ for each string x in (2.7). We have

Theorem 2.5. Let Γ be a right-linear grammar. Then $L(\Gamma)$ is regular.

Proof. Given a right-linear grammar Γ , we construct a regular grammar $\bar{\Gamma}$ as follows.

We replace each production of Γ of the form

$$U \rightarrow a_1 a_2 \cdots a_n V, \quad n > 1,$$

by the productions

$$\begin{aligned} U &\rightarrow a_1 Z_1, \\ Z_1 &\rightarrow a_2 Z_2, \\ &\vdots \\ Z_{n-2} &\rightarrow a_{n-1} Z_{n-1}, \\ Z_{n-1} &\rightarrow a_n V, \end{aligned}$$

where Z_1, \dots, Z_{n-1} are new variables. Also, we replace each production

$$U \rightarrow a_1 a_2 \cdots a_n, \quad n > 1,$$

by a list of productions similar to the preceding list except that instead of the last production we have

$$Z_{n-1} \rightarrow a_n.$$

It is obvious that $\bar{\Gamma}$ is regular and that $L(\bar{\Gamma}) = L(\Gamma)$. ■

Exercises

1. (a) For each regular language L described in Exercise 1.1 of Chapter 9, give a regular grammar Γ such that $L(\Gamma) = L - \{0\}$.

- (b) For each dfa \mathcal{M} in Exercise 1.3 of Chapter 9, give a regular grammar Γ such that $L(\Gamma) = L(\mathcal{M}) - \{0\}$.
- (c) For each ndfa \mathcal{M} in Exercise 2.1 of Chapter 9, give a regular grammar Γ such that $L(\Gamma) = L(\mathcal{M}) - \{0\}$.
2. Let Γ be the grammar with productions

$$\begin{array}{llll} S \rightarrow aS & X \rightarrow bX & Z \rightarrow aZ & Z \rightarrow a \\ S \rightarrow aX & X \rightarrow bZ & Z \rightarrow bZ & Z \rightarrow b \\ S \rightarrow aY & Y \rightarrow cX & Z \rightarrow cZ & Z \rightarrow c \\ & Y \rightarrow cZ & & \end{array}$$

and start symbol S . Give an ndfa \mathcal{M} such that $L(\mathcal{M}) = L(\Gamma)$.

3. Let Γ be the grammar with productions

$$\begin{array}{lll} S \rightarrow aX & Y \rightarrow aY & Z \rightarrow aS \\ S \rightarrow bY & Y \rightarrow bZ & Z \rightarrow bS \\ X \rightarrow aZ & & Z \rightarrow a \\ X \rightarrow bX & & Z \rightarrow b \end{array}$$

and start symbol S .

- (a) Use the construction in the proof of Theorem 2.2 to give an ndfa \mathcal{M} with five states such that $L(\mathcal{M}) = L(\Gamma)$.
- (b) Transform \mathcal{M} into a dfa \mathcal{M}' with four states such that $L(\mathcal{M}') = L(\Gamma)$.
4. Prove that for every regular language L , there is a regular grammar Γ with start symbol S such that $L = L(\Gamma)$ or $L = L(\Gamma) \cup \{0\}$ and such that every $w \in L(\Gamma)$ has exactly one derivation from S in Γ .
5. Prove that for every $n \geq 1$, there is a regular language generated by no regular grammar with fewer than n variables.
6. (a) Write a context-free grammar to generate all and only regular expressions over the alphabet $\{a, b\}$.
 (b) Can a regular grammar generate this language? Support your answer.
7. A grammar Γ is *self-embedding* if there is a variable X such that

$$X \xrightarrow[\Gamma]^* vXw, \quad \text{where } v, w \in (\mathcal{V} \cup T)^* - \{0\}.$$

Let L be a context-free language. Prove that L is regular if and only if there is a non-self-embedding context-free grammar Γ such that $L(\Gamma) = L$.

8. For a language L , the *reverse* of L , denoted L^R , is $\{w^R \mid w \in L\}$.
- Let Γ be a regular grammar and let $L = L(\Gamma)$. Show that there is an ndfa which accepts L^R .
 - Conclude from (a) that a language L is regular if and only if L^R is regular.
 - Let Γ be a grammar such that all productions are of the form $U \rightarrow Vs$ or $U \rightarrow s$, where U, V are variables and s is a terminal. Show that $L(\Gamma)$ is regular.
 - A grammar is *left-linear* if each of its productions is of the form $U \rightarrow Vx$ or $U \rightarrow x$, where U, V are variables and x is a word consisting entirely of terminals. Prove that a language L is regular if and only if there is a left-linear grammar Γ such that $L = L(\Gamma)$ or $L = L(\Gamma) \cup \{0\}$.

3. Chomsky Normal Form

Although context-free grammars are extremely simply, there are even simpler special classes of context-free grammars that suffice to give all context-free languages. Such classes are called *normal forms*.

Definition. A context-free grammar Γ with variables \mathcal{V} and terminals T is in *Chomsky normal form* if each of its productions has one of the two forms

$$X \rightarrow YZ \quad \text{or} \quad X \rightarrow a,$$

where $X, Y, Z \in \mathcal{V}$ and $a \in T$.

Then we can prove

Theorem 3.1. There is an algorithm that transforms a given positive context-free grammar Γ into a Chomsky normal form grammar Δ such that $L(\Gamma) = L(\Delta)$.

Proof. Using Theorem 1.5, we begin with a branching context-free grammar Γ with variables \mathcal{V} and terminals T . We continue by “disguising” the terminals as variables. That is, for each $a \in T$ we introduce a new variable X_a . Then we modify Γ by replacing each production $X \rightarrow x$ for which x is not a single terminal by $X \rightarrow x'$, where x' is obtained from x by replacing each terminal a by the corresponding new variable X_a . In addition all of the productions $X_a \rightarrow a$ are added. Clearly the grammar thus obtained generates the same language as Γ and has all of its productions in one of

the two forms

$$X \rightarrow X_1 X_2 \dots X_k, \quad k \geq 2, \quad (3.1)$$

$$X \rightarrow a, \quad (3.2)$$

where X, X_1, \dots, X_k are variables and a is a terminal. To obtain a Chomsky normal form grammar we need to eliminate all of the productions of type (3.1) for which $k > 2$. We can do this by introducing the new variables Z_1, Z_2, \dots, Z_{k-2} and replacing (3.1) by the productions

$$\begin{aligned} X &\rightarrow X_1 Z_1 \\ Z_1 &\rightarrow X_2 Z_2 \\ &\vdots \\ Z_{k-3} &\rightarrow X_{k-2} Z_{k-2} \\ Z_{k-2} &\rightarrow X_{k-1} X_k. \end{aligned}$$

Thus we obtain a grammar in Chomsky normal form that generates $L(\Gamma)$. ■

As an example, let us convert the grammar of Fig. 1.1 to Chomsky normal form.

Step 1. Eliminate productions of the form $X_1 \rightarrow X_2$: there are no such productions so we skip this step.

Step 2. Disguise the terminals as variables: the grammar now consists of the productions

$$\begin{array}{lll} S \rightarrow X_a XX_b Y & X \rightarrow X_a X & Y \rightarrow X_b Y \\ X \rightarrow a & Y \rightarrow b & \\ X_a \rightarrow a & X_b \rightarrow b. & \end{array}$$

Step 3. Obtain Chomsky normal form by replacing the production $S \rightarrow X_a XX_b Y$ by the productions

$$\begin{aligned} S &\rightarrow X_a Z_1, \\ Z_1 &\rightarrow XZ_2, \\ Z_2 &\rightarrow X_b Y. \end{aligned}$$

The final Chomsky normal form grammar thus obtained consists of the productions

$$\begin{array}{llll} S \rightarrow X_a Z_1 & Z_1 \rightarrow XZ_2 & Z_2 \rightarrow X_b Y & \\ X \rightarrow X_a X & X \rightarrow a & Y \rightarrow X_b Y & X_b \rightarrow b \\ X_a \rightarrow a & & Y \rightarrow b & \end{array}$$

Exercises

1. (a) Find context-free grammars Γ_1, Γ_2 such that

$$L(\Gamma_1) = \{a^{[i]}b^{[j]} \mid i \geq j > 0\}$$

$$L(\Gamma_2) = \{a^{[2i]}b^{[i]} \mid i > 0\}.$$

- (b) Find Chomsky normal form grammars that generate the same languages.
 2. Let $T = \{\downarrow, p, q\}$ be the set of terminals for the grammar Γ :

$$S \rightarrow p, \quad S \rightarrow q, \quad S \rightarrow \downarrow SS.$$

Find a Chomsky normal form grammar that generates $L(\Gamma)$.

- 3.* A context-free grammar is said to be in *Greibach normal form* if every production of the grammar is of the form

$$X \rightarrow aY_1Y_2 \cdots Y_k, \quad k \geq 0,$$

where $a \in T$ and $X, Y_1, Y_2, \dots, Y_k \in \mathcal{V}$. Show that there is an algorithm that transforms any positive context-free grammar into one in Greibach normal form that generates the same language.

- 4.* Show that there is an algorithm that transforms any positive context-free grammar into a grammar that generates the same language for which every production is of the form

$$A \rightarrow a,$$

$$A \rightarrow aB,$$

or

$$A \rightarrow aBC,$$

$$A, B, C \in \mathcal{V}, a \in T.$$

4. Bar-Hillel's Pumping Lemma

An important application of Chomsky normal form is in the proof of the following key theorem, which is an analog for context-free languages of the pumping lemma for regular languages.

Theorem 4.1 (Bar-Hillel's Pumping Lemma). Let Γ be a Chomsky normal form grammar with exactly n variables, and let $L = L(\Gamma)$. Then, for

every $x \in L$ for which $|x| > 2^n$, we have $x = r_1 q_1 r q_2 r_2$, where

1. $|q_1 r q_2| \leq 2^n$;
2. $q_1 q_2 \neq 0$;
3. for all $i \geq 0$, $r_1 q_1^{[i]} r q_2^{[i]} r_2 \in L$.

Lemma. Let $S \stackrel{*}{\Rightarrow}_{\Gamma} u$, where Γ is a Chomsky normal form grammar. Suppose that \mathcal{T} is a derivation tree for u in Γ and that no path in \mathcal{T} contains more than k nodes. Then $|u| \leq 2^{k-2}$.

Proof. First, suppose that \mathcal{T} has just one leaf labeled by a terminal a . Then $u = a$, and \mathcal{T} has just two nodes, which are labeled S and a , respectively. Thus, no path in \mathcal{T} contains more than two nodes and $|u| = 1 \leq 2^{2-2}$.

Otherwise, since Γ is in Chomsky normal form, the root of \mathcal{T} must have exactly two immediate successors α, β in \mathcal{T} labeled by variables, say, X and Y , respectively. (In this case, Γ contains the production $S \rightarrow XY$.) Now we will consider the two trees $\mathcal{T}_1 = \mathcal{T}^\alpha$ and $\mathcal{T}_2 = \mathcal{T}^\beta$ whose roots are labeled X and Y , respectively. (See Fig. 4.1.)

In each of \mathcal{T}_1 and \mathcal{T}_2 the longest path must contain $\leq k-1$ nodes. Proceeding inductively, we may assume that each of $\mathcal{T}_1, \mathcal{T}_2$ have $\leq 2^{k-3}$ leaves. Hence,

$$|u| \leq 2^{k-3} + 2^{k-3} = 2^{k-2}. \quad \blacksquare$$

Proof of Theorem 4.1. Let $x \in L$, where $|x| > 2^n$, and let \mathcal{T} be a derivation tree for x in Γ . Let $\alpha_1, \alpha_2, \dots, \alpha_m$ be a path in \mathcal{T} where m is as large as possible. Then $m \geq n+2$. (For, if $m \leq n+1$, by the lemma, $|x| \leq 2^{n-1}$.)

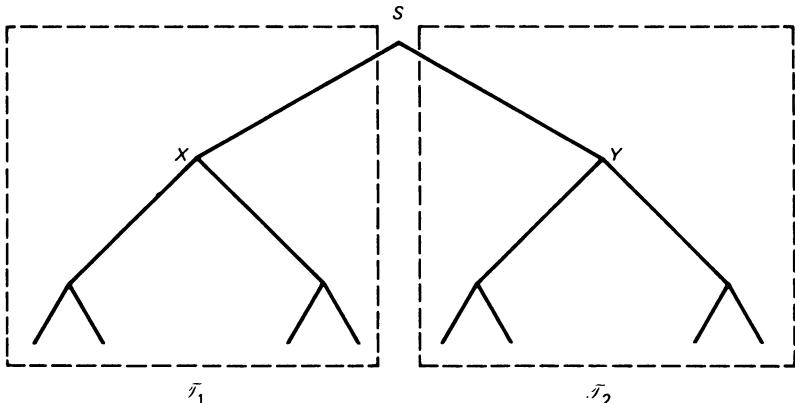


Figure 4.1

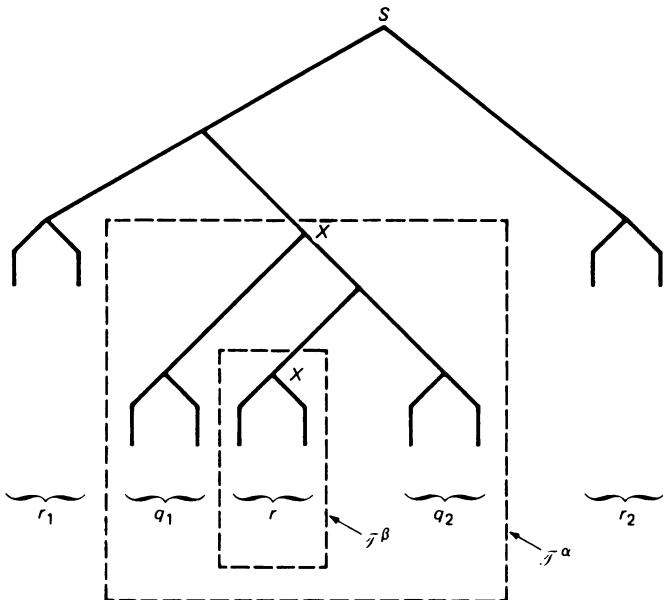


Figure 4.2

α_m is a leaf (otherwise we could get a longer path) and so is labeled by a terminal. $\alpha_1, \alpha_2, \dots, \alpha_{m-1}$ are all labeled by variables. Let us write

$$\gamma_i = \alpha_{m+i-n-2}, \quad i = 1, 2, \dots, n+2$$

so that the sequence of vertices $\gamma_1, \gamma_2, \dots, \gamma_{n+2}$ is simply the path consisting of the vertices

$$\alpha_{m-n-1}, \alpha_{m-n}, \alpha_{m-n+1}, \dots, \alpha_{m-1}, \alpha_m,$$

where $\gamma_{n+2} = \alpha_m$ is labeled by a terminal, and $\gamma_1, \dots, \gamma_{n+1}$ are labeled by variables. Since there are only n variables in the alphabet of Γ , the pigeon-hole principle guarantees that there is a variable X that labels two different vertices: $\alpha = \gamma_i$ and $\beta = \gamma_j$, $i < j$. (See Fig. 4.2.) Hence, the discussion of *pruning* and *splicing* at the end of Section 1 can be applied. We let the words $q_1 q_2, r_1, r_2$ be defined as in that discussion and set $r = \langle \mathcal{T}^\beta \rangle$. Then [recalling (1.3)] we have

$$\langle \mathcal{T}_p \rangle = r_1 r r_2,$$

$$\langle \mathcal{T}_s \rangle = r_1 q_1^{[2]} r q_2^{[2]} r_2,$$

$$\langle (\mathcal{T}_s)_s \rangle = r_1 q_1^{[3]} r q_2^{[3]} r_2.$$

Since pruning and splicing a derivation tree in Γ yields a new derivation tree in Γ , we see that all of these words belong to $L(\Gamma)$. If, in addition, we iterate the splicing operation, we see that all of the words $r_1 q_1^{[i]} r q_2^{[i]} r_2$, $i \geq 0$, belong to $L(\Gamma)$.

Finally, we note that the path $\gamma_i, \dots, \gamma_{n+2}$ in \mathcal{T}^α consists of $\leq n + 2$ nodes and that no path in \mathcal{T}^α can be longer. (This is true simply because if there were a path in \mathcal{T}^α consisting of more than $n + 3 - i$ vertices, it could be extended backward through $\alpha = \gamma_i$ to yield a path in \mathcal{T} consisting of more than m vertices.) Hence by the lemma ■

$$|q_1 r q_2| = |q_1 \langle \mathcal{T}^\beta \rangle q_2| = |\langle \mathcal{T}^\alpha \rangle| \leq 2^n.$$

As an example of the uses of Bar-Hillel's pumping lemma, we show that the language $L = \{a^{[n]} b^{[n]} c^{[n]} \mid n > 0\}$ is *not* context-free.

Suppose that L is context-free with $L = L(\Gamma)$, where Γ is a Chomsky normal form grammar with n variables. Choose k so large that $|a^{[k]} b^{[k]} c^{[k]}| > 2^n$ (i.e., choose $k > 2^n/3$). Then we would have $a^{[k]} b^{[k]} c^{[k]} = r_1 q_1 r q_2 r_2$, where, setting

$$x_i = r_1 q_1^{[i]} r q_2^{[i]} r_2,$$

we have $x_i \in L$ for $i = 0, 1, 2, 3, \dots$. In particular,

$$x_2 = r_1 q_1 q_1 r q_2 q_2 r_2 \in L.$$

Since the elements of L consist of a block of a 's, followed by a block of b 's, followed by a block of c 's, we see that q_1 and q_2 must each contain only one of these letters. Thus, one of the three letters occurs neither in q_1 nor in q_2 . But since as $i = 2, 3, 4, 5, \dots$, x_i contains more and more copies of q_1 and q_2 and since $q_1 q_2 \neq 0$, it is impossible for x_i to have the same number of occurrences of a , b , and c . This contradiction shows that L is not context-free.

We have proved

Theorem 4.2. The language $L = \{a^{[n]} b^{[n]} c^{[n]} \mid n > 0\}$ is not context-free.

Exercises

1. Show that $\{a^{[i]} \mid i \text{ is a prime number}\}$ is not context-free.
2. Show that $\{a^{[i^2]} \mid i > 0\}$ is not context-free.
3. Show that a context-free language on a one-letter alphabet is regular.

5. Closure Properties

We now consider for context-free languages, some of the closure properties previously discussed for regular languages.

Theorem 5.1. If L_1, L_2 are context-free languages, then so is $L_1 \cup L_2$.

Proof. Let $L_1 = L(\Gamma_1)$, $L_2 = L(\Gamma_2)$, where Γ_1, Γ_2 are context-free grammars with disjoint sets of variables \mathcal{V}_1 and \mathcal{V}_2 , and start symbols S_1, S_2 , respectively. Let Γ be the context-free grammar with variables $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \{S\}$ and start symbol S . The productions of Γ are those of Γ_1 and Γ_2 , together with the two additional productions $S \rightarrow S_1, S \rightarrow S_2$. Then obviously $L(\Gamma) = L(\Gamma_1) \cup L(\Gamma_2)$, so that $L_1 \cup L_2 = L(\Gamma)$. ■

Surprisingly enough, the class of context-free languages is not closed under intersection. In fact, let Γ_1 be the context-free grammar whose productions are

$$S \rightarrow Sc, \quad S \rightarrow Xc, \quad X \rightarrow aXb, \quad X \rightarrow ab.$$

Then clearly,

$$L_1 = L(\Gamma_1) = \{a^{[n]}b^{[n]}c^{[m]} \mid n, m > 0\}.$$

Now, let Γ_2 be the grammar whose productions are

$$S \rightarrow aS, \quad S \rightarrow aX, \quad X \rightarrow bXc, \quad X \rightarrow bc.$$

Then

$$L_2 = L(\Gamma_2) = \{a^{[m]}b^{[n]}c^{[n]} \mid n, m > 0\}.$$

Thus, L_1 and L_2 are context-free languages. But

$$L_1 \cap L_2 = \{a^{[n]}b^{[n]}c^{[n]} \mid n > 0\},$$

which, by Theorem 4.2, is not context-free. We have proved

Theorem 5.2. There are context-free languages L_1 and L_2 such that $L_1 \cap L_2$ is not context-free.

Corollary 5.3. There is a context-free language $L \subseteq A^*$ such that $A^* - L$ is not context-free.

Proof. Suppose otherwise, i.e., for every context-free language $L \subseteq A^*$, $A^* - L$ is also context-free. Then the De Morgan identity

$$L_1 \cap L_2 = A^* - ((A^* - L_1) \cap (A^* - L_2))$$

together with Theorem 5.1 would contradict Theorem 5.2. ■

Although, as we have just seen, the intersection of context-free languages need not be context-free, the situation is different if one of the two languages is regular.

Theorem 5.4. If R is a regular language and L is a context-free language, then $R \cap L$ is context-free.

Proof. Let A be an alphabet such that $L, R \subseteq A^*$. Let $L = L(\Gamma)$ or $L(\Gamma) \cup \{0\}$, where Γ is a positive context-free grammar with variables \mathcal{V} , terminals A and start symbol S . Finally, let \mathcal{M} be a dfa that accepts R with states Q , initial state $q_1 \in Q$, accepting states $F \subseteq Q$, and transition function δ . Now, for each symbol $\sigma \in A \cup \mathcal{V}$ and each ordered pair $p, q \in Q$, we introduce a new symbol σ^{pq} . We shall construct a positive context-free grammar $\tilde{\Gamma}$ whose terminals are just the elements of A (i.e., the terminals of Γ) and whose set of variables consists of a start symbol \tilde{S} together with all of the new symbols σ^{pq} for $\sigma \in A \cup \mathcal{V}$ and $p, q \in Q$. (Note that for $a \in A$, a is a terminal, but a^{pq} is a variable for each $p, q \in Q$.) The productions of $\tilde{\Gamma}$ are as follows:

1. $\tilde{S} \rightarrow S^{q_1 q}$ for all $q \in F$.
2. $X^{pq} \rightarrow \sigma_1^{p r_1} \sigma_2^{r_1 r_2} \cdots \sigma_n^{r_{n-1} q}$ for all productions $X \rightarrow \sigma_1 \sigma_2 \cdots \sigma_n$ of Γ and all $p, r_1, r_2, \dots, r_{n-1}, q \in Q$.
3. $a^{pq} \rightarrow a$ for all $a \in A$ and all $p, q \in Q$ such that $\delta(p, a) = q$.

We shall now prove that $L(\tilde{\Gamma}) = L(\Gamma) \cap R$. Since $\tilde{\Gamma}$ is clearly a positive context-free grammar, and since $R \cap L = L(\tilde{\Gamma})$ or $R \cap L = L(\tilde{\Gamma}) \cup \{0\}$, the theorem follows from Theorem 1.2.

First let $u = a_1 a_2 \cdots a_n \in L(\Gamma) \cap R$. Since $u \in L(\Gamma)$, we have

$$S \xrightarrow[\tilde{\Gamma}]{} a_1 a_2 \cdots a_n.$$

Using productions 1 and 2 of $\tilde{\Gamma}$, we have

$$\tilde{S} \xrightarrow[\tilde{\Gamma}]{} S^{q_1, q_{n+1}} \xrightarrow[\tilde{\Gamma}]{} a_1^{q_1 q_2} a_2^{q_2 q_3} \cdots a_n^{q_n q_{n+1}}, \quad (5.1)$$

where q_2, q_3, \dots, q_n are arbitrary states of \mathcal{M} and q_{n+1} is any state in F . (q_1 is of course the initial state.) But since $u \in L(\mathcal{M})$, we can choose the states q_2, q_3, \dots, q_{n+1} so that

$$\delta(q_i, a_i) = q_{i+1}, \quad i = 1, 2, \dots, n, \quad (5.2)$$

and $q_{n+1} \in F$. In this case, not only does (5.1) hold, but also the productions

$$a_i^{q_i q_{i+1}} \rightarrow a_i, \quad i = 1, 2, \dots, n, \quad (5.3)$$

all belong to $\tilde{\Gamma}$. Hence, finally,

$$\tilde{S} \xrightarrow[\tilde{\Gamma}]{}^* a_1 a_2 \cdots a_n.$$

Conversely, let $\tilde{S} \xrightarrow[\tilde{\Gamma}]{}^* u \in A^*$. We shall need the following

Lemma. Let $\sigma^{pq} \xrightarrow[\tilde{\Gamma}]{}^* u \in A^*$. Then, $\delta^*(p, u) = q$. Moreover, if σ is a variable, then $\sigma \xrightarrow[\tilde{\Gamma}]{}^* u$.

Since $\tilde{S} \xrightarrow[\tilde{\Gamma}]{}^* S^{q_1 q} \xrightarrow[\tilde{\Gamma}]{}^* u$ where $q \in F$, we can use the Lemma to conclude that $\delta^*(q_1, u) = q$, and $S \xrightarrow[\tilde{\Gamma}]{}^* u$. Hence, $u \in R \cap L(\Gamma)$. Theorem 5.4 then follows immediately. ■

It remains to prove the Lemma.

Proof of Lemma. The proof is by induction on the length of a derivation of u from σ^{pq} in $\tilde{\Gamma}$. If that length is 2, we must have $\sigma \in A$, $u = \sigma$. Then, $\delta^*(p, u) = \delta(p, u) = q$. Otherwise we can write

$$\sigma^{pq} \xrightarrow[\tilde{\Gamma}]{}^* \sigma_1^{r_0 r_1} \sigma_2^{r_1 r_2} \cdots \sigma_n^{r_{n-1} r_n} \xrightarrow[\tilde{\Gamma}]{}^* u$$

where we have written $r_0 = p$ and $r_n = q$. Thus, we have

$$\sigma_i^{r_{i-1} r_i} \xrightarrow[\tilde{\Gamma}]{}^* u_i, \quad i = 1, 2, \dots, n, \quad (5.4)$$

where $u = u_1 u_2 \cdots u_n$. Clearly, the induction hypothesis can be applied to the derivations in (5.4) so that $\delta^*(r_{i-1}, u_i) = r_i$, $i = 1, 2, \dots, n$. Hence $\delta^*(p, u) = r_n = q$. Also, if σ_i is a variable, the induction hypothesis will give $\sigma_i \xrightarrow[\tilde{\Gamma}]{}^* u_i$, while otherwise $\sigma_i \in A$ and $\sigma_i = u_i$. Finally,

$$\sigma \rightarrow \sigma_1 \sigma_2 \cdots \sigma_n$$

must be a production of Γ . Hence, we have

$$\sigma \xrightarrow[\tilde{\Gamma}]{}^* \sigma_1 \sigma_2 \cdots \sigma_n$$

$$\xrightarrow[\tilde{\Gamma}]{}^* u_1 u_2 \cdots u_n = u. \quad \blacksquare$$

Let A, P be alphabets such that $P \subseteq A$. For each letter $a \in A$, let us write

$$a^0 = \begin{cases} 0 & \text{if } a \in P \\ a & \text{if } a \in A - P. \end{cases}$$

If $x = a_1a_2 \cdots a_n \in A^*$, we write

$$\text{Er}_P(x) = a_1^0a_2^0 \cdots a_n^0.$$

In other words, $\text{Er}_P(x)$ is the word that results from x when all the symbols in it that are part of the alphabet P are “erased.” If $L \subseteq A^*$, we also write

$$\text{Er}_P(L) = \{\text{Er}_P(x) \mid x \in L\}.$$

Finally, if Γ is any context-free grammar with terminals T and if $P \subseteq T$, we write $\text{Er}_P(\Gamma)$ for the context-free grammar with terminals $T - P$, the same variables and start symbol as Γ , and productions

$$X \rightarrow \text{Er}_P(v)$$

for each production $X \rightarrow v$ of Γ . [Note that even if Γ is a positive context-free grammar, $\text{Er}_P(\Gamma)$ may not be positive; that is, it is possible that $\text{Er}_P(v) = 0$ even if $v \neq 0$.] We have

Theorem 5.5. If Γ is a context-free grammar and $\tilde{\Gamma} = \text{Er}_P(\Gamma)$, then $L(\tilde{\Gamma}) = \text{Er}_P(L(\Gamma))$.³

Proof. Let S be the start symbol of Γ and $\tilde{\Gamma}$. Suppose that $w \in L(\tilde{\Gamma})$. We have

$$S = w_1 \xrightarrow{\tilde{\Gamma}} w_2 \cdots \xrightarrow{\tilde{\Gamma}} w_m = w.$$

Let $v_i = \text{Er}_P(w_i)$, $i = 1, 2, \dots, m$. Then clearly,

$$S = v_1 \xrightarrow{\tilde{\Gamma}} v_2 \cdots \xrightarrow{\tilde{\Gamma}} v_m = \text{Er}_P(w),$$

so that $\text{Er}_P(w) \in L(\tilde{\Gamma})$. This proves that $L(\tilde{\Gamma}) \supseteq \text{Er}_P(L(\Gamma))$.

To complete the proof it will suffice to show that whenever $X \xrightarrow{*} v \in (T - P)^*$, there is a word $w \in T^*$ such that $X \xrightarrow{*} w$ and $v = \text{Er}_P(w)$. We do this by induction on the length of a derivation of v from X in $\tilde{\Gamma}$. If $X \xrightarrow{*} v$, then $X \rightarrow v$ is a production of $\tilde{\Gamma}$, so that $X \rightarrow w$ is a production of Γ for some w with $\text{Er}_P(w) = v$. Proceeding inductively, let there be a derivation of v from X in $\tilde{\Gamma}$ of length $k > 2$, where the result is known

³ Readers familiar with the terminology may enjoy noting that this theorem states that the “operators” L and Er_P commute.

for all derivations of length $< k$. Then, we can write

$$X \xrightarrow[\Gamma]{} u_0 V_1 u_1 V_2 u_2 \cdots V_s u_s \xrightarrow[\Gamma]{*} v,$$

where $u_0, u_1, \dots, u_s \in (T - P)^*$ and V_1, V_2, \dots, V_s are variables. Thus, there are words $\bar{u}_0, \bar{u}_1, \dots, \bar{u}_s \in T^*$ such that $u_i = \text{Er}_P(\bar{u}_i)$, $i = 0, 1, \dots, s$, and

$$X \rightarrow \bar{u}_0 V_1 \bar{u}_1 V_2 \bar{u}_2 \cdots V_s \bar{u}_s$$

is a production of Γ . Also we can write

$$v = u_0 v_1 u_1 v_2 u_2 \cdots v_s u_s,$$

where

$$V_i \xrightarrow[\Gamma]{*} v_i, \quad i = 1, \dots, s. \quad (5.5)$$

Since (5.5) clearly involves derivations of length $< k$, the induction hypothesis applies, and we can conclude that there are words $\bar{v}_i \in T^*$, $i = 1, 2, \dots, s$, such that $v_i = \text{Er}_P(\bar{v}_i)$ and $V_i \xrightarrow[\Gamma]{*} \bar{v}_i$, $i = 1, 2, \dots, s$. Hence, we have

$$X \xrightarrow[\Gamma]{} \bar{u}_0 V_1 \bar{u}_1 V_2 \bar{u}_2 \cdots V_s \bar{u}_s \xrightarrow[\Gamma]{*} \bar{u}_0 \bar{v}_1 \bar{u}_1 \bar{v}_2 \bar{u}_2 \cdots \bar{v}_s \bar{u}_s.$$

But

$$\text{Er}_P(\bar{u}_0 \bar{v}_1 \bar{u}_1 \bar{v}_2 \cdots \bar{v}_s \bar{u}_s) = u_0 v_1 u_1 v_2 u_2 \cdots v_s u_s = v,$$

which completes the proof. ■

Corollary 5.6. If $L \subseteq A^*$ is a context-free language and $P \subseteq A$, then $\text{Er}_P(L)$ is also a context-free language.

Proof. Let $L = L(\Gamma)$, where Γ is a context-free grammar, and let $\tilde{\Gamma} = \text{Er}_P(\Gamma)$. Then, by Theorem 5.5, $\text{Er}_P(L) = L(\tilde{\Gamma})$, so that $\text{Er}_P(L)$ is context-free. ■

Exercises

- For each of the following, give languages L_1, L_2 on alphabet $\{a, b\}$ such that
 - L_1, L_2 are context-free but not regular, and $L_1 \cup L_2$ is regular;
 - L_1, L_2 are context-free, $L_1 \neq L_2$, and $L_1 \cup L_2$ is not regular;

- (c) L_1, L_2 are context-free but not regular, $L_1 \cap L_2 \neq \emptyset$, and $L_1 \cap L_2$ is regular;
- (d) L_1, L_2 are context-free but not regular, $L_1 \neq L_2$, and $L_1 \cap L_2$ is context-free but not regular.
2. Let L, L' be context-free languages. Prove the following.
- $L \cdot L'$ is context-free.
 - L^* is context-free.
 - $L^R = \{w^R \mid w \in L\}$ is context-free.
3. Give languages R, L_1, L_2 on alphabet $\{a, b\}$ such that R is regular, L_1, L_2 are context-free but not regular, and
- $R \cap L_1$ is regular;
 - $R \cap L_2$ is not regular.
4. Give a context-free language L on alphabet $A = \{a, b\}$ such that L is not regular and $A^* - L$ is context-free.
5. Let $R = \{a^{[m]}b^{[n]} \mid m \geq 0, n > 0\}$, $L = \{a^{[n]}b^{[n]} \mid n > 0\}$. Use the construction in the proof of Theorem 5.4 to give a grammar Γ such that $L(\Gamma) = R \cap L$.
6. Give alphabets A, P such that $P \neq \emptyset$, $P \subseteq A$, $P \neq A$, and give languages $L_1, L_2 \subseteq A^*$ such that
- L_1 is not context-free and $\text{Er}_P(L_1)$ is regular;
 - L_2 is context-free and $\text{Er}_P(L_2)$ is not regular.
7. Prove that if $L \subseteq A^*$ is regular and $P \subseteq A$, then $\text{Er}_P(L)$ is also regular.
8. Let A_1, A_2 be alphabets, let $L \subseteq A_1^*$ be context-free, let f be a substitution on A_1 such that $f(a) \subseteq A_2^*$ is context-free for all $a \in A_1$, and let g be a homomorphism from A_1^* to A_2^* . [See Exercise 4.5 in Chapter 9 for the definitions of *substitution* and *homomorphism*.]
- Prove that $f(L)$ is context-free.
 - Prove that $g(L)$ is context-free.
9. Let $A_1 = \{a_1, \dots, a_n\}$, let $L \subseteq A_1^*$ be context-free, and let $R \subseteq A_1^*$ be regular.
- Let $A_2 = \{a'_1, \dots, a'_n\}$, where $A_1 \cap A_2 = \emptyset$, and let f be a substitution on A_1 such that $f(a_i) = \{a_i, a'_i\}$, $1 \leq i \leq n$. Show that $A_2^* \cdot R \cap f(L)$ is context-free. [See Exercise 8.]
 - Let g be the homomorphism on $A_1 \cup A_2$ such that $g(a_i) = 0$ and $g(a'_i) = a_i$, $1 \leq i \leq n$. Show that $g(A_2^* \cdot R \cap f(L))$ is context-free.

- (c) Show that $g(A_2^* \cdot R \cap f(L)) = L/R$, the right quotient of L by R . [See Exercise 7.13 in Chapter 9 for the definition of *right quotient*.]
- (d) Conclude that if L is context-free and R is regular, then L/R is context-free.

*6. Solvable and Unsolvable Problems⁴

Let Γ be a context-free grammar with terminals T and start symbol S , let $u \in T^*$, and let us consider the problem of determining whether $u \in L(\Gamma)$. First let $u = 0$. Then we can use the algorithms provided in Section 1 to compute $\ker(\Gamma)$. Since $0 \in L(\Gamma)$ if and only if $S \in \ker(\Gamma)$, we can answer the question in this case. For $u \neq 0$, we use Theorems 1.2 and 3.1 to obtain a Chomsky normal form grammar Δ such that $u \in L(\Gamma)$ if and only if $u \in L(\Delta)$. To test whether $u \in L(\Delta)$, we use the following:

Lemma. Let Δ be a Chomsky normal form grammar with terminals T . Let V be a variable of Δ and let

$$V \xrightarrow[\Delta]{} u \in T^*.$$

Then there is a derivation of u from V in Δ of length $2|u|$.

Proof. The proof is by induction on $|u|$. If $|u| = 1$, then u is a terminal and Δ must contain a production $V \rightarrow u$, so that we have a derivation of u from V of length 2.

Now, let $V \xrightarrow[\Delta]{} u$, where $|u| > 1$, and let us assume the result known for all strings of length $< |u|$. Recalling the definition of a Chomsky normal form grammar, we see that

$$V \Rightarrow XY \xrightarrow{*} u.$$

Thus, we must have $X \xrightarrow{*} v$, $Y \xrightarrow{*} w$, $u = vw$ where $|v|, |w| < |u|$. By the induction hypothesis we have derivations

$$\begin{aligned} X &= \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_{2|v|} = v, \\ Y &= \beta_1 \Rightarrow \beta_2 \Rightarrow \cdots \Rightarrow \beta_{2|w|} = w. \end{aligned}$$

Hence, we can write the derivation

$$V \Rightarrow XY = \alpha_1 Y \Rightarrow \alpha_2 Y \Rightarrow \cdots \Rightarrow \alpha_{2|v|} Y = v \beta_1 \Rightarrow v \beta_2 \Rightarrow \cdots \Rightarrow v \beta_{2|w|},$$

⁴ The * does *not* refer to the material through Theorem 6.4.

where $v\beta_{2|w|} = vw = u$. But this derivation is of length $2|v| + 2|w| = 2|u|$, which completes the proof. ■

Now to test $u \in L(\Delta)$, we simply write out all derivations from S of length $2|u|$. We have $u \in L(\Delta)$ if and only if at least one of these derivations terminates in the string u .

We have proved

Theorem 6.1.⁵ There is an algorithm that will test a given context-free grammar Γ and a given word u to determine whether $u \in L(\Gamma)$.

Next we wish to consider the question of whether a given context-free grammar generates the empty language \emptyset . Let Γ be a given context-free grammar. We first check as previously to decide whether $0 \in L(\Gamma)$. If $0 \in L(\Gamma)$, we know that $L(\Gamma) \neq \emptyset$. Otherwise we use Theorems 1.2 and 1.5 to obtain a branching context-free grammar $\bar{\Gamma}$ such that $L(\Gamma) = L(\bar{\Gamma})$. Let $\bar{\Gamma}$ have n variables and set of terminals T . Suppose that $L(\bar{\Gamma}) \neq \emptyset$. Let $u \in L(\bar{\Gamma})$, where u has the shortest possible length of any word in $L(\bar{\Gamma})$. Then *in any derivation tree for u in $\bar{\Gamma}$, each path contains fewer than $n + 2$ nodes*. This is because, if there were a path containing at least $n + 2$ nodes, at least $n + 1$ of them would be labeled by variables, and by the pigeon-hole principle, Theorem 1.6 would apply and yield a word $v \in L(\bar{\Gamma})$ with $|v| < |u|$. Thus, we conclude that

$L(\bar{\Gamma}) \neq \emptyset$ if and only if there is a derivation tree \mathcal{T} in Γ of a word $u \in T^*$ such that each path in \mathcal{T} contains fewer than $n + 2$ nodes.

It is a straightforward matter (at least in principle) to write out explicitly all derivation trees in Γ in which no path has length $\geq n + 2$. To test whether $L(\bar{\Gamma}) \neq \emptyset$, it suffices to note whether there is such a tree \mathcal{T} for which $\langle \mathcal{T} \rangle \in T^*$. Thus we have

Theorem 6.2. There is an algorithm to test a given context-free grammar Γ to determine whether $L(\Gamma) = \emptyset$.

Next we seek an algorithm to test whether $L(\Gamma)$ is finite or infinite for a given context-free grammar Γ . Such an algorithm can easily be obtained from the following.

⁵ This result follows at once from Theorem 5.4 in Chapter 7; but the algorithm given here is of some independent interest.

Theorem 6.3. Let Γ be a Chomsky normal form grammar with exactly n variables. Then $L(\Gamma)$ is infinite if and only if there is a word $x \in L(\Gamma)$ such that

$$2^n < |x| \leq 2^{n+1}.$$

Proof. If there is a word $x \in L(\Gamma)$ with $|x| > 2^n$, then by Bar-Hillel's pumping lemma, $L(\Gamma)$ is infinite.

Conversely, let $L(\Gamma)$ be infinite. Let u be a word of shortest possible length such that $u \in L(\Gamma)$ and $|u| > 2^{n+1}$. By Bar-Hillel's pumping lemma, we have $u = r_1 q_1 r q_2 r_2$ where $q_1 q_2 \neq 0$, $|q_1 r q_2| \leq 2^n$ and $x = r_1 r r_2 \in L(\Gamma)$. Now,

$$|x| \geq |r_1 r_2| = |u| - |q_1 r q_2| > 2^n.$$

Since $|x| < |u|$, the manner in which we chose u guarantees that $|x| \leq 2^{n+1}$. ■

Theorem 6.4. There is an algorithm to test a given context-free grammar Γ to determine whether $L(\Gamma)$ is finite or infinite.

Proof. Given context-free grammar Γ with terminals T , we use the algorithms of Theorems 1.2 and 3.1 to construct a Chomsky normal form grammar Δ with $L(\Gamma) = L(\Delta)$ or $L(\Delta) \cup \{0\}$. Let Δ have n variables and let $l = 2^n$. Then we simply use Theorem 6.1 to test each word $u \in T^*$ for which $l < |u| \leq 2l$ to see whether $u \in L(\Gamma)$. $L(\Gamma)$ is infinite if and only if at least one of these words u does belong to $L(\Gamma)$. ■

Remarkably enough, there are also some very simple unsolvable problems related to context-free grammars.⁶ The easiest way to obtain these results is to associate a pair of context-free grammars with each Post correspondence system.

Thus, suppose we are given the finite set of dominoes:



$i = 1, 2, \dots, n$, where $u_i, v_i \in A^*$ for some given alphabet A . We introduce n new symbols c_1, c_2, \dots, c_n and define two context-free grammars Γ_1, Γ_2 , both of which have as their terminals $A \cup \{c_1, c_2, \dots, c_n\}$. Γ_1 has the

⁶ The remainder of this section depends on Chapter 7. Readers who have not covered this material should move on to Section 7.

single variable S_1 , its start symbol, and Γ_2 has S_2 as its only variable and start symbol. The productions of Γ_1 are

$$S_1 \rightarrow u_i S_1 c_i, \quad S_1 \rightarrow u_i c_i, \quad i = 1, 2, \dots, n,$$

and those of Γ_2 are

$$S_2 \rightarrow v_i S_2 c_i, \quad S_2 \rightarrow v_i c_i, \quad i = 1, 2, \dots, n.$$

Now, the given Post correspondence system has a solution if and only if we can have

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m}$$

for some sequence i_1, i_2, \dots, i_m . Moreover,

$$L(\Gamma_1) = \{u_{i_1} u_{i_2} \cdots u_{i_m} c_{i_m} \cdots c_{i_2} c_{i_1}\}$$

and

$$L(\Gamma_2) = \{v_{i_1} v_{i_2} \cdots v_{i_m} c_{i_m} \cdots c_{i_2} c_{i_1}\}.$$

Thus, we have

Theorem 6.5. $L(\Gamma_1) \cap L(\Gamma_2) \neq \emptyset$ if and only if the given Post correspondence problem has a solution.

Using Theorem 4.1 in Chapter 7, we conclude

Theorem 6.6. There is no algorithm to test a given pair of context-free grammars Γ_1, Γ_2 to determine whether $L(\Gamma_1) \cap L(\Gamma_2) = \emptyset$.

Another important unsolvability result about context-free grammars concerns ambiguity.

Definition. A context-free grammar Γ is called *ambiguous* if there is a word $u \in L(\Gamma)$ that has two different leftmost derivations in Γ . If Γ is not ambiguous, it is said to be *unambiguous*.

Theorem 6.7. There is no algorithm to test a given context-free grammar to determine whether it is ambiguous.

Proof. Once again we begin with a Post correspondence system, and form the two context-free grammars Γ_1, Γ_2 used in proving Theorem 6.5. Γ_1 and Γ_2 are obviously both unambiguous. Now let Γ have start symbol S and all of the productions of Γ_1 and Γ_2 , together with $S \rightarrow S_1$ and $S \rightarrow S_2$. Then, since the first step of a derivation from S in Γ involves an irreversible

commitment to either Γ_1 or Γ_2 , Γ will be ambiguous just in case $L(\Gamma_1) \cap L(\Gamma_2) \neq \emptyset$. By Theorem 6.5 this will be the case if and only if the given Post correspondence system has a solution. The result now follows again from Theorem 4.1 in Chapter 7. ■

Another unsolvability result is given in Exercise 8.16.

Exercises

- Let Γ_1 be the grammar with productions $S \rightarrow aS$, $S \rightarrow a$, and let Γ_2 be the grammar with productions $S \rightarrow SS$, $S \rightarrow a$.
 - How many derivation trees are there for $a^{[6]}$ in Γ_1 ? In Γ_2 ?
 - How many derivations of $a^{[4]}$ from S are there in Γ_1 ? In Γ_2 ?
 - How many leftmost derivations of $a^{[6]}$ from S are there in Γ_1 ? In Γ_2 ?
- Write a context-free grammar Γ such that

$$L(\Gamma) = \{a^{[i]}b^{[j]}c^{[k]} \mid i = j \vee j = k\}.$$

This language is an example of an *inherently ambiguous language*, i.e., a language such that every grammar that generates it is ambiguous. Explain why this language is inherently ambiguous.

- Give an unambiguous context-free grammar that generates the same language as the ambiguous grammar

$$\begin{aligned} S &\rightarrow aB \\ S &\rightarrow Ab \\ A &\rightarrow aAB \\ B &\rightarrow ABb \\ A &\rightarrow a \\ B &\rightarrow b. \end{aligned}$$

7. Bracket Languages

Let A be some finite set. Although we think of A as an alphabet, we will also wish to permit $A = \emptyset$. Let B be the alphabet we get from A by adjoining the $2n$ new symbols $\langle_i, i\rangle$, $i = 1, 2, \dots, n$, where n is some given positive integer. We will write $\text{PAR}_n(A)$ for the language consisting of all strings in B^* that are correctly “paired,” thinking of each pair $\langle_i, i\rangle$ as matching left and right brackets. More precisely, $\text{PAR}_n(A) = L(\Gamma_0)$, where

Γ_0 is the context-free grammar with the single variable S , terminals B , and the productions

1. $S \rightarrow a$ for all $a \in A$,
2. $S \rightarrow_i^{\langle} S_i^{\rangle}, i = 1, 2, \dots, n.$
3. $S \rightarrow SS, S \rightarrow 0.$

The languages $\text{PAR}_n(A)$ are called *bracket languages*.

Let us consider the example $A = \{a, b, c\}$, $n = 2$. For ease of reading we will use the symbol (for ${}_1^{\langle}$,) for ${}_1^{\rangle}$, [for ${}_2^{\langle}$, and] for ${}_2^{\rangle}$. Then $cb[(ab)c](a[b]c) \in \text{PAR}_2(A)$, as the reader should easily verify. Also, $(\cdot)[] \in \text{PAR}_2(A)$, since we have

$$S \Rightarrow SS \xrightarrow{*} (S)[S] \xrightarrow{*} (\cdot)[\cdot].$$

Bracket languages have the following properties.

Theorem 7.1. $\text{PAR}_n(A)$ is a context-free language such that

- a. $A^* \subseteq \text{PAR}_n(A)$;
- b. if $x, y \in \text{PAR}_n(A)$, so is xy ;
- c. if $x \in \text{PAR}_n(A)$, so is ${}_i^{\langle} x_i^{\rangle}$, for $i = 1, 2, \dots, n$;
- d. if $x \in \text{PAR}_n(A)$ and $x \notin A^*$, then we can write $x = u {}_i^{\langle} v_i^{\rangle} w$, for some $i = 1, 2, \dots, n$, where $u \in A^*$ and $v, w \in \text{PAR}_n(A)$.

Proof. Since $\text{PAR}_n(A) = L(\Gamma_0)$ where Γ_0 is a context-free grammar, $\text{PAR}_n(A)$ must be context-free. Property a follows at once on considering the productions 1 and 3. For b, let $S \xrightarrow{*} x, S \xrightarrow{*} y$. Then using the productions 3, we have

$$S \Rightarrow SS \xrightarrow{*} xy.$$

For c, let $S \xrightarrow{*} x$. Then using the productions 2, we have

$$S \Rightarrow_i^{\langle} S_i^{\rangle} \xrightarrow{*} {}_i^{\langle} x_i^{\rangle}.$$

To prove d, note first that we can assume $|x| > 1$ because otherwise $x \in A^*$. Then, a derivation of x from S must begin by using a production containing S on the right. We proceed by induction assuming the result for all strings of length $< |x|$. There are two cases.

Case 1. $S \Rightarrow_i^{\langle} S_i^{\rangle} \xrightarrow{*} {}_i^{\langle} v_i^{\rangle} = x$, where $S \xrightarrow{*} v$; the result then follows (without using the induction hypothesis) with $u = w = 0$.

Case 2. $S \Rightarrow SS \xrightarrow{*} rs = x$ where $S \xrightarrow{*} r, S \xrightarrow{*} s$, and $r \neq 0, s \neq 0$. Clearly, $|r|, |s| < |x|$. If $r \in A^*$, then $|s| > 1$ and we can use the induction hypothesis to write $s = u {}_i^{\langle} v_i^{\rangle} w$, where $u \in A^*$ and $v, w \in$

$\text{PAR}_n(A)$, and the desired result follows since $ru \in A^*$. Otherwise, we can use the induction hypothesis to write $r = u_i^l v_i^l w$ where $u \in A^*$ and $v, w \in \text{PAR}_n(A)$, so that the result follows since $ws \in \text{PAR}_n(A)$ by b. ■

Historically, the special case $A = \emptyset$ has played an important role in studying context-free languages. The language $\text{PAR}_n(\emptyset)$ is called the *Dyck language* of order n and is usually written D_n .

Now let us begin with a Chomsky normal form grammar Γ , with terminals T and productions

$$X_i \rightarrow Y_i Z_i, \quad i = 1, 2, \dots, n, \quad (7.1)$$

in addition to certain productions of the form $V \rightarrow a$ with $a \in T$. We will construct a new grammar Γ_s which we call the *separator* of Γ . The terminals of Γ_s are the symbols of T together with $2n$ new symbols $(,)_i$, $i = 1, 2, \dots, n$. Thus a pair of “brackets” has been added for each of the productions (7.1).

The productions of Γ_s are

$$X_i \rightarrow_i (Y_i) Z_i, \quad i = 1, 2, \dots, n,$$

as well as all of the productions of Γ of the form $V \rightarrow a$ with $a \in T$.

As an example, let Γ have the productions

$$\begin{aligned} S &\rightarrow XY, & S &\rightarrow YX, & Y &\rightarrow ZZ, \\ X &\rightarrow a, & Z &\rightarrow a. \end{aligned}$$

Then Γ is ambiguous as we can see from the leftmost derivations:

$$\begin{aligned} S &\Rightarrow XY \Rightarrow aY \Rightarrow aZZ \Rightarrow aaZ \Rightarrow aaa, \\ S &\Rightarrow YX \Rightarrow ZZX \Rightarrow aZX \Rightarrow aaX \Rightarrow aaa. \end{aligned}$$

The productions of Γ_s can be written

$$\begin{aligned} S &\rightarrow (X)Y, & S &\rightarrow [Y]X, & Y &\rightarrow \{Z\}Z, \\ X &\rightarrow a, & Z &\rightarrow a, \end{aligned}$$

using $($, $[$, and $\{$ in place of the numbered brackets. The two derivations just given then become

$$\begin{aligned} S &\Rightarrow (X)Y \Rightarrow (a)Y \Rightarrow (a)\{Z\}Z \Rightarrow (a)\{a\}Z \Rightarrow (a)\{a\}a, \\ S &\Rightarrow [Y]X \Rightarrow [\{Z\}Z]X \Rightarrow [\{a\}Z]X \Rightarrow [\{a\}a]X \Rightarrow [\{a\}a]a. \end{aligned}$$

Γ_s thus separates the two derivations in Γ . The bracketing in the words $(a)\{a\}a, [\{a\}a]a$ enables their respective derivation trees to be recovered.

If we write P for the set of brackets \langle , \rangle_i , $i = 1, 2, \dots, n$, then clearly $\Gamma = \text{Er}_P(\Gamma_s)$. Hence by Theorem 5.5,

Theorem 7.2. $\text{Er}_P(L(\Gamma_s)) = L(\Gamma)$.

We also will prove

Lemma 1. $L(\Gamma_s) \subseteq \text{PAR}_n(T)$.

Proof. We show that if $X \xrightarrow[\Gamma_s]{}^* w \in (T \cup P)^*$ for any variable X , then $w \in \text{PAR}_n(T)$. The proof is by induction on the length of a derivation of w from X in Γ_s . If this length is 2, then w is a single terminal and the result is clear. Otherwise we can write

$$X = X_i \xrightarrow[\Gamma_s]{}^* i Y_{ii} Z_i \xrightarrow[\Gamma_s]{}^* i u_i v = w,$$

where $Y_i \xrightarrow[\Gamma_s]{}^* u$ and $Z_i \xrightarrow[\Gamma_s]{}^* v$. By the induction hypothesis, $u, v \in \text{PAR}_n(T)$. By b and c of Theorem 7.1, so is w . ■

Now let Δ be the grammar whose variables, start symbol, and terminals are those of Γ_s and whose productions are as follows:

1. all productions $V \rightarrow a$ from Γ (or equivalently Γ_s) with $a \in T$,
2. all productions $X_i \rightarrow i Y_i$, $i = 1, 2, \dots, n$,
3. all productions $V \rightarrow a_i Z_i$, $i = 1, 2, \dots, n$, for which $V \rightarrow a$ is a production of Γ with $a \in T$.

We have

Lemma 2. $L(\Delta)$ is regular.

Proof. Since Δ is obviously right-linear, the result follows at once from Theorem 2.5. ■

Lemma 3. $L(\Gamma_s) \subseteq L(\Delta)$.

Proof. We show that if $X \xrightarrow[\Gamma_s]{}^* u \in (T \cup P)^*$ then $X \xrightarrow[\Delta]{}^* u$. If u has a derivation of length 2, then $u \in T$, and $X \rightarrow u$ is a production of Γ_s and of Γ and therefore also of Δ . Thus $X \xrightarrow[\Delta]{}^* u$.

Proceeding by induction, let

$$X = X_i \xrightarrow[\Gamma_s]{}^* i Y_{ii} Z_i \xrightarrow[\Gamma_s]{}^* i v_i w = u,$$

where the induction hypothesis applies to $Y_i \xrightarrow[\Gamma_s]{} v$ and to $Z_i \xrightarrow[\Gamma_s]{} w$. Thus, $Y_i \xrightarrow[\Delta]{} v$ and $Z_i \xrightarrow[\Delta]{} w$. Let $v = za$, $a \in T$. (See Exercise 3.) Then, examining the productions of the grammar Δ , we see that we must have

$$Y_i \xrightarrow[\Delta]{} zV \xrightarrow[\Delta]{} za = v,$$

where $V \rightarrow a$ is a production of Γ . But then we have

$$X_i \xrightarrow[\Delta]{} Y_i \xrightarrow[\Delta]{} (zV \xrightarrow[\Delta]{} za) Z_i \xrightarrow[\Delta]{} (v_i) w = u.$$
■

Lemma 4. $L(\Delta) \cap \text{PAR}_n(T) \subseteq L(\Gamma_s)$.

Proof. Let $X \xrightarrow[\Delta]{} u$, where $u \in \text{PAR}_n(T)$. We shall prove that $X \xrightarrow[\Gamma_s]{} u$. The proof is by induction on the total number of occurrences of the symbols \langle , \rangle_i in u . If this number is 0, then, examining the productions of Δ , we see that $u \in T$ and the production $X \rightarrow u$ is in Δ and hence in Γ_s .

Thus $X \xrightarrow[\Gamma_s]{} u$.

Now let $X \xrightarrow[\Delta]{} u$, where u contains occurrences of the bracket symbols \langle , \rangle_i and where the result is known for words v containing fewer occurrences of these symbols than u . Examining the productions of Δ , we see that our derivation of u from X must begin with one of the productions 2. (If the derivation began with a production of the form 1, then u would be a terminal. If the derivation began with a production of the form 3, then $u = a\}w$ for some word w , which is impossible by Theorem 7.1d.) Therefore $u = \langle_i z$, for some word z and some $i = 1, 2, \dots, n$. By Theorem 7.1d, $u = \langle_i v_i \rangle_i w$, where $v, w \in \text{PAR}_n(T)$. In our derivation of u in Δ , the symbol \rangle_i can only arise from the use of one of the productions of the form 3, say, $V \rightarrow a_i^j Z_i$, where $a \in T$ and $V \rightarrow a$ is a production of Γ . Then v must end in a , so that we can write $v = \bar{v}a$, where

$$X = X_i \xrightarrow[\Delta]{} Y_i \xrightarrow[\Delta]{} (\bar{v}V \xrightarrow[\Delta]{} \bar{v}a_i) Z_i \xrightarrow[\Delta]{} (v_i) w$$

and $Z_i \xrightarrow[\Delta]{} w$. Moreover, since $V \rightarrow a$ is a production of Γ , it is also one of the productions of Δ of the form 1. Therefore, we have in Δ

$$Y_i \xrightarrow[\Delta]{} \bar{v}V \xrightarrow[\Delta]{} \bar{v}a = v.$$

Since v and w must each contain fewer occurrences of \langle , \rangle_i than u , we have by the induction hypothesis

$$Y_i \xrightarrow[\Gamma_s]{} v, \quad Z_i \xrightarrow[\Gamma_s]{} w.$$

Hence,

$$X_i \xrightarrow[\Gamma_s]{} {}^i Y_{ii} Z_i \xrightarrow[\Gamma_s]{*} {}^i v_i w = u. \quad \blacksquare$$

We are now ready to state

Theorem 7.3. Let Γ be a grammar in Chomsky normal form with terminals T . Then there is a regular language R such that

$$L(\Gamma_s) = R \cap \text{PAR}_n(T).$$

Proof. Let Δ be defined as above and let $R = L(\Delta)$. The result then follows at once from Lemmas 1–4. \blacksquare

Theorem 7.4 (Chomsky–Schützenberger Representation Theorem). A language $L \subseteq T^*$ is context-free if and only if there is a regular language R and a number n such that

$$L = \text{Er}_P(R \cap \text{PAR}_n(T)), \quad (7.2)$$

where $P = \{\langle i, i \rangle \mid i = 1, 2, \dots, n\}$.

Proof. It is clear by Theorems 7.2 and 7.3 that for every grammar Γ in Chomsky normal form, $L = L(\Gamma)$ satisfies (7.2). For an arbitrary context-free language L , by Theorems 1.2 and 3.1, there is a Chomsky normal form grammar Γ such that

$$L = L(\Gamma) \text{ or } L = L(\Gamma) \cup \{0\}.$$

If

$$L(\Gamma) = \text{Er}_P(R \cap \text{PAR}_n(T)),$$

then

$$L(\Gamma) \cup \{0\} = \text{Er}_P((R \cup \{0\}) \cap \text{PAR}_n(T))$$

since, by Theorem 7.1a, $0 \in \text{PAR}_n(T)$. But, by Theorems 4.2 and 4.5 in Chapter 9, $R \cup \{0\}$ is a regular language.

It remains only to show that any language L that satisfies (7.2) must be context-free. But since, by Theorem 7.1, $\text{PAR}_n(T)$ is context-free, this result follows at once from Theorem 5.4 and Corollary 5.6. \blacksquare

The Chomsky–Schützenberger theorem is usually expressed in terms of the Dyck languages $D_n = \text{PAR}_n(\emptyset)$. Since our form of the theorem is equivalent to the more usual form, we will give only a very brief sketch of the proof of the usual form. It is necessary to go back to the construction

of Γ_s . Each element a of T is now thought of as a “left bracket” and is supplied with a “twin” a' to act as its corresponding right bracket. A new grammar Γ_t is then defined to have the same productions $X_i \rightarrow_i Y_{ii}$, $i = 1, 2, \dots, n$, as Γ_s but to have productions

$$V \rightarrow aa'$$

for each production $V \rightarrow a$ of Γ . Then clearly, $L(\Gamma_t)$ can be obtained from $L(\Gamma_s)$ by simply replacing all occurrences of letters $a \in T$ in words of $L(\Gamma_s)$ by aa' . By replacing a by aa' in productions of the forms 1 and 3 of Δ , we obtain a right linear grammar Δ' such that

$$L(\Gamma_t) = L(\Delta') \cap \text{PAR}_n(T'),$$

where $T' = \{a, a' \mid a \in T\}$. But in fact $L(\Gamma_t) \subseteq D_m$, where $m = n + k$ and there are k letters in T . Thus,

$$L(\Gamma_t) = L(\Delta') \cap D_m.$$

Finally letting $Q = \{\langle, \rangle_i \mid i = 1, 2, \dots, n\} \cup \{a' \mid a \in T\}$, we have

$$L(\Gamma) = \text{Er}_Q(L(\Gamma_t)) = \text{Er}_Q(L(\Delta') \cap D_m).$$

Thus, we get

Theorem 7.5. A language L is context-free if and only if there is a regular language R , an alphabet Q , and an integer m such that

$$L = \text{Er}_Q(R \cap D_m).$$

Exercises

- Let A be a finite set of symbols, n a positive integer, and $\text{PAR}_n(A) = L(\Gamma_0)$, where Γ_0 is the grammar given in the definition of $\text{PAR}_n(A)$. Show that Γ_0 is ambiguous. [See Section 6 for the definition of *ambiguous grammars*.]
- Let Γ be the grammar with productions

$$\begin{array}{lll} S \rightarrow XZ & X \rightarrow a \\ S \rightarrow XY & Y \rightarrow b & Z \rightarrow SY \end{array}$$

and start symbol S .

- (a) Give Γ_s .
- (b) Give Δ , as defined following Lemma 1, for Γ .

- (c) Show that $L(\Gamma_s) \neq \text{PAR}_3(\{a, b\})$.
 - (d) Show that $L(\Gamma_s) \neq L(\Delta)$.
3. Let Γ be a grammar in Chomsky normal form with variables \mathcal{V} and terminals T , and let $T \cup P$ be the terminals of Γ_s . Prove that for all $V \in \mathcal{V}$ and all w such that $V \xrightarrow{\Gamma_s} w$, $w = vs$ for some $v \in (\mathcal{V} \cup T \cup P)^*$ and some $s \in \mathcal{V} \cup T$.
4. Let Γ be a regular grammar, and let Γ' be the Chomsky normal form grammar derived from Γ by the construction in the proof of Theorem 3.1. Prove that $L(\Gamma'_s)$ is regular.

8. Pushdown Automata

We are now ready to discuss the question of what kind of automaton is needed for accepting context-free languages. We take our cue from Theorem 7.2, and begin by trying to construct an appropriate automaton for recognizing $L(\Gamma_s)$, where Γ is a given Chomsky normal form grammar. We know that $L(\Gamma_s) = R \cap \text{PAR}_n(T)$, where R is a regular language. Thus R is accepted by a finite automaton. The problem we need to solve is this: what additional facilities does this finite automaton require in order to check that some given word belongs to $\text{PAR}_n(T)$? Those familiar with “stacks” and their uses will see at once that what is needed is a “pushdown stack” as an auxiliary storage device. Such a device behaves in a last-in-first-out manner. At each step in a computation with a pushdown stack one or both of a pair of operations can be performed:

1. The symbol at the “top” of the stack may be read and discarded.
(This operation is called *popping the stack*.)
2. A new symbol may be “pushed” onto the stack.

A stack can be used to identify a string as belonging to $\text{PAR}_n(T)$ as follows: For each pair $\langle i, i \rangle$, $i = 1, 2, \dots, n$, a special symbol J_i is introduced. Now, as our automaton moves from left to right over a string, it pushes J_i onto the stack whenever it sees $\langle i, i \rangle$, and it pops the stack, eliminating a J_i , whenever it sees \rangle_i . Such an automaton will successfully scan the entire string and terminate with an empty stack just in case the string belongs to $\text{PAR}_n(T)$.

To move toward making these ideas precise, let T be a given alphabet and let $P = \{\langle i, i \rangle \mid i = 1, 2, \dots, n\}$. Let $\Omega = \{J_1, J_2, \dots, J_n\}$, where we have introduced a single symbol J_i for each pair $\langle i, i \rangle$, $1 \leq i \leq n$. Let $u \in (T \cup P)^*$, say, $u = c_1 c_2 \dots c_k$, where $c_1, c_2, \dots, c_k \in T \cup P$. We define a

sequence $\gamma_j(u)$ of elements of Ω^* as follows:

$$\begin{aligned}\gamma_1(u) &= 0 \\ \gamma_{j+1}(u) &= \begin{cases} \gamma_j(u) & \text{if } c_j \in T \\ J_i \gamma_j(u) & \text{if } c_j = i \\ \alpha & \text{if } c_j = i \text{ and } \gamma_j(u) = J_i \alpha, \end{cases}\end{aligned}$$

for $j = 1, 2, \dots, k$. Note that if $c_j = i$, $\gamma_{j+1}(u)$ will be undefined unless γ_j begins with the symbol J_i for the very same value of i . Of course, if a particular $\gamma_r(u)$ is undefined, all $\gamma_j(u)$ with $j > r$ will also be undefined.

Definition. We say that the word $u \in (T \cup P)^*$ is *balanced* if $\gamma_j(u)$ is defined for $1 \leq j \leq |u| + 1$ and $\gamma_{|u|+1}(u) = 0$.

The heuristic considerations with which we began suggest

Theorem 8.1. Let T be an alphabet and let

$$P = \left\{ \begin{matrix} (,) \\ i, i \end{matrix} \mid i = 1, 2, \dots, n \right\}, \quad T \cap P = \emptyset.$$

Let $u \in (T \cup P)^*$, let $\Omega = \{J_1, J_2, \dots, J_n\}$. Then $u \in \text{PAR}_n(T)$ if and only if u is balanced.

The proof is via a series of easy lemmas.

Lemma 1. If $u \in T^*$, then u is balanced.

Proof. Clearly $\gamma_j(u) = 0$ for $1 \leq j \leq |u| + 1$ in this case. ■

Lemma 2. If u and v are balanced, so is uv .

Proof. Clearly $\gamma_j(uv) = \gamma_j(u)$ for $1 \leq j \leq |u| + 1$. Since $\gamma_{|u|+1}(u) = 0 = \gamma_{|u|+1}(uv) = \gamma_j(v)$, we have $\gamma_{|u|+j}(uv) = \gamma_j(v)$ for $1 \leq j \leq |v| + 1$. Hence, $\gamma_{|uv|+1}(uv) = \gamma_{|u|+|v|+1}(uv) = \gamma_{|v|+1}(v) = 0$. ■

Lemma 3. Let $v = i u$. Then u is balanced if and only if v is balanced.

Proof. We have $\gamma_1(v) = 0$, $\gamma_2(v) = J_i$, $\gamma_{j+1}(v) = \gamma_j(u) J_i$, $j = 1, 2, \dots, |v| - 1$. In particular, $\gamma_{|v|}(v) = \gamma_{|u|+2}(v) = \gamma_{|u|+1}(u) J_i$. Thus, if u is balanced, then $\gamma_{|u|+1}(u) = 0$, so that $\gamma_{|v|}(v) = J_i$ and $\gamma_{|v|+1}(v) = 0$. Conversely, if v is balanced, $\gamma_{|v|+1}(v) = 0$, so that $\gamma_{|v|}(v)$ must be J_i and $\gamma_{|u|+1}(u) = 0$. ■

Lemma 4. If u is balanced and uv is balanced, then v is balanced.

Proof. $\gamma_j(uv) = \gamma_j(u)$ for $1 \leq j \leq |u| + 1$. Since $\gamma_{|u|+1}(u) = 0$, we have $\gamma_{|u|+j}(uv) = \gamma_j(v)$ for $1 \leq j \leq |v| + 1$. Finally,

$$0 = \gamma_{|uv|+1}(uv) = \gamma_{|u|+|v|+1}(uv) = \gamma_{|v|+1}(v). \quad \blacksquare$$

Lemma 5. If $u \in \text{PAR}_n(T)$, then u is balanced.

Proof. The proof is by induction on the total number of occurrences of the symbols \langle, \rangle_i in u . If this number is 0, then $u \in T^*$, so by Lemma 1, u is balanced.

Proceeding by induction, let u have $k > 0$ occurrences of the symbols \langle, \rangle_i , where the result is known for all strings with fewer than k occurrences of these symbols. Then, by Theorem 7.1d, we can write $u = v_i^l w_i^l z$, where $v, w, z \in \text{PAR}_n(T)$. By the induction hypothesis, v, w, z are all balanced, and by Lemmas 2 and 3, u is therefore balanced. \blacksquare

Lemma 6. If u is balanced, then $u \in \text{PAR}_n(T)$.

Proof. If $u \in T^*$, the result follows from Theorem 7.1a. Otherwise, we can write $u = xy$, where $x \in T^*$ and the initial symbol of y is in P . By the definition of $\gamma_j(u)$, we will have $\gamma_j(u) = 0$ for $1 \leq j \leq |x| + 1$. Therefore, the initial symbol of y cannot be one of the \langle, \rangle_i . Thus we can write $u = x_i^l z$, and $\gamma_{|x|+2}(u) = J_i$. Since u is balanced, $\gamma_{|u|+1}(u) = 0$, and we can let k be the least integer $> |x| + 1$ for which $\gamma_k(u) = 0$. Then $\gamma_{k-1}(u) = J_i$ and the $(k - 1)$ th symbol of u must be \rangle_i . Thus $u = x_i^l v_i^l w$, where $k = |x| + |v| + 3$. Thus $0 = \gamma_{|x|+|v|+3}(u) = \gamma_{|x|+|v|+3}(x_i^l v_i^l)$. Hence $x_i^l v_i^l$ is balanced. By Lemma 4, w is balanced. Since $x \in T^*$, x is balanced, and by Lemma 4 again, $\langle_i v_i^l \rangle$ is balanced. By Lemma 3, v is balanced. Since $x \in T^*$, $x \in \text{PAR}_n(T)$. Since $|v|, |w| < |u|$, we can assume by mathematical induction that it is already known that $v, w \in \text{PAR}_n(T)$. By b and c of Theorem 7.1, we conclude that $u \in \text{PAR}_n(T)$. \blacksquare

Theorem 8.1 is an immediate consequence of Lemmas 5 and 6.

We now give a precise definition of *pushdown automata*. We begin with a finite set of *states* $Q = \{q_1, \dots, q_m\}$, q_1 being the *initial state*, a subset $F \subseteq Q$ of *final*, or *accepting*, states, a *tape alphabet* A , and a *pushdown alphabet* Ω . (We usually use lowercase letters for elements of A and capital letters for elements of Ω .) We assume that the symbol $\mathbf{0}$ does not belong to either A or Ω and write $\bar{A} = A \cup \{\mathbf{0}\}$, $\bar{\Omega} = \Omega \cup \{\mathbf{0}\}$. A *transition* is a quintuple of the form

$$q_i a U : V q_j$$

where $a \in \bar{A}$ and $U, V \in \bar{\Omega}$. Intuitively, if $a \in A$ and $U, V \in \Omega$, this is to read: "In state q_i scanning a , with U on top of the stack, move one square

to the right, ‘pop’ the stack removing U , ‘push’ V onto the stack, and enter state q_j .” If $a = \mathbf{0}$, motion to the right does not take place and the stack action can occur regardless of what symbol is actually being scanned. Similarly, $U = \mathbf{0}$ indicates that nothing is to be popped and $V = \mathbf{0}$ that nothing is to be pushed. A *pushdown automaton* is specified by a finite set of transitions. The *distinct* transitions $q_i a U : V q_j$, $q_i b W : X q_k$ are called *incompatible* if one of the following is the case:

1. $a = b$ and $U = W$;
2. $a = b$ and U or W is $\mathbf{0}$;
3. $U = W$ and a or b is $\mathbf{0}$;
4. a or b is $\mathbf{0}$ and U or W is $\mathbf{0}$.

A pushdown automaton is *deterministic* if it has no pair of incompatible transitions.

Let $u \in A^*$ and let \mathcal{M} be a pushdown automaton. Then a *u -configuration for \mathcal{M}* is a triple $\Delta = (k, q_i, \alpha)$, where $1 \leq k \leq |u| + 1$, q_i is a state of \mathcal{M} , and $\alpha \in \Omega^*$. [Intuitively, the u -configuration (k, q_i, α) stands for the situation in which u is written on \mathcal{M} ’s tape, \mathcal{M} is scanning the k th symbol of u —or, if $k = |u| + 1$, has completed scanning u —and α is the string of symbols on the pushdown stack.] We speak of q_i as the *state at configuration Δ* and of α as the *stack contents at configuration Δ* . If $\alpha = \mathbf{0}$, we say the *stack is empty at Δ* . For a pair of u -configurations, we write

$$u: (k, q_i, \alpha) \vdash_{\mathcal{M}} (l, q_j, \beta)$$

if \mathcal{M} contains a transition $q_i a U : V q_j$, where $\alpha = U\gamma$, $\beta = V\gamma$ for some $\gamma \in \Omega^*$, and either

1. $l = k$ and $a = \mathbf{0}$, or
2. $l = k + 1$ and the k th symbol of u is a .

Note that the equation $\alpha = U\gamma$ is to be read simply $\alpha = \gamma$ in case $U = \mathbf{0}$; likewise for $\beta = V\gamma$.

A sequence $\Delta_1, \Delta_2, \dots, \Delta_m$ of u -configurations is called a *u -computation by \mathcal{M}* if

1. $\Delta_1 = (1, q, \mathbf{0})$ for some $q \in Q$,
2. $\Delta_m = (|u| + 1, p, \gamma)$ for some $p \in Q$ and $\gamma \in \Omega^*$, and
3. $u: \Delta_i \vdash_{\mathcal{M}} \Delta_{i+1}$ for $1 \leq i < m$.

This u -computation is called *accepting* if the state at Δ_1 is the initial state q_1 , the state p at Δ_m is in F , and the stack at Δ_m is empty. We say that \mathcal{M} *accepts* the string $u \in A^*$ if there is an accepting u -computation by \mathcal{M} . We write $L(\mathcal{M})$ for the set of strings accepted by \mathcal{M} , and we call $L(\mathcal{M})$ the *language accepted by \mathcal{M}* .

Acceptance can alternatively be defined either by requiring only that the state at Δ_m is in F or only that $\gamma = 0$. It is not difficult to prove that the class of languages accepted by pushdown automata is not changed by either of these alternatives. (See Exercise 8.)

A few examples should provide readers with some intuition for working with pushdown automata.

Example \mathcal{M}_1 Tape alphabet = $\{a, b\}$, pushdown alphabet = $\{A\}$, $Q = \{q_1, q_2\}$, $F = \{q_2\}$. The transitions are

$$\begin{aligned} q_1 a \mathbf{0} &: A q_1 \\ q_1 b A &: \mathbf{0} q_2 \\ q_2 b A &: \mathbf{0} q_2. \end{aligned}$$

The reader should verify that $L(\mathcal{M}_1) = \{a^{[n]}b^{[n]} \mid n > 0\}$.

Example \mathcal{M}_2 Tape alphabet = $\{a, b, c\}$, pushdown alphabet = $\{A, B\}$, $Q = \{q_1, q_2\}$, $F = \{q_2\}$. The transitions are

$$\begin{aligned} q_1 a \mathbf{0} &: A q_1 \\ q_1 b \mathbf{0} &: B q_1 \\ q_1 c \mathbf{0} &: \mathbf{0} q_2 \\ q_2 a A &: \mathbf{0} q_2 \\ q_2 b B &: \mathbf{0} q_2. \end{aligned}$$

Here, $L(\mathcal{M}_2) = \{ucu^R \mid u \in \{a, b\}^*\}$.

Example \mathcal{M}_3 Tape alphabet = $\{a, b\}$, pushdown alphabet = $\{A, B\}$, $Q = \{q_1, q_2\}$, $F = \{q_2\}$,

$$\begin{aligned} q_1 a \mathbf{0} &: A q_1 \\ q_1 b \mathbf{0} &: B q_1 \\ q_1 a A &: \mathbf{0} q_2 \\ q_1 b B &: \mathbf{0} q_2 \\ q_2 a A &: \mathbf{0} q_2 \\ q_2 b B &: \mathbf{0} q_2. \end{aligned}$$

In this case, $L(\mathcal{M}_3) = \{uu^R \mid u \in \{a, b\}^*, u \neq 0\}$. Note that while $\mathcal{M}_1, \mathcal{M}_2$ are deterministic, \mathcal{M}_3 is a nondeterministic pushdown automaton. Does there exist a deterministic pushdown automaton that accepts $L(\mathcal{M}_3)$? Why not?

$L(\mathcal{M}_1)$, $L(\mathcal{M}_2)$, and $L(\mathcal{M}_3)$ are all context-free languages. We begin our investigation of the relationship between context-free languages and pushdown automata with the following theorem.

Theorem 8.2. Let Γ be a Chomsky normal form grammar with separator Γ_s . Then there is a deterministic pushdown automaton \mathcal{M} such that $L(\mathcal{M}) = L(\Gamma_s)$.

Proof. Let T be the set of terminals of Γ . By Theorem 7.3, for suitable n ,

$$L(\Gamma_s) = R \cap \text{PAR}_n(T),$$

where R is a regular language. Let $P = \{\langle , \rangle_i \mid i = 1, 2, \dots, n\}$. Let \mathcal{M}_0 be a dfa with alphabet $T \cup P$ that accepts R . Let $Q = \{q_1, \dots, q_m\}$ be the states of \mathcal{M}_0 , q_1 the initial state, $F \subseteq Q$ the accepting states, and δ the transition function. We construct a pushdown automaton \mathcal{M} with tape alphabet $T \cup P$ and the same states, initial state, and accepting states as \mathcal{M}_0 . \mathcal{M} is to have the pushdown alphabet $\Omega = \{J_1, \dots, J_n\}$. The transitions of \mathcal{M} are as follows for all $q \in Q$:

- a. for each $a \in T$, $qa0 : 0p$, where $p = \delta(q, a)$;
- b. for $i = 1, 2, \dots, n$, $q_i0 : J_i p_i$, where $p_i = \delta(q_i, \langle)_i \rangle$;
- c. for $i = 1, 2, \dots, n$, $q_i J_i : 0\bar{p}_i$, where $\bar{p}_i = \delta(q_i, \rangle_i \rangle)$.

Since the second entry in these transitions is never 0 , we see that for any $u \in (T \cup P)^*$, a u -computation must be of length $|u| + 1$. It is also clear that no two of the transitions in a–c are incompatible; thus, \mathcal{M} is deterministic.

Now, let $u \in L(\Gamma_s)$, $u = c_1 c_2 \cdots c_K$, where $c_1, c_2, \dots, c_K \in (T \cup P)$. Since $u \in R$, the dfa \mathcal{M}_0 accepts u . Thus, there is a sequence $p_1, p_2, \dots, p_{K+1} \in Q$ such that $p_1 = q_1$, $p_{K+1} \in F$, and $\delta(p_i, c_i) = p_{i+1}$, $i = 1, 2, \dots, K$. Since $u \in \text{PAR}_n(T)$, by Theorem 8.1, u is balanced, so that $\gamma_j(u)$ is defined for $j = 1, 2, \dots, K + 1$ and $\gamma_{K+1}(u) = 0$. We let

$$\Delta_j = (j, p_j, \gamma_j(u)), \quad j = 1, 2, \dots, K + 1.$$

To see that the sequence $\Delta_1, \Delta_2, \dots, \Delta_{K+1}$ is an accepting u -computation by \mathcal{M} , it remains only to check that

$$u : \Delta_j \vdash_{\mathcal{M}} \Delta_{j+1}, \quad j = 1, 2, \dots, K.$$

But this is clear from the definition of $\gamma_j(u)$.

Conversely, let \mathcal{M} accept $u = c_1 c_2 \cdots c_K$. Thus, let $\Delta_1, \Delta_2, \dots, \Delta_{K+1}$ be an accepting u -computation by \mathcal{M} . Let

$$\Delta_j = (j, p_j, \gamma_j), \quad j = 1, 2, \dots, K + 1.$$

Since

$$u: \Delta_j \vdash_{\mathcal{M}} \Delta_{j+1}, \quad j = 1, 2, \dots, K,$$

and $\gamma_1 = 0$, we see that γ_j satisfies the defining recursion for $\gamma_j(u)$ and hence, $\gamma_j = \gamma_j(u)$ for $j = 1, 2, \dots, K + 1$. Since $\gamma_{K+1} = 0$, u is balanced and hence $u \in \text{PAR}_n(T)$. Finally, we have $p_1 = q_1$, $p_{K+1} \in F$, and $\delta(p_j, c_j) = p_{j+1}$. Therefore the dfa \mathcal{M}_0 accepts u , and $u \in R$. ■

We call a pushdown automaton *atomic* (whether or not it is deterministic) if all of its transitions are of one of the forms

- i. $pa0: 0q$,
- ii. $p0U: 0q$,
- iii. $p00: Vq$.

Thus, at each step in a computation an atomic pushdown automaton can read the tape and move right, or pop a symbol off the stack or push a symbol on the stack. But, unlike pushdown automata in general, it cannot perform more than one of these actions in a single step.

Let \mathcal{M} be a given *atomic* pushdown automaton with tape alphabet T and pushdown alphabet $\Omega = \{J_1, J_2, \dots, J_n\}$. We set

$$P = \left\{ \left(\begin{smallmatrix} i \\ i \end{smallmatrix} \right) \mid i = 1, 2, \dots, n \right\}$$

and show how to use the “brackets” to define a kind of “record” of a computation by \mathcal{M} . Let $\Delta_1, \Delta_2, \dots, \Delta_m$ be a (not necessarily accepting) v -computation by \mathcal{M} , where $v = c_1 c_2 \cdots c_K$ and $c_k \in T$, $k = 1, 2, \dots, K$, and where $\Delta_i = (l_i, p_i, \gamma_i)$, $i = 1, 2, \dots, m$. We set

$$w_1 = 0$$

$$w_{i+1} = \begin{cases} w_i c_{l_i} & \text{if } \gamma_{i+1} = \gamma_i \\ w_{ij}^{\left(\begin{smallmatrix} i \\ i \end{smallmatrix} \right)} & \text{if } \gamma_{i+1} = J_j \gamma_i \\ w_{ij}^{\left(\begin{smallmatrix} i \\ i+1 \end{smallmatrix} \right)} & \text{if } \gamma_i = J_j \gamma_{i+1} \end{cases} \quad 1 \leq i < m.$$

[Note that $\gamma_{i+1} = \gamma_i$ is equivalent to $l_{i+1} = l_i + 1$ and is the case when a transition of form i is used in getting from Δ_i to Δ_{i+1} ; the remaining two cases occur when transitions of the form iii or ii, respectively, are used.] Now let $w = w_m$, so that $\text{Er}_P(w) = v$ and $m = |w| + 1$. This word w is called the *record* of the given v -computation $\Delta_1, \dots, \Delta_m$ by \mathcal{M} . From w we can read off not only the word v but also the sequence of “pushes” and “pops” as they occur. In particular, $w_i, 1 < i \leq m$, indicates how \mathcal{M} goes from Δ_{i-1} to Δ_i .

Now we want to modify the pushdown automaton \mathcal{M} of Theorem 8.2 so that it will accept $L(\Gamma)$ instead of $L(\Gamma_s)$. In doing so we will have to give up determinism. The intuitive idea is to use nondeterminism by permitting our modified pushdown automaton to “guess” the location of the “brackets” $_{i, i}^{(,)}$. Thus, continuing to use the notation of the proof of Theorem 8.2, we define a pushdown automaton $\bar{\mathcal{M}}$ with the same states, initial state, accepting states, and pushdown alphabet as \mathcal{M} . However, the tape alphabet of $\bar{\mathcal{M}}$ will be T (rather than $T \cup P$). The transitions of $\bar{\mathcal{M}}$ are, for all $q \in Q$:

- for each $a \in T$, $qa0: 0p$, where $p = \delta(q, a)$ [i.e., the same as the transitions a of \mathcal{M}];
- for $i = 1, 2, \dots, n$, $q00: J_i p_i$, where $p_i = \delta(q, {}_i^{\langle})$;
- for $i = 1, 2, \dots, n$, $q0J_i: 0\bar{p}_i$ where $\bar{p}_i = \delta(q, {}_i^{\rangle})$.

Depending on the transition function δ , $\bar{\mathcal{M}}$ can certainly be nondeterministic. We shall prove that $L(\bar{\mathcal{M}}) = L(\Gamma)$. Note that $\bar{\mathcal{M}}$ is atomic (although \mathcal{M} is not).

First, let $v \in L(\Gamma)$. Then, since $Er_p(L(\Gamma_s)) = L(\Gamma)$, there is a word $w \in L(\Gamma_s)$ such that $Er_p(w) = v$. By Theorem 8.2, $w \in L(\mathcal{M})$. Let $\Delta_1, \Delta_2, \dots, \Delta_m$ be an accepting w -computation by \mathcal{M} (where in fact $m = |w| + 1$). Let

$$\Delta_i = (i, p_i, \gamma_i), \quad i = 1, 2, \dots, m.$$

Let $n_i = 1$ if $w: \Delta_i \vdash_{\mathcal{M}} \Delta_{i+1}$ via a transition belonging to group a; otherwise $n_i = 0, 1 \leq i < m$. Let

$$l_1 = 1,$$

$$l_{i+1} = l_i + n_i, \quad 1 \leq i < m.$$

Finally let

$$\bar{\Delta}_i = (l_i, p_i, \gamma_i), \quad i = 1, 2, \dots, m.$$

Then, as is easily checked,

$$v: \bar{\Delta}_i \vdash_{\bar{\mathcal{M}}} \bar{\Delta}_{i+1}, \quad 1 \leq i < m.$$

Since $\bar{\Delta}_m = (|v| + 1, q, 0)$ with $q \in F$, we have $v \in L(\bar{\mathcal{M}})$.

Conversely, let $v \in L(\bar{\mathcal{M}})$. Let $\bar{\Delta}_1, \bar{\Delta}_2, \dots, \bar{\Delta}_m$ be an accepting v -computation by $\bar{\mathcal{M}}$, where we may write

$$\bar{\Delta}_i = (l_i, p_i, \gamma_i), \quad i = 1, 2, \dots, m.$$

Using the fact that $\bar{\mathcal{M}}$ is atomic, we can let w be the *record* of this computation in the sense defined earlier so that $\text{Er}_P(w) = v$ and $m = |w| + 1$. We write

$$\Delta_i = (i, p_i, \gamma_i), \quad i = 1, 2, \dots, m,$$

and easily observe that

$$w: \Delta_i \vdash_{\bar{\mathcal{M}}} \Delta_{i+1}, \quad i = 1, 2, \dots, m.$$

[In effect, whenever $\bar{\mathcal{M}}$ pushes J_i onto its stack, J_i is inserted into w ; and whenever $\bar{\mathcal{M}}$ pops J_i , J_i is inserted into w . This makes the transitions b, c of \mathcal{M} behave on w just the way the corresponding transitions of $\bar{\mathcal{M}}$ behave on v .] Since $p_m \in F$ and $\gamma_m = 0$, $\Delta_1, \Delta_2, \dots, \Delta_m$ is an accepting w -computation by $\bar{\mathcal{M}}$. Thus, by Theorem 8.2, $w \in L(\Gamma_s)$. Hence $v \in L(\Gamma)$.

We have shown that $L(\Gamma) = L(\bar{\mathcal{M}})$. Hence we have proved

Theorem 8.3. Let Γ be a Chomsky normal form grammar. Then there is a pushdown automaton $\bar{\mathcal{M}}$ such that $L(\bar{\mathcal{M}}) = L(\Gamma)$.

Now let L be any context-free language. By Theorems 1.2 and 3.1 there is a Chomsky normal form grammar Γ such that $L = L(\Gamma)$ or $L(\Gamma) \cup \{0\}$. In the former case, we have shown how to obtain a pushdown automaton $\bar{\mathcal{M}}$ such that $L = L(\bar{\mathcal{M}})$. For the latter case we first modify the dfa \mathcal{M}_0 used in the *proof of Theorem 8.2 so that it is nonrestarting*. We know that this can be done without changing the regular language that \mathcal{M}_0 accepts by Theorem 4.1 in Chapter 9. By carrying out the construction of a pushdown automaton $\bar{\mathcal{M}}$ for which $L(\bar{\mathcal{M}}) = L(\Gamma)$ using the modified version of \mathcal{M}_0 , $\bar{\mathcal{M}}$ will have the property that none of its transitions has q_1 as its final symbol. That is, $\bar{\mathcal{M}}$ will never return to its initial state. Thus, if we define \mathcal{M}' to be exactly like $\bar{\mathcal{M}}$ except for having as its set of accepting states

$$F' = F \cup \{q_1\},$$

we see that $L(\mathcal{M}') = L(\bar{\mathcal{M}}) \cup \{0\} = L(\Gamma) \cup \{0\}$. Thus we have proved

Theorem 8.4. For every context-free language L , there is a pushdown automaton $\bar{\mathcal{M}}$ such that $L = L(\bar{\mathcal{M}})$.

We will end this section by proving the converse of this result. Thus we must begin with a pushdown automaton and prove that the language it accepts is context-free. As a first step toward this goal, we will show that we can limit our considerations to atomic pushdown automata.

Theorem 8.5. Let \mathcal{M} be a pushdown automaton. Then there is an atomic pushdown automaton $\bar{\mathcal{M}}$ such that $L(\mathcal{M}) = L(\bar{\mathcal{M}})$.

Proof. For each transition

$$paU: Vq$$

of \mathcal{M} for which $a, U, V \neq \mathbf{0}$, we introduce two new states r, s and let $\bar{\mathcal{M}}$ have the transitions

$$\begin{aligned} p\mathbf{0}: \mathbf{0}r, \\ r\mathbf{0}U: \mathbf{0}s, \\ s\mathbf{0}V: Vq. \end{aligned}$$

If exactly one of a, U, V is $\mathbf{0}$, then only two transitions are needed for $\bar{\mathcal{M}}$. Finally, for each transition $p\mathbf{00}: \mathbf{0}q$, we introduce a new state t and replace $p\mathbf{00}: \mathbf{0}q$ with the transitions $p\mathbf{00}: Jt$, $t\mathbf{0}J: \mathbf{0}q$, where J is an arbitrary symbol of the pushdown alphabet (or a new symbol if the pushdown alphabet of \mathcal{M} is empty). Otherwise, $\bar{\mathcal{M}}$ is exactly like \mathcal{M} . Clearly, $L(\bar{\mathcal{M}}) = L(\mathcal{M})$. ■

Theorem 8.6. For every pushdown automaton \mathcal{M} , $L(\mathcal{M})$ is a context-free language.

Proof. Without loss of generality, by using Theorem 8.5 we can assume that \mathcal{M} is atomic. Let \mathcal{M} have states $Q = \{q_1, \dots, q_m\}$, initial state q_1 , final states F , tape alphabet T , and pushdown alphabet $\Omega = \{J_1, \dots, J_n\}$. Let $P = \{\langle \cdot, \cdot \rangle \mid i = 1, \dots, n\}$. Let $L \subseteq (T \cup P)^*$ consist of the records of every accepting u -computation by \mathcal{M} , and let $R = L(\mathcal{M}_0)$, where \mathcal{M}_0 is the ndfa with alphabet $T \cup P$, the same states, initial state, and accepting states as \mathcal{M} , and transition function δ defined as follows. For each $q \in Q$,

$$\delta(q, a) = \{p \in Q \mid \mathcal{M} \text{ has the transition } qa\mathbf{0}: \mathbf{0}p\} \text{ for } a \in T,$$

$$\delta(q, \langle \cdot, \cdot \rangle) = \{p \in Q \mid \mathcal{M} \text{ has the transition } q\mathbf{00}: J_i p\}, i = 1, \dots, n,$$

$$\delta(q, \langle \cdot, \cdot \rangle) = \{p \in Q \mid \mathcal{M} \text{ has the transition } q\mathbf{0}J_i: \mathbf{0}p\}, i = 1, \dots, n.$$

Let $w \in L$ be the record of an accepting u -computation $\Delta_1, \dots, \Delta_m$, where $\Delta_i = (l_i, p_i, \gamma_i)$, $i = 1, \dots, m$. An easy induction on i shows that $p_i \in \delta^*(q_1, w_i)$, $i = 1, \dots, m$, so, in particular, $p_m \in \delta^*(q_1, w)$, which implies $w \in R$, since p_m must be an accepting state. Moreover, another easy induction on i shows that $\gamma_i(w) = \gamma_i$, $i = 1, \dots, m$, which implies that $\gamma_i(w)$ is defined for $1 \leq i \leq |w| + 1$ and $\gamma_{|w|+1}(w) = \gamma_{|w|+1} = 0$ (since $\Delta_1, \dots, \Delta_m$ is accepting), i.e., w is balanced. Therefore, by Theorem 8.1, $w \in R \cap \text{PAR}_n(T)$, and so $L \subseteq R \cap \text{PAR}_n(T)$.

On the other hand, let $w = c_1 \cdots c_r$ be a balanced word in R , i.e., $w \in R \cap \text{PAR}_n(T)$, let $u = d_1 \cdots d_s$ be $\text{Er}_P(w)$, and let p_1, \dots, p_{r+1} be

some sequence of states such that $p_1 = q_1$, $p_{r+1} \in F$, and $p_{i+1} \in \delta(p_i, c_i)$ for $i = 1, \dots, r$. We claim that

$$(l_1, p_1, \gamma_1(w)), (l_2, p_2, \gamma_2(w)), \dots, (l_{r+1}, p_{r+1}, \gamma_{r+1}(w)), \quad (8.1)$$

where

$$l_1 = 1$$

$$l_{i+1} = \begin{cases} l_i + 1 & \text{if } c_i \in T \\ l_i & \text{otherwise,} \end{cases}$$

is an accepting u -computation by \mathcal{M} and that w is its record. Clearly, we have $(l_1, p_1, \gamma_1(w)) = (1, q_1, 0)$, $l_{r+1} = |u| + 1$, $p_{r+1} \in F$, and $\gamma_{r+1}(w) = 0$ (since w is balanced), so we just need to show that

$$u: (l_i, p_i, \gamma_i(w)) \vdash_{\mathcal{M}} (l_{i+1}, p_{i+1}, \gamma_{i+1}(w)) \quad (8.2)$$

for $i = 1, \dots, r$. For arbitrary $i = 1, \dots, r$, if $\gamma_{i+1}(w) = \gamma_i(w)$, then $c_i \in T$, so $p_{i+1} \in \delta(p_i, c_i)$, and \mathcal{M} has the transition $p_i c_i \mathbf{0} : \mathbf{0} p_{i+1}$. Now, a simple induction on i shows that $\text{Er}_P(c_1 \cdots c_{i-1}) = d_1 \cdots d_{l_{i-1}}$, $i = 1, \dots, r+1$ (where $c_1 \cdots c_0$ represents 0), from which we can show

$$\text{if } c_i \in T \text{ then } d_{l_i} = c_i, \quad i = 1, \dots, r.$$

In particular, for any $i = 1, \dots, r$, if $c_i \in T$, then

$$\text{Er}_P(c_1 \cdots c_i) = \text{Er}_P(c_1 \cdots c_{i-1}) c_i = d_1 \cdots d_{l_{i-1}} c_i,$$

so c_i must be d_{l_i} since c_i is not deleted when Er_P is applied to w . Therefore, \mathcal{M} has the transition $p_i d_{l_i} \mathbf{0} : \mathbf{0} p_{i+1}$ and $l_{i+1} = l_i + 1$, so (8.2) is satisfied. If, instead, $\gamma_{i+1}(w) = J_j \gamma_i(w)$ for some $j = 1, \dots, n$, then $c_i = \underbrace{\}_{j}}$, so $p_{i+1} \in \delta(p_i, \underbrace{\}_{j})$ and \mathcal{M} has the transition $p_i \mathbf{0} \mathbf{0} : J_j p_{i+1}$. Moreover, $c_i \notin T$, so $l_{i+1} = l_i$, and (8.2) is satisfied in this case as well. Finally, if $\gamma_i(w) = J_j \gamma_{i+1}(w)$ for some $j = 1, \dots, n$, then $c_i = \underbrace{\}_{j}}$, so $p_{i+1} \in \delta(p_i, \underbrace{\}_{j})$ and \mathcal{M} has the transition $p_i \mathbf{0} J_j : \mathbf{0} p_{i+1}$. Moreover, $l_{i+1} = l_i$, so again (8.2) is satisfied. Therefore, (8.1) is an accepting u -computation by \mathcal{M} . If we set $w_i = c_1 \cdots c_{i-1}$, $i = 1, \dots, r+1$, then an induction on i shows that w is indeed the record of (8.1), so $w \in L$, and we have $R \cap \text{PAR}_n(T) \subseteq L$. Therefore, $L = R \cap \text{PAR}_n(T)$, and

$$L(\mathcal{M}) = \text{Er}_P(R \cap \text{PAR}_n(T)).$$

Finally, by Theorems 5.4 and 7.1 and Corollary 5.6, $L(\mathcal{M})$ is context-free. ■

Exercises

1. Let T be an alphabet, $P = \{i^l i^r \mid i = 1, \dots, n\}$, $w = a_1 \cdots a_m \in \text{PAR}_n(T)$. Integers j, k , where $1 \leq j < k \leq m$, are *matched* in w if $w = a_1 \cdots a_{j-1} i^l a_{j+1} \cdots a_{k-1} i^r a_{k+1} \cdots a_m$, for some $1 \leq i \leq n$, and if $a_{j+1} \cdots a_{k-1}$ is balanced. Let Γ be a Chomsky normal form grammar.
 - (a) Let $w = \{x \in L(\Gamma_s)\}$. Prove that there is exactly one k , $1 < k \leq |w|$, such that 1 and k are matched in w .
 - (b) Show that Γ_s is unambiguous. [See Section 6 for the definition of *ambiguous grammars*.]
2. (a) For pushdown automaton \mathcal{M}_1 in the examples, give the accepting u_1 -computation for $u_1 = aabb$.
- (b) For pushdown automaton \mathcal{M}_2 in the examples and $u_2 = abcbba$, give the longest sequence $\Delta_1 = (1, q_1, 0), \Delta_2, \dots, \Delta_m$ of u_2 -configurations that satisfies condition 3 in the definition of u -computations.
- (c) For pushdown automaton \mathcal{M}_3 in the examples, give all possible u_3 -computations, accepting or not, for $u_3 = aaaa$.
3. For each of the following languages L , give a pushdown automaton that accepts L .
 - (a) $\{a^{[n]}b^{[2n]} \mid n > 0\}$.
 - (b) $\{a^{[n]}b^{[m]} \mid 0 < n \leq m\}$.
 - (c) $\{a^{[n]}b^{[m]} \mid n \neq m\}$.
 - (d) $\{a^{[n]}b^{[m]}a^{[n]} \mid m, n > 0\} \cup \{a^{[n]}c^{[n]} \mid n > 0\}$.
4. Let \mathcal{M} be the pushdown automaton with $Q = \{q_1\}$, $F = \{q_1\}$, and transitions

$$\begin{aligned} q_1a\mathbf{0}: Aq_1 &\quad q_1aB: \mathbf{0}q_1 \\ q_1b\mathbf{0}: Bq_1 &\quad q_1bA: \mathbf{0}q_1. \end{aligned}$$

What is $L(\mathcal{M})$?

5. Let \mathcal{M} be the pushdown automaton with $Q = \{q_1, q_2, q_3, q_4, q_5\}$, $F = \{q_5\}$, and transitions

$$\begin{aligned} q_1\mathbf{00}: Zq_2 &\quad q_3bA: \mathbf{0}q_3 && q_4a\mathbf{0}: \mathbf{0}q_4 \\ q_2a\mathbf{0}: Aq_2 &\quad q_3bZ: \mathbf{0}q_4 && q_4b\mathbf{0}: \mathbf{0}q_4 \\ q_2bA:\mathbf{0}q_3 &\quad q_3a\mathbf{0}: \mathbf{0}q_4 && q_4\mathbf{0}A:\mathbf{0}q_4 \\ q_2bZ:\mathbf{0}q_4 &\quad q_3\mathbf{0}Z: \mathbf{0}q_5 && q_4\mathbf{0}Z:\mathbf{0}q_4. \end{aligned}$$

- (a) What is $L(\mathcal{M})$?
- (b) Prove that for every $u \in \{a, b\}^*$, there is a u -computation by \mathcal{M} .
6. Show that every regular language accepted by a (deterministic) finite automaton with n states is accepted by a (deterministic) pushdown automaton with n states and an empty pushdown alphabet.
7. Show that every regular language R is accepted by a pushdown automaton with at most two states, and if $0 \in R$ then R is accepted by a pushdown automaton with one state.
8. Let \mathcal{M} be a pushdown automaton with initial state q_1 , accepting states F , and tape alphabet A , let $u \in A^*$, and let $\Delta_1 = (1, q_1, 0), \dots, \Delta_m = (|u| + 1, p, \gamma)$ be a u -computation by \mathcal{M} . We say that \mathcal{M} accepts u by final state if $p \in F$, and that \mathcal{M} accepts u by empty stack if $\gamma = 0$. $T(\mathcal{M}) = \{u \in A^* \mid \mathcal{M} \text{ accepts } u \text{ by final state}\}$, and $N(\mathcal{M}) = \{u \in A^* \mid \mathcal{M} \text{ accepts } u \text{ by empty stack}\}$.
- (a) Let $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$ be the pushdown automata from the examples. Give $T(\mathcal{M}_i), N(\mathcal{M}_i)$, $i = 1, 2, 3$.
 - (b) Prove that a language L is context-free if and only if $L = T(\mathcal{M})$ for some pushdown automaton \mathcal{M} .
 - (c) Prove that a language L is context-free if and only if $L = N(\mathcal{M})$ for some pushdown automaton \mathcal{M} .
9. Let \mathcal{M} be a pushdown automaton with tape alphabet A , and let $u \in A^*$. An infinite sequence $\Delta_1, \Delta_2, \dots$ of u -configurations for \mathcal{M} is an *infinite u -computation* by \mathcal{M} if for some n and some x such that $u = xy$ for some y , each finite sequence $\Delta_1, \dots, \Delta_n, \dots, \Delta_{n+m}$, $m \geq 0$, is an x -computation by \mathcal{M} . It is an *accepting* infinite u -computation if $\Delta_1, \dots, \Delta_k$ is an accepting u -computation by \mathcal{M} for some k .
- (a) Give a pushdown automaton \mathcal{M}_1 and word u_1 such that there is a nonaccepting infinite u -computation by \mathcal{M}_1 .
 - (b) Give a pushdown automaton \mathcal{M}_2 and word u_2 such that there is an accepting infinite u_2 -computation $(l_1, p_1, \gamma_1), (l_2, p_2, \gamma_2), \dots$ by \mathcal{M}_2 where, for some k , p_l is an accepting state for all $l \geq k$.
 - (c) Give a pushdown automaton \mathcal{M}_3 and word u_3 such that there is an accepting infinite u_3 -computation $(l_1, p_1, \gamma_1), (l_2, p_2, \gamma_2), \dots$ by \mathcal{M}_3 where there is no k such that p_l is an accepting state for all $l \geq k$.
10. Give the incompatible pairs among the following transitions. In each case, give the condition(s) 1, 2, 3, or 4 by which the pair is incompatible.

$$\begin{array}{lll}
 q_1 a J_1 : 0q_1 & q_1 a \mathbf{0} : J_2 q_1 & q_1 b J_2 : J_1 q_1 \\
 q_1 b J_1 : 0q_1 & q_1 \mathbf{0} J_1 : 0q_1 & q_1 \mathbf{00} : J_1 q_1 \\
 q_1 a J_1 : 0q_2 & &
 \end{array}$$

11. Let $T = \{a, b\}$, $P = \{\langle \begin{smallmatrix} (\\) \\ , \end{smallmatrix} \rangle, \langle \begin{smallmatrix} [\\] \\ , \end{smallmatrix} \rangle\}$, $\Omega = \{J_1, J_2\}$. We will write $(,)$, $[,]$ for $\langle \begin{smallmatrix} (\\) \\ , \end{smallmatrix} \rangle, \langle \begin{smallmatrix} [\\] \\ , \end{smallmatrix} \rangle$, respectively. Give $\gamma_i(w)$, $1 \leq i \leq |w| + 1$, for each of the following.
- (a) $w = a(b[ba]a)b[a]$.
 - (b) $w = (ab[ab])a$.
 - (c) $w = a[b]]a$.
 - (d) $w = (a[b])a$.
12. Let Γ be the grammar with productions $S \rightarrow SS$, $S \rightarrow a$.
- (a) Use the construction in the proof of Theorem 8.2 to give a deterministic pushdown automaton that accepts $L(\Gamma_s)$.
 - (b) Use the construction in the proof of Theorem 8.3 to give a pushdown automaton that accepts $L(\Gamma)$.
13. (a) For pushdown automata $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$ in the examples, use the construction in the proof of Theorem 8.5 to give atomic pushdown automata $\bar{\mathcal{M}}_1, \bar{\mathcal{M}}_2, \bar{\mathcal{M}}_3$.
- (b) Answer Exercise 2 for $\bar{\mathcal{M}}_1, \bar{\mathcal{M}}_2, \bar{\mathcal{M}}_3$.
14. Let \mathcal{M} be the pushdown automaton with $Q = \{q_1, q_2\}$, initial state q_1 , $F = \{q_2\}$, tape alphabet $\{a, b\}$, pushdown alphabet $\{A\}$, and transitions
- $$\begin{array}{llll}
 q_1 a \mathbf{0} : 0q_1 & q_1 \mathbf{00} : Aq_1 & q_2 a \mathbf{0} : 0q_1 & q_2 \mathbf{0} A : 0q_2 \\
 q_1 b \mathbf{0} : 0q_2 & q_1 \mathbf{0} A : 0q_1 & q_2 b \mathbf{0} : 0q_2 & q_2 \mathbf{00} : Aq_2 .
 \end{array}$$
- Use the constructions in Theorems 8.6 and 5.4 to give a context-free grammar Γ such that $L(\Gamma) = L(\mathcal{M})$.
15. Let us call a *generalized pushdown automaton* a device that functions just like a pushdown automaton except that it can write any finite sequence of symbols on the stack in a single step. Show that, for every generalized pushdown automaton \mathcal{M} , there is a pushdown automaton $\bar{\mathcal{M}}$ such that $L(\mathcal{M}) = L(\bar{\mathcal{M}})$.
- 16.* Let

$$P = \left\{ \boxed{\frac{u_i}{sv_i}} \mid i = 1, 2, \dots, k \right\}$$

be a set of dominoes on the alphabet A . Let $B = \{c_1, \dots, c_k\}$ be an alphabet such that $A \cap B = \emptyset$. Let $c \notin A \cup B$. Let

$$R = \{ycy^R \mid y \in A^*B^*\},$$

$$L_1 = \{u_{i_1}u_{i_2} \cdots u_{i_n}c_{i_n}c_{i_{n-1}} \cdots c_{i_2}c_{i_1}\},$$

$$L_2 = \{v_{i_1}v_{i_2} \cdots v_{i_n}c_{i_n}c_{i_{n-1}} \cdots c_{i_2}c_{i_1}\},$$

$$S_p = \{ycz^R \mid y \in L_1, z \in L_2\}.$$

Recall that by Theorem 6.5, the Post correspondence problem P has a solution if and only if $L_1 \cap L_2 \neq \emptyset$.

- (a) Show that the Post correspondence problem P has *no* solution if and only if $R \cap S_p = \emptyset$.
- (b) Show that $(A \cup B \cup \{c\})^* - R$ and $(A \cup B \cup \{c\})^* - S_p$ are both context-free. [Hint: Construct pushdown automata.]
- (c) From (a) and (b) show how to conclude that there is no algorithm that can determine for a given context-free grammar Γ with terminals T whether $L(\Gamma) \cup \{0\} = T^*$.
- (d) Now show that there is no algorithm that can determine for a given context-free grammar Γ_1 and regular grammar Γ_2 whether

$$(i) \quad L(\Gamma_1) = L(\Gamma_2),$$

$$(ii) \quad L(\Gamma_1) \supseteq L(\Gamma_2).$$

- 17.* Let \mathcal{M} be a pushdown automaton with $Q = \{q_1, \dots, q_m\}$, tape alphabet $A = \{a_1, \dots, a_n\}$, and pushdown alphabet $\Omega = \{J_1, \dots, J_l\}$, and let $p, p' \in Q, J, J' \in \Omega$. A sequence $(1, p_1, \gamma_1), \dots, (1, p_k, \gamma_k)$ of 0-configurations for \mathcal{M} is a *reaching sequence by \mathcal{M} from (p, J) to (p', J')* if $p_1 = p$, $\gamma_1 = J$, $p_k = p'$, $\gamma_k = J'\delta$ for some $\delta \in \Omega^*, |\gamma_i| > 0$ for $1 \leq i \leq k$, and $0: (1, p_i, \gamma_i) \vdash_{\mathcal{M}} (1, p_{i+1}, \gamma_{i+1})$ for $1 \leq i < k$. (p, J) is a *looping pair of \mathcal{M}* if there is a reaching sequence by \mathcal{M} from (p, J) to (p, J) .

- (a) Prove that if \mathcal{M} has a u -computation $\Delta_1, \dots, \Delta_k = (|u| + 1, p, J\gamma)$ for some looping pair (p, J) of \mathcal{M} , then \mathcal{M} has an infinite uw -computation for every $w \in A^*$. [See Exercise 9 for the definition of infinite u -computations.]
- (b) Prove that if (p, J) is a looping pair for \mathcal{M} , then there is a reaching sequence $\Delta_1 = (1, p, J), \dots, \Delta_k = (1, p, J\delta)$ by \mathcal{M} from (p, J) to (p, J) such that $|\delta| \leq lm$ [Hint: Consider the pigeon-hole principle and the proofs of the pumping lemmas.]

- (c) Prove that if (p, J) is a looping pair of \mathcal{M} , then there is a reaching sequence $\Delta_1, \dots, \Delta_k$ by \mathcal{M} from (p, J) to (p, J) with $k \leq m(l + 1)^{lm} + 1$.
- (d) Give an algorithm that will determine, for a pushdown automaton \mathcal{M} and pair (p, J) , whether or not (p, J) is a looping pair of \mathcal{M} .
- (e) Prove that if \mathcal{M} has an infinite u -computation, for some $u \in A^*$, then \mathcal{M} has a looping pair.
- (f) Suppose now that \mathcal{M} is deterministic. Prove that there is a deterministic pushdown automaton \mathcal{M}' such that
 - (i) there is no infinite u -computation by \mathcal{M}' for any $u \in A^*$;
 - (ii) there is a u -computation by \mathcal{M}' for every $u \in A^*$, and
 - (iii) $T(\mathcal{M}') = T(\mathcal{M})$. [See Exercise 8 for the definition of $T(\mathcal{M})$.]
- (g) A language L is a *deterministic context-free language* if $L = T(\mathcal{M})$ for some deterministic pushdown automaton \mathcal{M} . Prove that if $L \subseteq A^*$ is a deterministic context-free language, then $A^* - L$ is also a deterministic context-free language.
- (h) Show that $\{a^{[i]}b^{[j]}c^{[k]} \mid i \neq j \text{ or } j \neq k\}$ is a context-free language which is not deterministic.
- (i) Show that there is an algorithm that can determine for a given deterministic pushdown automaton \mathcal{M} and dfa \mathcal{M}' whether $T(\mathcal{M}) = L(\mathcal{M}')$.

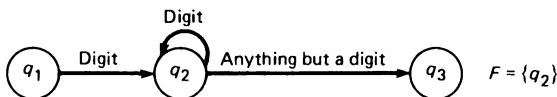
9. Compilers and Formal Languages

A *compiler* is a program that takes as input a program (known as the *source program*) written in a high-level language such as COBOL, FORTRAN, or Pascal and translates it into an equivalent program (known as the *object program*) in a low-level language such as an assembly language or a machine language. Just as in Chapters 2 and 5 we found it easier to write programs with the aid of macros, most programmers find programming in a high-level language faster, easier, and less tedious than in a low-level language. Thus the need for compilers.

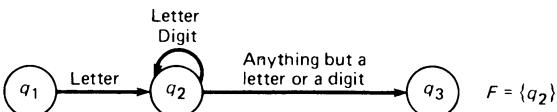
The translation process is divided into a sequence of phases, of which the first two are of particular interest to us. *Lexical analysis*, which is the first phase of the compilation process, consists of dividing the characters of the source program into groups called *tokens*. Tokens are the logical units of an instruction and include keywords such as **IF**, **THEN**, and **DO**,

operators such as + and *, predicates such as >, variable names, labels, constants, and punctuation symbols such as (and ;.

The reason that the lexical analysis phase of compilation is of interest to us is that it represents an application of the theory of finite automata and regular expressions. The lexical analyzer must identify tokens, determine types, and store this information into a *symbol table* for later use. Typically, compiler writers use nondeterministic finite automata to design these token recognizers. For example, the following is an ndfa that recognizes unsigned integer constants.



Similarly, a nondeterministic finite automaton that recognizes variable names might look like this:



We end our brief discussion of lexical analysis by noting that it is not always a simple task to properly determine the division into tokens. For example, in FORTRAN, the statements

DO 10 I = 1.11

and

DO 10 I = 1,11

look very similar but are in fact totally unrelated instructions. The first is an assignment statement that assigns to a variable named DO10I (embedded blanks are ignored in FORTRAN) the value 1.11. The second is a DO loop that indicates that the body is to be performed 11 times. It is not until the “.” or “,” is encountered that the statement type can be determined.

At the completion of the lexical analysis phase of compilation, tokens have been identified, their types determined, and when appropriate, the value entered in the symbol table. At this point, the second phase of compilation, known as *syntactic analysis* or *parsing*, begins. It is in this second phase that context-free grammars play a central role.

For programming languages that are context-free, the parsing problem amounts to determining for a given context-free grammar Γ and word w

1. whether $w \in L(\Gamma)$, and
2. if $w \in L(\Gamma)$, how w could have been generated.

Intuitively, the parsing phase of the compilation process consists of the construction of derivation or parse trees whose leaves are the tokens identified by the lexical analyzer.

Thus, for example, if our grammar included the productions

$$\begin{aligned}
 S &\rightarrow \text{while-statement} \\
 S &\rightarrow \text{assignment-statement} \\
 \text{while-statement} &\rightarrow \mathbf{while} \text{ cond } \mathbf{do} \text{ } S \\
 \text{cond} &\rightarrow \text{cond } \vee \text{ cond} \\
 \text{cond} &\rightarrow \text{rel} \\
 \text{rel} &\rightarrow \text{exp pred exp} \\
 \text{exp} &\rightarrow \text{exp } + \text{ exp} \\
 \text{exp} &\rightarrow \text{var} \\
 \text{exp} &\rightarrow \text{const} \\
 \text{pred} &\rightarrow > \\
 \text{pred} &\rightarrow = \\
 \text{assignment-statement} &\rightarrow \text{var } \leftarrow \text{exp}
 \end{aligned}$$

then the parse tree for the statement

while $x > y \vee z = 2$ **do** $w \leftarrow x + 4$

is given by Fig. 9.1.

The parsing is usually accomplished by simulating the behavior of a pushdown automaton that accepts $L(\Gamma)$ either starting from the root of the tree or the leaves of the tree. In the former case, this is known as *top-down parsing* and in the latter case, *bottom-up parsing*.

Most programming languages are for the most part context-free. (A major exception is the coordination of declarations and uses.) A common technique involves the definition of a superset of the programming language which can be accepted by a deterministic pushdown automaton. This is desirable since there are particularly fast algorithms for parsing grammars associated with deterministic pushdown automata.

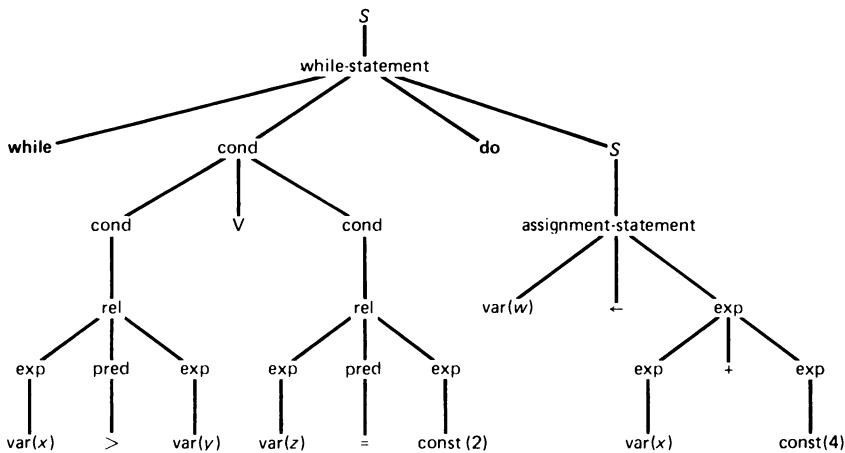


Figure 9.1

Exercise

1. Give a context-free grammar for generating valid Pascal arithmetic expressions over the alphabet $\{a, b, +, -, *, /, \uparrow, (,), \}\cup\{a, b\}^*$, where variable names are elements of $\{a, b\}^*$ of length at least 1. Is the grammar ambiguous? What are the implications of this?

11

Context-Sensitive Languages

1. The Chomsky Hierarchy

We are now going to place our work in the context of Noam Chomsky's hierarchy of grammars and languages. An arbitrary (phrase structure) grammar (recall Chapter 7, Section 5) is called a type 0 grammar. A *context-sensitive grammar* (recall Chapter 7, Section 5) is called a type 1 grammar. A *positive context-free grammar* (recall Chapter 10, Section 1) is called a type 2 grammar, and a *regular grammar* (recall Chapter 10, Section 2) is called a type 3 grammar. The inclusions suggested by the numbering obviously hold: every regular grammar is context-free, and every positive context-free grammar is context-sensitive. (Of course, grammars containing productions of the form $V \rightarrow 0$ cannot be context-sensitive.)

For each type of grammar, there is a corresponding class of languages:

A language L is $\begin{bmatrix} \text{regular} \\ \text{context-free} \\ \text{context-sensitive} \\ r.e. \end{bmatrix}$ or of type $\begin{bmatrix} 3 \\ 2 \\ 1 \\ 0 \end{bmatrix}$

if and only if there is a $\begin{bmatrix} \text{regular} \\ \text{positive context-free} \\ \text{context-sensitive} \\ \text{phrase structure} \end{bmatrix}$ grammar Γ

such that

$$L = L(\Gamma) \quad \text{or} \quad L = L(\Gamma) \cup \{0\}.$$

For regular languages this statement is just Theorem 2.3 in Chapter 10. For context-free languages, it is Theorem 1.2 in Chapter 10. For context-sensitive languages we take it as a definition. For r.e. languages it is Theorem 5.2 in Chapter 7, and the special reference to $\{0\}$ is not needed. We have

Theorem 1.1. Every regular language is context-free. Every context-free language is context-sensitive. Every context-sensitive language is recursive.

Proof. The first two statements follow simply from the corresponding inclusions among the types of grammar. The third follows at once from Theorem 5.4 in Chapter 7. ■

We would like to show that the inclusions of Theorem 1.1 are proper, that is, that none of the four classes mentioned in the theorem is identical to any of the others. We have seen in Theorem 1.1 in Chapter 10, that the language $L = \{a^{[n]}b^{[n]}|n > 0\}$ is context-free but not regular. Similarly, we saw in Theorem 4.2 in Chapter 10 that the language $\{a^{[n]}b^{[n]}c^{[n]}|n > 0\}$ is not context-free, while Exercise 5.4 in Chapter 7 shows that it is context-sensitive. This takes care of the first two inclusions of Theorem 1.1. The following theorem takes care of the remaining one.

Theorem 1.2. There is a recursive language on the alphabet $\{1\}$ that is not context-sensitive.

Proof. We first code each context-sensitive grammar Γ with terminal alphabet $\{1\}$ by a string on the five-letter alphabet $A = \{1, V, b, \rightarrow, /\}$. We do this simply by replacing each variable by a distinct string of the form $Vb^{[j]}$, using the arrow “ \rightarrow ” as usual between the left and right sides of productions, and using the slash “ $/$ ” to separate productions. (Of course, not every string on this alphabet is actually the code for a context-sensitive grammar.) Now, the strings that code context-sensitive grammars may be placed in alphabetic order (or equivalently, in numerical order, regarding each string on A as the base 5 notation for an integer, as in Chapter 5). We let L_i be the context-sensitive language generated by the i th context-sensitive grammar in this enumeration, $i = 1, 2, 3, \dots$. Then we set

$$L = \{1^{[i]} | 1^{[i]} \notin L_i, i \neq 0\}.$$

This is, of course, a typical diagonal construction, and we easily show that

L is not context-sensitive. For, if $L = L_{i_0}$, then

$$\begin{aligned} 1^{[i_0]} \in L &\quad \text{if and only if } 1^{[i_0]} \notin L_{i_0} \\ &\quad \text{if and only if } 1^{[i_0]} \notin L. \end{aligned}$$

To see that L is recursive we note that there is an algorithm which given i will return a context-sensitive grammar Γ_i that generates L_i . Then $1^{[i]}$ can be tested for membership in L_i using the algorithm developed in the proof of Theorem 5.4 in Chapter 7. ■

For each class of languages corresponding to types 0, 1, 2, 3, we are concerned with questions of the following kinds: What can we determine algorithmically about a language from a grammar which generates it? What kinds of device will accept precisely the languages belonging to the class? Under what operations are the classes closed? We have been dealing with these questions for languages of types 0, 2, and 3. Now, we will see what can be said about languages of type 1, i.e., context-sensitive languages. We begin by considering the question of closure under union. We will need the

Lemma. There is an algorithm that will transform a given context-sensitive grammar Γ into a context-sensitive grammar Δ such that the left sides of the productions of Δ contain no terminals and $L(\Gamma) = L(\Delta)$.

Proof. We “disguise” the terminals as variables as in the proof of Theorem 3.1 in Chapter 10, except that now we need to replace the terminals on both the left and right sides of the productions. The resulting grammar, Δ , consists of productions of the form $X_1 \cdots X_m \rightarrow Y_1 \cdots Y_n$, $m \leq n$, and $X_a \rightarrow a$, where $X_1, \dots, X_m, Y_1, \dots, Y_n, X_a$ are variables and a is a terminal. Clearly, $L(\Delta) = L(\Gamma)$. ■

Theorem 1.3. If L_1, L_2 are context-sensitive languages, then so is $L_1 \cup L_2$.

Proof. Assume $L_1 = L(\Gamma_1)$ or $L(\Gamma_1) \cup \{0\}$, $L_2 = L(\Gamma_2)$ or $L(\Gamma_2) \cup \{0\}$, where Γ_1 and Γ_2 are context-sensitive grammars with disjoint sets of variables of the form obtained in the Lemma. We construct Γ from Γ_1 and Γ_2 exactly as in the proof of Theorem 5.1 in Chapter 10, so that Γ is also context-sensitive and $L(\Gamma) = L(\Gamma_1) \cup L(\Gamma_2)$. Clearly, $L_1 \cup L_2 = L(\Gamma)$ or $L(\Gamma) \cup \{0\}$. ■

Exercises

1. Show that $\{w \in \{a, b, c\}^* | w \text{ has an equal number of } a's, b's, c's\}$ is context-sensitive.

2. Let Γ be the grammar with productions

$$\begin{array}{lll}
 S \rightarrow \lambda XY\rho & Aa \rightarrow aA & AY \rightarrow XYa \\
 \lambda X \rightarrow \lambda aA & Ab \rightarrow bA & BY \rightarrow XYb \\
 \lambda X \rightarrow \lambda bB & Ba \rightarrow aB & CY \rightarrow cc \\
 \lambda X \rightarrow \lambda ccC & Bb \rightarrow bB & aX \rightarrow Xa \\
 & Ca \rightarrow aC & bX \rightarrow Xb, \\
 & Cb \rightarrow bC &
 \end{array}$$

where $\mathcal{V} = \{S, X, Y, A, B, C\}$ and $T = \{\lambda, \rho, a, b, c\}$. What is $L(\Gamma)$?

3. Show that $\{ww \mid w \in \{a, b\}^*\}$ is context-sensitive.
4. Apply the construction in the proof of the Lemma to the grammar in Exercise 2.
5. Show that the proof of Theorem 1.3 fails if we do not assume that Γ_1, Γ_2 conform to the conditions of the Lemma.
6. (a) Let Γ be a context-sensitive grammar. Show that there is a context-sensitive grammar Γ' such that $L(\Gamma') = L(\Gamma)$ and such that, for every production $u \rightarrow v$ in Γ , $|u| \leq 2$ and $|v| \leq 2$.
- (b) Prove that a language L is context-sensitive if and only if it is generated by a grammar Γ , with variables \mathcal{V} and terminals T , such that every production in Γ has the form $uVw \rightarrow uvw$, where $u, w \in (\mathcal{V} \cup T)^*$, $V \in \mathcal{V}$, and $v \in (\mathcal{V} \cup T)^* - \{0\}$. [Note: This explains the origin of the term *context-sensitive*.]

2. Linear Bounded Automata

We are now going to deal with the question: which devices accept context-sensitive languages? We define a *linear bounded automaton* on the alphabet $C = \{s_1, s_2, \dots, s_n\}$ to be a *nondeterministic Turing machine* \mathcal{M} on the alphabet $C \cup \{\lambda, \rho\}$ such that the only quadruples \mathcal{M} contains beginning $q\lambda$ or $q\rho$ are of the forms $q\lambda Rp$ and $q\rho Lp$, respectively, such that \mathcal{M} has a *final state*, written \tilde{q} , where no quadruple of \mathcal{M} begins \tilde{q} , and finally such that for every quadruple $qabp$ in \mathcal{M} , we have $b \neq \lambda, \rho$. Thus, when scanning λ , \mathcal{M} can move only right, and when scanning ρ , \mathcal{M} can move only left, and the symbols λ, ρ can never be printed in the course of a computation. Thus, the effect of the additional symbols λ and ρ is simply to prevent the machine from moving beyond the confines of the given string on the tape. Because of this we can code a configuration of \mathcal{M} by a triple $(i, q, \lambda w \rho)$, where $0 \leq i \leq |w| + 1$; i gives the position of the tape-head (i.e., of the scanned square), q is the current state; and $\lambda w \rho$ is the

Table 2.1

Quadruple in \mathcal{M}	Corresponding transition
$q \ a \ b \ p$	$(u , q, uav) \xrightarrow{\mathcal{M}} (u , p, ubv)$
$q \ a \ R \ p$	$(u , q, uav) \xrightarrow{\mathcal{M}} (u + 1, p, uav)$
$q \ a \ L \ p$	$(u , q, uav) \xrightarrow{\mathcal{M}} (u - 1, p, uav)$

tape contents, $w \in (C \cup \{s_0\})^*$. (Recall that s_0 is the blank.) As usual, for configurations γ, δ we write $\gamma \vdash_{\mathcal{M}} \delta$ to mean that one of the quadruples of \mathcal{M} permits the transition from γ to δ , and write $\gamma \xrightarrow{*}_{\mathcal{M}} \delta$ to mean that there is a sequence of configurations $\gamma = \gamma_1, \gamma_2, \dots, \gamma_k = \delta$ such that $\gamma_i \vdash_{\mathcal{M}} \gamma_{i+1}$ for $1 \leq i < k$. Table 2.1 shows which transitions are permitted by each quadruple in \mathcal{M} (here $a \in C \cup \{s_0, \lambda, \rho\}$, $b \in C \cup \{s_0\}$). (Of course, for $a = \lambda, \rho$, only quadruples of the second and third kind, respectively, can occur in \mathcal{M} .)

\mathcal{M} is said to *accept* a string $w \in C^*$ if

$$(1, q_1, \lambda w \rho) \xrightarrow{*}_{\mathcal{M}} (i, \tilde{q}, \lambda w' \rho),$$

where q_1 is the initial state of \mathcal{M} and, of course, \tilde{q} is the final state. (Note carefully that unlike the situation for Turing machines, a configuration will be regarded as “accepting” only if \mathcal{M} is in its final state \tilde{q} .) If $A \subseteq C$, we write $L_A(\mathcal{M})$ for the set of all $w \in A^*$ that are accepted by \mathcal{M} . The main theorem is

Theorem 2.1 (Landweber–Kuroda). The language $L \subseteq A^*$ is context-sensitive if and only if there is a linear bounded automaton \mathcal{M} such that $L = L_A(\mathcal{M})$.

We begin with

Lemma 1. There is an algorithm that transforms any given context-sensitive grammar Γ with terminals T into a linear bounded automaton \mathcal{M} such that $L(\Gamma) = L_T(\mathcal{M})$.

Proof. Let \mathcal{V} be the set of variables of Γ , and let $S \in \mathcal{V}$ be the start symbol. The alphabet of \mathcal{M} will be $T \cup \mathcal{V}$. Let the productions of Γ be $u_i \rightarrow v_i$, $i = 1, 2, \dots, m$, where

$$u_i = \alpha_1^{(i)} \alpha_2^{(i)} \cdots \alpha_{k_i}^{(i)} \quad \text{and} \quad v_i = \beta_1^{(i)} \beta_2^{(i)} \cdots \beta_{l_i}^{(i)}; \quad (2.1)$$

$$\alpha_1^{(i)}, \alpha_2^{(i)}, \dots, \alpha_{k_i}^{(i)}, \beta_1^{(i)}, \beta_2^{(i)}, \dots, \beta_{l_i}^{(i)} \in T \cup \mathcal{V},$$

and $k_i \leq l_i$. Then we set

$$\alpha_{k_i+1}^{(i)} = \alpha_{k_i+2}^{(i)} = \cdots = \alpha_{l_i}^{(i)} = s_0.$$

That is, we fill out the left side of each production with blanks. Since \mathcal{M} is operating nondeterministically, it can seek the word v_i on the tape and replace it by u_i , thus undoing the work of the production. It will help in following the construction of the automaton \mathcal{M} if we think of it as operating in one of these four phases: *initialization*, *searching*, *production undoing*, and *termination*. The states of \mathcal{M} will be the *initial state* q_1 , the *search state* σ , the *return state* $\bar{\sigma}$, the *undoing states* $p_j^{(i)}, q_j^{(i)}$ for $1 \leq i \leq m$ and $1 \leq j \leq l_i$ [l_i is as defined in Eqs. (2.1)], and the *termination states* $\tau, \bar{\tau}$.

Phase 1 (Initialization) We place in \mathcal{M} the quadruples

$$\left. \begin{array}{llll} q_1 & a & a & \sigma \\ q_1 & a & a & \tau \end{array} \right\} \quad a \in \mathcal{V} \cup T \cup \{s_0\}.$$

Thus in Phase 1, \mathcal{M} operating nondeterministically “decides” to enter either the search or the termination phase.

Phase 2 (Search) We place in \mathcal{M} the quadruples

$$\begin{array}{llllll} \sigma & a & R & \sigma & & a \neq \rho \\ \sigma & \rho & L & \bar{\sigma} & & \\ \sigma & \beta_1^{(i)} & \beta_1^{(i)} & p_1^{(i)} & 1 \leq i \leq m \\ \bar{\sigma} & a & L & \bar{\sigma} & a \neq \lambda \\ \bar{\sigma} & \lambda & R & q_1. & & \end{array}$$

In Phase 2, \mathcal{M} moves right along the tape searching for one of the initial symbols $\beta_1^{(i)}$ of the right side of a production. Finding one, \mathcal{M} may enter an *undoing state*. If \mathcal{M} encounters the right end marker ρ while still in state σ , it enters the *return state* $\bar{\sigma}$ and goes back to the beginning.

Phase 3 (Production Undoing) We place in \mathcal{M} the quadruples, for $1 \leq j < l_i, 1 \leq i \leq m$,

$$\begin{array}{llll} p_j^{(i)} & \beta_j^{(i)} & \alpha_j^{(i)} & q_j^{(i)} \\ q_j^{(i)} & \alpha_j^{(i)} & R & p_{j+1}^{(i)} \\ p_{l_i}^{(i)} & \beta_{l_i}^{(i)} & \alpha_{l_i}^{(i)} & \bar{\sigma} \end{array}$$

together with the quadruples

$$p_j^{(i)} \ s_0 \ R \ p_j^{(i)}.$$

When operating in Phase 3, \mathcal{M} has the opportunity to replace the right side of one of the productions on the tape by the left side (ignoring any

blanks that might have been introduced by previous replacements). If \mathcal{M} succeeds, it can enter the return state $\bar{\sigma}$, return to the left, and begin again.

Phase 4 (Termination) We place in \mathcal{M} the quadruples

$$\begin{array}{llll} \tau & s_0 & R & \tau \\ \tau & S & R & \bar{\tau} \\ \bar{\tau} & s_0 & R & \bar{\tau} \\ \bar{\tau} & \rho & L & \tilde{q}. \end{array}$$

Thus if \mathcal{M} ever returns to state q_1 with the tape contents

$$\lambda s_0^{(i)} S s_0^{(j)} \rho \quad i, j \geq 0$$

(where, of course, S is the start symbol of Γ), then \mathcal{M} will have the opportunity to move all the way to the right in this phase and to enter the final state \tilde{q} .

Thus, \mathcal{M} will accept a word $w \in T^*$ just in case there is a derivation of w from S in Γ . ■

Lemma 2. If $L \subseteq A^*$ is a context-sensitive language, then there is a linear bounded automaton \mathcal{M} such that $L = L_A(\mathcal{M})$.

Proof. We have $L = L(\Gamma)$ or $L(\Gamma) \cup \{0\}$ for a context-sensitive grammar Γ . In the first case, \mathcal{M} can be obtained as in Lemma 1. In the second case, we modify the automaton \mathcal{M} of Lemma 1 by adding the quadruple $q_1 \rho L \tilde{q}$. The modified automaton accepts 0 as well as the strings that \mathcal{M} accepts. ■

Now, we wish to discuss the converse situation: we are given a linear bounded automaton \mathcal{M} and alphabet A and wish to obtain a context-sensitive grammar Γ such that $L(\Gamma) = L_A(\mathcal{M}) - \{0\}$. The construction will be similar to the simulation, in Chapter 7, of a Turing machine by a semi-Thue process. However, the coding must be tighter because all the productions need to be non-length-decreasing.

Let \mathcal{M} be the given linear bounded automaton with alphabet C where $A \subseteq C$, initial state q_1 , and final state \tilde{q} . To begin with, we will only consider words $u \in C^*$ for which $|u| \geq 2$; such words can be written awb , where $w \in C^*$, $a, b \in C$. We wish to code a configuration $(i, q, \lambda awb\rho)$ of \mathcal{M} by a word of length $|awb| = |w| + 2$. To help us in doing this, we will use five variants on each letter $a \in C$:

$$a^\dagger a^\ddagger \bar{a} \vec{a}.$$

The interpretation of these markings is

- $|a|$: a on the left end of the word;
- a^{\dagger} : a on the right end of the word;
- \bar{a} : a on the left end, but the symbol being scanned is λ , one square to the left of a ;
- \vec{a} : a on the right end, but the symbol being scanned is ρ , one square to the right of a .

Finally, the current state will ordinarily be indicated by a subscript on the scanned symbol. If however, the scanned symbol is λ or ρ , the subscript will be on the adjacent symbol, marked, as just indicated, by an arrow. Thus, if \mathcal{M} has n states we introduce $3(n + 1) + 2n$ symbols for each $a \in C$. (Note that \bar{a} and \vec{a} always have a subscript.) The examples in Table 2.2 should make matters plain. Of course, this encoding only works for words $\lambda w\rho$ for which $|w| \geq 2$.

Now we will construct a semi-Thue process Σ such that given configurations γ, δ of \mathcal{M} and their codes $\bar{\gamma}, \bar{\delta}$, respectively, we shall have

$$\gamma \vdash_{\mathcal{M}} \delta \text{ if and only if } \bar{\gamma} \Rightarrow \bar{\delta}.$$

As for Turing machines, we define Σ by introducing suitable productions corresponding to each quadruple of \mathcal{M} . The correspondence is shown in Table 2.3, where we have written \bar{C} for $C \cup \{s_0\}$.

Now, since these productions simulate the behavior of \mathcal{M} in an obvious and direct manner, we see that \mathcal{M} will accept the string $aub, u \in C^*$, $a, b \in C$, just in case there is a derivation, from the initial word $|a_{q_1}ub|$ using these productions, of a word containing \tilde{q} as a subscript. To put this result in a more manageable form, we add to the alphabet of Σ the symbol S and add to Σ the “cleanup” productions

$$\alpha_{\tilde{q}} \rightarrow S, \quad \alpha S \rightarrow S, \quad S\alpha \rightarrow S, \quad (2.2)$$

where α can be any one of $a, |a, a^{\dagger}|, \bar{a}$, or \vec{a} , for any $a \in C$. Since these productions will transform the codes for configurations with the final state

Table 2.2

Configuration	Code
$(3, q, \lambda ababc\rho)$	$ aba_{q_1}bc $
$(1, q, \lambda ababc\rho)$	$ a_{q_1}babc $
$(5, q, \lambda ababc\rho)$	$ ababc _{q_1}$
$(0, q, \lambda ababc\rho)$	$\bar{a}_{q_1}babc $
$(6, q, \lambda ababc\rho)$	$ abab\vec{c}_{q_1}$

Table 2.3

Quadruple of \mathcal{M}	Productions of Σ
$q \ a \ b \ p, \quad a, b \in \bar{C}$	$a_q \rightarrow b_p$ $a_q \rightarrow [b_p]$ $a_q^1 \rightarrow b_p^1$
$q \ a \ R \ p, \quad a \in \bar{C}$	$a_q b \rightarrow ab_p$ $[a_q b] \rightarrow [ab_p]$ $[a_q b^1] \rightarrow [ab_p^1]$ $[a_q b^1] \rightarrow [ab_p]$ $a_q^1 \rightarrow \vec{a}_p$
$q \ \lambda \ R \ p$	$\vec{a}_q \rightarrow [a_p], \quad \text{all } a \in \bar{C}$
$q \ a \ L \ p, \quad a \in \bar{C}$	$ba_q \rightarrow b_p a$ $ba_q^1 \rightarrow b_p a^1$ $[ba_q] \rightarrow [b_p a]$ $[ba_q^1] \rightarrow [b_p a^1]$ $[a_q] \rightarrow \vec{a}_p$
$q \ \rho \ L \ p$	$\vec{a}_q \rightarrow a_p^1, \quad \text{all } a \in \bar{C}$

\tilde{q} into the single symbol S , and since there is no other way to obtain the single symbol S using the productions of Σ , we have

Lemma 3. \mathcal{M} accepts the string aub , $a, b \in C, u \in C^*$, if and only if

$$[a_{q_1} ub] \xrightarrow[\Sigma]{} S.$$

Now let Ω be the semi-Thue process whose productions are the *inverses* of the productions of Σ . (See Chapter 7, Section 2.) Then we have

Lemma 4. \mathcal{M} accepts the string aub , $a, b \in C, u \in C^*$, if and only if

$$S \xrightarrow[\Omega]{*} [a_{q_1} ub].$$

Now we are ready to define a context-sensitive grammar Γ . Let the terminals of Γ be the members of A , let the variables of Γ be

1. the symbols from the alphabet of Ω that do not belong to A , and
2. symbols a^0 for each $a \in A$.

Finally, the productions of Γ are the productions of Ω together with

$$\left. \begin{array}{l} a_{q_1} \rightarrow a^0 \\ a^0 b \rightarrow ab^0 \\ a^0 b^1 \rightarrow ab \end{array} \right\} \text{ for all } a, b \in A. \quad (2.3)$$

It is easy to check that Γ is in fact context-sensitive. [Of course, the productions (2.2) must be read from right to left, since it is the inverses of (2.2) that appear in Γ .] Moreover, using Lemma 4 and (2.3), we have

Lemma 5. Let $w \in A^*$. Then $w \in L(\Gamma)$ if and only if $|w| \geq 2$ and $w \in L_A(\mathcal{M})$.

Now let \mathcal{M} be a given linear bounded automaton, A a given alphabet, and let Γ be the context-sensitive grammar just constructed. Then, by Lemma 5, we have

$$L_A(\mathcal{M}) = L(\Gamma) \cup L_0,$$

where L_0 is the set of words $w \in A^*$ accepted by \mathcal{M} such that $|w| < 2$. But L_0 is finite, hence (Corollary 4.7 in Chapter 9) L_0 is a regular language, and so is certainly context-sensitive. Finally, using Theorem 1.3, we see that $L_A(\mathcal{M})$ is context-sensitive. This, together with Lemma 2, completes the proof of Theorem 2.1. ■

Exercises

- Let \mathcal{M} be the linear bounded automaton with initial state q_1 , final state \tilde{q} , and quadruples

$$\begin{array}{ccccccccc} q_1 & a & R & q_2 & & q_2 & b & R & q_1 \\ q_1 & b & R & q_3 & & q_2 & c & R & q_1 \\ q_1 & c & R & q_1 & & q_3 & a & R & q_1 \\ q_1 & \rho & L & \tilde{q} & & q_3 & c & R & q_1. \end{array}$$

What is $L(\mathcal{M})$?

- Give a deterministic linear bounded automaton \mathcal{M} that accepts $\{w \in \{a, b, c\}^* \mid w \text{ has an equal number of } a's, b's, c's\}$.
- Give a linear bounded automaton \mathcal{M} that accepts $\{ww \mid w \in \{a, b\}^*\}$.
- Let \mathcal{M} be the linear bounded automaton with initial state q_1 , final state \tilde{q} , and quadruples

$$\begin{array}{ccccccccc} q_1 & a & R & q_1 & & q_2 & b & R & q_2 \\ q_1 & b & R & q_2 & & q_2 & \rho & L & \tilde{q}. \end{array}$$

- (a) Use the construction in the proof of Theorem 2.1 to give a grammar Γ such that $L(\Gamma) = L(\mathcal{M})$.
 - (b) Give a derivation of $aabb$ in Γ .
5. Let Γ be the grammar with start symbol S and productions $S \rightarrow aSb$, $S \rightarrow ab$.
- (a) Use the construction in the proof of Theorem 2.1 to give a linear bounded automaton \mathcal{M} such that $L(\mathcal{M}) = L(\Gamma)$.
 - (b) Give an accepting computation by \mathcal{M} for input $aabb$.
6. Prove that every context-free language is accepted by a deterministic linear bounded automaton.
7. Show that there is an algorithm to test a given linear bounded automaton \mathcal{M} and word w to determine whether or not \mathcal{M} will eventually halt on input w . That is, the halting problem is solvable for linear bounded automata. [Hint: Consider the pigeon-hole principle.]

3. Closure Properties

We have already seen that the context-sensitive languages are closed under union (Theorem 1.3), and now we consider *intersection*. Here, although the context-free languages are *not* closed under intersection (Theorem 5.2 in Chapter 10), we can prove

Theorem 3.1. If L_1 and L_2 are context-sensitive languages, then so is $L_1 \cap L_2$.

Proof. Let $L_1 = L_A(\mathcal{M}_1)$, $L_2 = L_A(\mathcal{M}_2)$, where $\mathcal{M}_1, \mathcal{M}_2$ are linear bounded automata. The idea of the proof is to test a string w for membership in $L_1 \cap L_2$ by first seeing whether \mathcal{M}_1 will also accept w and then, if \mathcal{M}_1 does, to see whether \mathcal{M}_2 will also accept w . The difficulty is that \mathcal{M}_1 may destroy the input w in the process of testing it. If we were working with Turing machines, we would be able to deal with this kind of problem by saving a copy of the input on a part of the tape that remained undisturbed. Since linear bounded automata have no extra space, the problem must be solved another way. The solution uses an important idea: we think of our tape as consisting of a number of separate “tracks,” in this case two tracks. We will construct a linear bounded automaton \mathcal{M} that will work as follows:

1. \mathcal{M} will copy the input so it appears on both the upper and the lower track of the tape;

2. \mathcal{M} will simulate \mathcal{M}_1 working on the upper track only;
3. if \mathcal{M}_1 has accepted, \mathcal{M} will then simulate \mathcal{M}_2 working on the lower track (on which the original input remains undisturbed).

Thus, let us assume that \mathcal{M}_1 and \mathcal{M}_2 both have the alphabet $C = \{s_1, s_2, \dots, s_n\}$. (Of course, in addition they may use the symbols λ, ρ, s_0 .) \mathcal{M} will be a linear bounded automaton using, in addition, the symbols $b_j^i, 0 \leq i, j \leq n$. We think of the presence of the symbol b_j^i as indicating that s_i is on the “upper track” while s_j is on the “lower track” at the indicated position. Finally we assume that q_1 is the initial state of \mathcal{M}_1 , that \bar{q} is its final state, and that q_2 is the initial state of \mathcal{M}_2 . We also assume that the sets of states of \mathcal{M}_1 and \mathcal{M}_2 are disjoint. \mathcal{M} is to have initial state q_0 and have the same final state as \mathcal{M}_2 . \mathcal{M} is to contain the following quadruples (for $0 \leq i \leq n$):

- (1) Initialization:

$$\begin{array}{cccc} q_0 & s_i & b_i^i & \bar{q} \\ \bar{q} & b_i^i & R & q_0 \\ q_0 & \rho & L & \bar{\bar{q}} \\ \bar{\bar{q}} & b_i^i & L & \bar{\bar{q}} \\ \bar{\bar{q}} & \lambda & R & q_1. \end{array}$$

Here $\bar{q}, \bar{\bar{q}}$ are not among the states of \mathcal{M}_1 and \mathcal{M}_2 . These quadruples cause \mathcal{M} to copy the input on both “tracks” and then to return to the leftmost symbol of the input.

(2) For each quadruple of \mathcal{M}_1 , the corresponding quadruples, obtained by replacing each s_i by $b_j^i, j = 0, 1, \dots, n$, are to be in \mathcal{M} . These quadruples cause \mathcal{M} to simulate \mathcal{M}_1 operating on the “upper” track. In addition, \mathcal{M} is to have the quadruples for $0 \leq i, j \leq n$:

$$\begin{array}{cccc} \tilde{q} & b_j^i & R & \tilde{q} \\ \tilde{q} & \rho & L & \tilde{p} \\ \tilde{p} & b_j^i & s_j & \tilde{p} \\ \tilde{p} & s_j & L & \tilde{p} \\ \tilde{p} & \lambda & R & q_2. \end{array}$$

Here again \tilde{p} does not occur among the states of $\mathcal{M}_1, \mathcal{M}_2$. These quadruples cause \mathcal{M} to restore the “lower” track and then to enter the initial state of \mathcal{M}_2 scanning the leftmost input symbol.

(3) Finally, \mathcal{M} is to contain all the quadruples of \mathcal{M}_2 .
Since it is plain that

$$L_A(\mathcal{M}) = L_A(\mathcal{M}_1) \cap L_A(\mathcal{M}_2),$$

the proof is complete. ■

As an application, we obtain an unsolvability result about context-sensitive grammars.

Theorem 3.2. There is no algorithm for determining of a given context-sensitive grammar Γ whether $L(\Gamma) = \emptyset$.

Proof. Suppose there were such an algorithm. We can show that there would then be an algorithm for determining of two given *context-free* grammars Γ_1, Γ_2 whether $L(\Gamma_1) \cap L(\Gamma_2) = \emptyset$, thus contradicting Theorem 6.6 in Chapter 10. For, since Γ_1, Γ_2 are context-sensitive, the constructive nature of the proofs of Theorems 2.1 and 3.1 will enable us to obtain a context-sensitive grammar Γ with $L(\Gamma) = L(\Gamma_1) \cap L(\Gamma_2)$. ■

We turn now to a question about context-sensitive languages that was one of the outstanding open problems in theoretical computer science for over two decades. In 1964 Kuroda raised the question: Are the context-sensitive languages closed under complementation? It remained unsettled until 1987, when Neil Immerman showed that the answer is yes. What is particularly interesting is that, after more than twenty years, the solution turned out to be surprisingly straightforward.

We will show that if $L \subseteq A^*$ is accepted by a linear bounded automaton, then so is $A^* - L$. Suppose that L is accepted by the linear bounded automaton \mathcal{M} with alphabet $\{s_1, \dots, s_{n-1}\}$ and states $\{q_1, \dots, q_k\}$. (We take q_k to be \tilde{q} , and we will sometimes write λ, ρ as s_n, s_{n+1} , respectively.) We want to find another linear bounded automaton \mathcal{N} which accepts when \mathcal{M} rejects and vice versa. This would be easy if \mathcal{M} were deterministic, but suppose \mathcal{M} is nondeterministic. If $w \notin L$ then every computation by \mathcal{M} on input $\lambda w \rho$ is nonaccepting, so if we constructed \mathcal{N} to simulate \mathcal{M} and enter the final state \tilde{q} precisely when \mathcal{M} halts in a state other than \tilde{q} , then every halting computation by \mathcal{N} would enter \tilde{q} and \mathcal{N} would accept w (if it has at least one halting computation). However, if $w \in L$, then \mathcal{M} could still have some computations which halt in some state other than \tilde{q} , in which case \mathcal{N} would still accept w . Thus, we need \mathcal{N} to accept only when every computation of \mathcal{M} fails to end in state \tilde{q} .

The problem is that it is not at all clear how to construct \mathcal{N} so that a single computation by \mathcal{N} can correctly gather information about every computation by \mathcal{M} . We could deterministically simulate \mathcal{M} , using a stack

to remember “branch points.” However, \mathcal{M} has $|\lambda w\rho| \cdot k \cdot n^{|w|}$ distinct configurations with $|\lambda w\rho|$ tape squares, so a nonlooping computation by \mathcal{M} on input $\lambda w\rho$ could run for as many as $|\lambda w\rho| \cdot k \cdot n^{|w|} - 1$ steps, and each step could require adding more information to the stack. There is no way, then, that such a stack can be stored in $|w|$ tape squares, even using multiple tracks as in the proof of Theorem 3.1. Actually, there are simulation techniques that are much more efficient in terms of space, but none are known that are sufficiently parsimonious for our purposes here.

The solution discovered by Immerman is to store sufficient information about the possible computations by *counting* configurations. The largest value that needs to be stored is $|\lambda w\rho| \cdot k \cdot n^{|w|}$, which for any $w \neq 0$ can be represented in base n notation by a string of length

$$\leq \log_n |\lambda w\rho| + \log_n k + |w| + 1 \leq c \cdot |w|$$

for some constant c . (We can ignore the case $w = 0$ since the decision to accept or reject 0 can be built explicitly into the quadruples of \mathcal{N} .) The important thing is that c does not depend on w , so we can construct \mathcal{N} to maintain each such counter on c tracks, regardless of the length of the input. In fact, it will be convenient to consider the c tracks holding a counter as a single track with c “subtracks.”

The other objects we need to represent are configurations. If the initial configuration is $(1, q_1, \lambda w\rho)$, then it is clear that we can represent on a single track any configuration $(i, q, \lambda x\rho)$ where $|x| = |w|$. For example, we could add to the alphabet of some track new symbols s_i^j , $0 \leq i \leq n + 1$ and $1 \leq j \leq k$. Then s_i^j in square l on this track represents \mathcal{M} in state q_j scanning square l (on its own tape) holding symbol s_i . Not every string on the alphabet

$$\{s_0, \dots, s_{n+1}\} \cup \{s_i^j \mid 0 \leq i \leq n + 1; 1 \leq j \leq k\}$$

represents a configuration of \mathcal{M} , but it is clear that the representations of all configurations of \mathcal{M} with $|\lambda w\rho|$ tape squares can be written one after another, say, in ascending numerical order, on some track. We will call the i th configuration in this enumeration C_i . Of these configurations, some may never occur in any computation by \mathcal{M} on input $\lambda w\rho$. We say that a configuration $(i, q, \lambda x\rho)$ is *reachable* from w if $(1, q_1, \lambda w\rho) \vdash^* (i, q, \lambda x\rho)$.

We describe the behavior of \mathcal{N} by means of two nondeterministic procedures, the COUNT phase and the TEST phase. Although these are written in an informal high-level notation, it should be clear that \mathcal{N} can be constructed to carry them out, using no more than $|\lambda w\rho|$ tape squares. We begin with the TEST phase, described in Figure 3.1, where we will see the importance of being able to count the reachable configurations. Suppose

```

COUNTER ← 0
for  $i = 1$  to  $|\lambda w\rho| \cdot k \cdot n^{|w|}$ 
    CONFIG ←  $C_i$ 
    nondeterministically simulate some computation by  $\mathcal{M}$ 
        on  $\lambda w\rho$  until it reaches CONFIG or terminates
    if CONFIG has been reached then
        if CONFIG is accepting
            then enter  $\tilde{q}$  and halt
        else COUNTER ← COUNTER + 1
    end for
    if COUNTER =  $r$  then enter  $\tilde{q}$  and halt
    else enter  $q'$  and halt

```

Figure 3.1. The TEST phase of \mathcal{N} .

we have a tape with w on track 1 and r on track 2, where r is the number of configurations reachable from w . We will write this tape as $\lambda w/r\rho$. The TEST phase needs four tracks in addition to tracks 1 and 2. Two are needed for variables i and COUNTER, which hold numbers $\leq |\lambda w\rho| \cdot k \cdot n^{|w|}$, one is needed for CONFIG, which holds representations of configurations with $|\lambda w\rho|$ tape squares, and a fourth is needed to simulate computations by \mathcal{M} on input $\lambda w\rho$. It is clear that each track is large enough for its purpose. Let q' be some nonfinal state of \mathcal{N} .

Claim 1. Executing the TEST phase, \mathcal{N} accepts w/r if and only if \mathcal{M} rejects w .

If \mathcal{M} accepts w , there are at most $r - 1$ nonaccepting reachable configurations, so any computation by \mathcal{N} will either

- run forever simulating some computation by \mathcal{M} ;
- simulate some computation by \mathcal{M} that halts in state \tilde{q} , or
- end with COUNTER $< r$.

Therefore, no computation by \mathcal{N} ends in state \tilde{q} , and \mathcal{N} rejects w/r . If \mathcal{M} rejects w then \mathcal{N} can “guess” computations by \mathcal{M} that reach every reachable configuration. None of these is accepting, so \mathcal{N} finishes with COUNTER = r and accepts w/r . This proves Claim 1.

Finally, we need to show that \mathcal{N} can correctly compute r prior to entering the TEST phase. It might seem that \mathcal{N} could simply guess r and then continue with the TEST phase. The problem is that, if \mathcal{N} incorrectly guesses some $r' < r$, then some computation by \mathcal{N} in the TEST phase might end with COUNTER = r' and accept w when it should reject it. Therefore, it is not enough that *some* computation by \mathcal{N} reach the TEST phase with the correct value of r . We must ensure that *every* computation

```

 $i \leftarrow 0$ 
COUNTER  $\leftarrow 0$ 
NEW-COUNTER  $\leftarrow 1$ 
[Main] if NEW_COUNTER = COUNTER then
    delete all but tracks 1 and 2 from tape
    goto TEST phase
 $i \leftarrow i + 1$ 
COUNTER  $\leftarrow$  NEW_COUNTER
NEW_COUNTER  $\leftarrow 0$ 
for  $j = 1$  to  $|\lambda w\rho| \cdot k \cdot n^{|w|}$ 
     $t \leftarrow 0$ 
    CONFIG1  $\leftarrow C_j$ 
    for  $l = 1$  to  $|\lambda w\rho| \cdot k \cdot n^{|w|}$ 
        CONFIG2  $\leftarrow C_l$ 
        nondeterministically simulate some computation
        by  $\mathcal{M}$  on  $\lambda w\rho$  until it reaches CONFIG2 or
        until  $i$  steps have been executed
        if CONFIG2 has been reached then
             $t \leftarrow t + 1$ 
            if CONFIG2 = CONFIG1 or
                CONFIG2  $\leftarrow$  CONFIG1 then
                    NEW_COUNTER  $\leftarrow$  NEW_COUNTER + 1
                    leave inner loop
            end for
            if  $l > |\lambda w\rho| \cdot k \cdot n^{|w|}$  and  $t <$  COUNTER then
                enter  $q'$  and halt
        end for
    goto Main

```

Figure 3.2. The COUNT phase of \mathcal{N} .

by \mathcal{N} that gets as far as the TEST phase must do so with the correct value of r . We will now show that this can be done.

For all $i \geq 0$, let r_i be the number of configurations of \mathcal{M} that can be reached from $(1, q_1, \lambda w\rho)$ in no more than i steps. Then there is some i_0 such that $r_{i_0} = r_{i_0+1} = r$. We will argue by induction that each r_i , for $1 \leq i \leq i_0$, is correctly computed by the COUNT phase, given in Figure 3.2. The input is the initial tape $\lambda w\rho$. We also need tracks to hold variables NEW_COUNTER, COUNTER, i , j , l , t , CONFIG1, and CONFIG2, and a track to use in simulating computations by \mathcal{M} . Again it is clear that sufficient space is available. We stipulate that NEW_COUNTER, which will eventually hold r , should be stored on track 2.

Claim 2. For $i \geq 0$, any computation by \mathcal{N} on $\lambda w\rho$ that completes i executions of the main loop has the correct value of r_i in NEW_COUNTER.

The claim is obvious for $i = 0$, so we assume it is true for some $i \geq 0$ and show that it is true for $i + 1$. Suppose some computation completes $i + 1$ executions of the main loop. Then throughout the $i + 1$ st execution of the main loop, $\text{COUNTER} = r_i$ by the induction hypothesis (since COUNTER is set to NEW_COUNTER at the beginning of the loop). CONFIG1 ranges over all configurations with $|\lambda w\rho|$ tape squares, and for each value of CONFIG1 we want NEW_COUNTER to be incremented just in case CONFIG1 is reachable within $i + 1$ steps. Now, for each value of CONFIG1 , the inner *for* loop¹ ends either with $l \leq |\lambda w\rho| \cdot k \cdot n^{|w|}$, meaning that the current CONFIG1 has been found to be reachable within $i + 1$ steps, or with $l > |\lambda w\rho| \cdot k \cdot n^{|w|}$ and $t = \text{COUNTER}$, meaning that all r_i of the configurations reachable within i steps have been found and none of them leads to CONFIG1 in 0 or 1 steps, i.e., CONFIG1 is not reachable within $i + 1$ steps. In the first case NEW_COUNTER is incremented and in the second case it is not, so the claim is true for $i + 1$.

To conclude we simply note that at least one computation by \mathcal{N} on $\lambda w\rho$ will correctly guess the appropriate computations by \mathcal{M} to simulate and will execute the main loop $i_0 + 1$ times, leaving r on track 2. Any such computation will then go on to execute the TEST phase, and, by Claim 1, \mathcal{N} will accept w if and only if \mathcal{M} rejects w . Therefore, by Theorem 2.1 we have proved

Theorem 3.3. If $L \subseteq A^*$ is context-sensitive, then so is $A^* - L$.

We conclude this chapter by mentioning another major problem concerning context-sensitive languages that remains open: is every context-sensitive language accepted by a *deterministic* linear bounded automaton?

Exercises

- Let L, L' be context-sensitive languages. Prove the following.
 - $L \cdot L'$ is context-sensitive.
 - L^* is context-sensitive.
 - $L^R = \{w^R \mid w \in L\}$ is context-sensitive.
- Let $L \subseteq A^*$ be an r.e. language. Show that there is a context-sensitive language $L' \subseteq (A \cup \{c\})^*$ such that for all $w \in A^*$, we have

$$w \in L \quad \text{if and only if} \quad wc^{[i]} \in L' \text{ for some } i \geq 0.$$

¹ We are assuming here that when a loop of the form *for* $i = 1$ to n runs to completion, it leaves $i = n + 1$.

3. Show that for every r.e. language L there is a context-sensitive grammar Γ such that the grammar obtained from Γ by adding a single production of the form $V \rightarrow 0$ generates L . [Hint: Use Exercise 2 and take c to be the variable V .]
4. Give alphabets A, P and a context-sensitive language $L \subseteq A^*$ such that $\text{Er}_P(L)$ is not context-sensitive.
5. Let A_1, A_2 be alphabets and let $L \subseteq A_1^*$ be context-sensitive. Let f be a substitution on A_1 such that for each $a \in A$, $f(a) \subseteq A_2^*$ is context-sensitive and $0 \notin f(a)$. Let g be a homomorphism from A_1^* to A_2^* such that $g(a) \neq 0$ for all $a \in A$. [See Exercise 4.5 in Chapter 9 for the definitions of *substitution* and *homomorphism*.]
 - (a) Prove that $f(L)$ is context-sensitive.
 - (b) Prove that $g(L)$ is context-sensitive.
 - (c) Give a context-sensitive language L' and homomorphism h such that $h(L')$ is not context-sensitive.

Part 3

Logic

12

Propositional Calculus

1. Formulas and Assignments

Let A be some given alphabet and let $\mathcal{A} \subseteq A^*$. Let $B = A \cup \{\neg, \wedge, \vee, \supset, \leftrightarrow, (,)\}$, where we assume that these additional symbols are not already in A . $\neg, \wedge, \vee, \supset, \leftrightarrow$ are called (*propositional*) *connectives*. Then by a *propositional formula over \mathcal{A}* we mean any element of B^* which either belongs to \mathcal{A} or is obtainable from elements of \mathcal{A} by repeated applications of the following operations on B^* :

1. transform α into $\neg \alpha$;
2. transform α and β into $(\alpha \wedge \beta)$;
3. transform α and β into $(\alpha \vee \beta)$;
4. transform α and β into $(\alpha \supset \beta)$;
5. transform α and β into $(\alpha \leftrightarrow \beta)$.

When the meaning is clear from the context, *propositional formulas over \mathcal{A}* will be called *\mathcal{A} -formulas* or even just *formulas* for short. In this context the elements of \mathcal{A} (which are automatically \mathcal{A} -formulas) are called *atoms*.

To make matters concrete we can take $A = \{p, q, r, s, I\}$, and let

$$\mathcal{A} = \{p^{[i]}, q^{[i]}, r^{[i]}, s^{[i]} | i \in N\}.$$

In this case the atoms are called propositional variables. We can think of the suffix $^{[i]}$ as a subscript and write $p_i = p^{[i]}$, $q_i = q^{[i]}$, etc. Here are a

few examples of formulas:

$$\begin{aligned} & ((\neg p \supset q) \supset p), \\ & (((p \wedge q) \supset r) \wedge ((p_1 \wedge q_1) \supset r_1)) \supset \neg s, \\ & (((p_1 \vee \neg p_2) \vee p_3) \wedge (\neg p_1 \vee p_3)). \end{aligned}$$

Although the special case of propositional variables really suffices for studying propositional formulas, it is useful in order to include later applications, to allow the more general case of an arbitrary language of atoms. (In fact our assumption that the atoms form a language is not really necessary.)

By an *assignment* on a given set of atoms \mathcal{A} we mean a function v which maps each atom into the set $\{\text{FALSE}, \text{TRUE}\} = \{0, 1\}$, where (recall Chapter 1, Section 4), as usual, we are identifying FALSE with 0 and TRUE with 1. Thus for each atom α we will have $v(\alpha) = 0$ or $v(\alpha) = 1$. Given an assignment v on a set of atoms \mathcal{A} , we now show how to define a value $\gamma^v \in \{0, 1\}$ for each \mathcal{A} -formula γ . The definition is by recursion and proceeds as follows:

1. if α is an atom, then $\alpha^v = v(\alpha)$;
2. if $\gamma = \neg \beta$, then $\gamma^v = \begin{cases} 1 & \text{if } \beta^v = 0 \\ 0 & \text{if } \beta^v = 1; \end{cases}$
3. $(\alpha \wedge \beta)^v = \begin{cases} 1 & \text{if } \alpha^v = \beta^v = 1 \\ 0 & \text{otherwise;} \end{cases}$
4. $(\alpha \vee \beta)^v = \begin{cases} 0 & \text{if } \alpha^v = \beta^v = 0 \\ 1 & \text{otherwise;} \end{cases}$
5. $(\alpha \supset \beta)^v = \begin{cases} 0 & \text{if } \alpha^v = 1 \text{ and } \beta^v = 0 \\ 1 & \text{otherwise;} \end{cases}$
6. $(\alpha \leftrightarrow \beta)^v = \begin{cases} 1 & \text{if } \alpha^v = \beta^v \\ 0 & \text{otherwise.} \end{cases}$

A set Ω of \mathcal{A} -formulas is said to be *truth-functionally satisfiable*, or just *satisfiable* for short, if there is an assignment v on \mathcal{A} such that $\alpha^v = 1$ for all $\alpha \in \Omega$; otherwise Ω is said to be (truth-functionally) *unsatisfiable*. If $\Omega = \{\gamma\}$ consists of a single formula, then we say that γ is (truth-functionally) *satisfiable* if Ω is; γ is (truth-functionally) *unsatisfiable* if Ω is unsatisfiable. γ is called a *tautology* if $\gamma^v = 1$ for all assignments v . It is obvious that

Theorem 1.1. γ is tautology if and only if $\neg \gamma$ is unsatisfiable.

We agree to write $\alpha = \beta$ for \mathcal{A} -formulas α, β to mean that for every assignment v on \mathcal{A} , $\alpha^v = \beta^v$. This convention amounts to thinking of an \mathcal{A} -formula as naming a mapping from $\{0, 1\}^n$ into $\{0, 1\}$ for some $n \in N$, so that two \mathcal{A} -formulas are regarded as the same if they determine the same

Table 1.1

α	β	$\neg \alpha$	$(\neg \alpha \vee \beta)$	$(\alpha \supset \beta)$	$(\beta \supset \alpha)$	$(\alpha \leftrightarrow \beta)$
1	1	0	1	1	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
0	0	1	1	1	1	1

mappings. [Thus, in high school algebra one writes $x^2 - 1 = (x - 1) \times (x + 1)$, although $x^2 - 1$ and $(x - 1)(x + 1)$ are quite different as *expressions*, because they determine the same mappings on numbers.] With this understanding, we are able to eliminate some of the connectives in favor of others in a systematic manner. In particular, the equations

$$(\alpha \supset \beta) = (\neg \alpha \vee \beta), \quad (1.1)$$

$$(\alpha \leftrightarrow \beta) = ((\alpha \supset \beta) \wedge (\beta \supset \alpha)) \quad (1.2)$$

enable us to limit ourselves to the connectives \neg , \wedge , \vee . The truth of these two equations is easily verified by examining the “truth” tables in Table 1.1, which show all four possibilities for the pair α^v, β^v .

With our use of the equal sign, all tautologies are equal to one another and likewise all unsatisfiable formulas are equal to one another. Since the equations

$$\alpha^v = 1 \quad \text{for all } v, \quad \beta^v = 0 \quad \text{for all } v$$

determine α to be a tautology and β to be unsatisfiable, it is natural to write 1 for any \mathcal{A} -formula which is a tautology and 0 for any \mathcal{A} -formula which is unsatisfiable. Thus $\alpha = 1$ means that α is a tautology, and $\alpha = 0$ means that α is unsatisfiable.

The system of \mathcal{A} -formulas, under the operations \neg , \wedge , \vee and involving the “constants” 0, 1 obeys algebraic laws, some of which are analogous to laws satisfied by the real numbers under the operations $-$, \cdot , $+$; but there are some striking differences as well. Specifically, we have, for all \mathcal{A} -formulas α, β, γ

absorption:

$$(\alpha \wedge 1) = \alpha$$

$$(\alpha \vee 0) = \alpha$$

contradiction; excluded middle:

$$(\alpha \wedge \neg \alpha) = 0$$

$$(\alpha \vee \neg \alpha) = 1$$

$$(\alpha \wedge 0) = 0$$

$$(\alpha \vee 1) = 1$$

idempotency:

$$(\alpha \wedge \alpha) = \alpha$$

$$(\alpha \vee \alpha) = \alpha$$

commutativity:

$$(\alpha \wedge \beta) = (\beta \wedge \alpha)$$

$$(\alpha \vee \beta) = (\beta \vee \alpha)$$

associativity:

$$(\alpha \wedge (\beta \wedge \gamma)) = ((\alpha \wedge \beta) \wedge \gamma)$$

$$(\alpha \vee (\beta \vee \gamma)) = ((\alpha \vee \beta) \vee \gamma)$$

distributivity:

$$(\alpha \wedge (\beta \vee \gamma)) = ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$$

$$(\alpha \vee (\beta \wedge \gamma)) = ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$$

De Morgan laws:

$$\neg(\alpha \wedge \beta) = (\neg \alpha \vee \neg \beta)$$

$$\neg(\alpha \vee \beta) = (\neg \alpha \wedge \neg \beta)$$

double negation:

$$\neg \neg \alpha = \alpha$$

These equations, which are easily checked using truth tables, are the basis of the so-called Boolean algebra. In each row, the equations on the left and right can be obtained from one another by simply interchanging all occurrences of “ \vee ” with “ \wedge ” and of “0” with “1.” This is a special case of a general principle. The truth tables in Table 1.2 show that if we think of 0 as representing “TRUE,” and 1, “FALSE” (instead of the other way around), the tables for “ \wedge ” and “ \vee ” will simply be interchanged. Thus a being from another planet watching us doing propositional calculus might be able to guess that that was in fact what we were doing. But this being would have no way to tell which truth value we were representing by 0 and which by 1, and therefore could not say which of the two connectives represents “and” and which “or.” Therefore we have the

General Principle of Duality: Any correct statement involving \wedge , \vee and 0, 1, can be translated into another correct statement in which 0 and 1 have been interchanged and \wedge and \vee have been interchanged.

Of course, in carrying out the translation, notions defined in terms of 0, 1, \wedge , and \vee must be replaced by their *duals*. For example, the dual of “ α is a tautology” is “ α is unsatisfiable.” (The first is “ $\alpha^v = 1$ for all v ”; the

Table 1.2

α	β	$(\alpha \wedge \beta)$	$(\alpha \vee \beta)$
1	1	1	1
0	1	0	1
1	0	0	1
0	0	0	0

second is “ $\alpha = 0$ for all v ”.) Thus the dual of the correct statement

$$\text{if } \alpha \text{ is a tautology, so is } (\alpha \vee \beta)$$

is the equally correct statement

$$\text{if } \alpha \text{ is unsatisfiable, so is } (\alpha \wedge \beta).$$

Returning to our list of algebraic laws, we note that in particular the operations \wedge and \vee are commutative and associative. We take advantage of this associativity to write simply

$$\bigwedge_{i \leq k} \alpha_i = (\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_k)$$

$$\bigvee_{i \leq k} \alpha_i = (\alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_k)$$

without bothering to specify any particular grouping of the indicated formulas. We freely omit parentheses that are not necessary to avoid ambiguity.

Exercises

1. For each of the following formulas tell whether it is (i) satisfiable, (ii) a tautology, (iii) unsatisfiable.
 - (a) $((p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r)))$.
 - (b) $((p \supset (q \supset r)) \leftrightarrow ((p \wedge q) \supset r))$.
 - (c) $(p \wedge \neg q)$.
 - (d) $((p \vee q) \supset p)$.
 - (e) $((\neg(p \supset q) \supset (p \wedge \neg q))$.
2. Apply the general principle of duality to each of the following true statements:
 - (a) $(p \vee \neg p)$ is a tautology.
 - (b) $(p \supset (q \supset p))$ is a tautology.
3. Prove that if α and β are formulas, then $\alpha = \beta$ if and only if the formula $(\alpha \leftrightarrow \beta)$ is a tautology.
4. Verify the laws of absorption, contradiction, etc. given in this section.
5. Let \mathcal{A} be a set of atoms, and define

$$\mathcal{A}_0 = \mathcal{A}$$

$$\mathcal{A}_{n+1} = \mathcal{A} \cup \{ \neg \alpha, (\alpha \wedge \beta),$$

$$(\alpha \vee \beta), (\alpha \supset \beta), (\alpha \leftrightarrow \beta) \mid \alpha, \beta \in \mathcal{A}_n \}.$$

Show by induction on n that for all $\alpha \in \mathcal{A}_n$, the number of left

parentheses equals the number of right parentheses. Conclude that any propositional formula over \mathcal{A} has an equal number of left and right parentheses.

6. Let $\mathcal{A}, \mathcal{A}'$ be sets of atoms such that $\mathcal{A} \subseteq \mathcal{A}'$, and let v, v' be assignments on $\mathcal{A}, \mathcal{A}'$, respectively, such that $v(\alpha) = v'(\alpha)$ for all atoms α in \mathcal{A} . Define \mathcal{A}_n , $n \geq 0$, as in Exercise 5, and show by induction on n that $v(\alpha) = v'(\alpha)$ for all formulas $\alpha \in \mathcal{A}_n$. Conclude that $v(\alpha) = v'(\alpha)$ for all propositional formulas over \mathcal{A} .

2. Tautological Inference

Let $\gamma_1, \gamma_2, \dots, \gamma_n, \gamma$ be \mathcal{A} -formulas. Then we write

$$\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$$

and call γ a *tautological consequence* of the *premises* $\gamma_1, \dots, \gamma_n$ if for every assignment v on \mathcal{A} for which $\gamma_1^v = \gamma_2^v = \dots = \gamma_n^v = 1$, we have also $\gamma^v = 1$. This relation of tautological consequence is the most important concept in the propositional calculus. However, we can easily prove

Theorem 2.1. The relation $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$ is equivalent to each of the following:

1. the formula $((\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n) \supset \gamma)$ is a tautology;
2. the formula $(\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n \wedge \neg \gamma)$ is unsatisfiable.

Proof. $((\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n) \supset \gamma)$ is *not* a tautology just in case for some assignment v , $(\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n)^v = 1$ but $\gamma^v = 0$. That is, just in case for some assignment v , $\gamma_1^v = \gamma_2^v = \dots = \gamma_n^v = 1$ but $\gamma^v = 0$, which means simply that it is not the case that $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$. Likewise

$$(\gamma_1 \wedge \gamma_2 \wedge \dots \wedge \gamma_n \wedge \neg \gamma)$$

is *satisfiable* if and only if for some assignment v , $\gamma_1^v = \gamma_2^v = \dots = \gamma_n^v = (\neg \gamma)^v = 1$, i.e., $\gamma_1^v = \gamma_2^v = \dots = \gamma_n^v = 1$, but $\gamma^v = 0$. ■

Thus the problem of tautological inference is reduced to testing a formula for satisfiability, or for being a tautology. Of course, in principle, such a test can be carried out by simply constructing a truth table. However, a truth table for a formula containing n different atoms will require 2^n rows. Hence, truth table construction may be quite unfeasible even for formulas of modest size.

Consider the example

$$((p \wedge q) \supset (r \wedge s)), ((p_1 \wedge q_1) \supset r_1), ((r_1 \wedge s) \supset s_1), p, q, q_1, p_1 \vDash s_1. \quad (2.1)$$

Since there are eight atoms, a truth table would contain $2^8 = 256$ rows. In this example we can reason directly. If v makes all the premises TRUE, then $(p \wedge q)^v = (p_1 \wedge q_1)^v = 1$. Therefore, $(r \wedge s)^v = r_1^v = 1$, and in particular $s^v = 1$. Thus, $(r_1 \wedge s)^v = 1$ and finally, $s_1^v = 1$. We will use Theorem 2.1 to develop more systematic methods for doing such problems.

Exercises

1. Which of the following are correct?
 - (a) $(p \supset q), p \vDash q$.
 - (b) $(p \supset q), q \vDash p$.
 - (c) $(p \supset q), \neg q \vDash \neg p$.
 - (d) $(p \supset (q \supset r)), (\neg s \vee p), q \vDash (s \supset r)$.
2. Apply Theorem 2.1 to Exercise 1.
3. Prove or disprove each of the following.
 - (a) $\alpha, \beta \vDash \gamma$ if and only if $\alpha \vDash (\beta \supset \gamma)$.
 - (b) $\alpha \vDash \beta$ and $\beta \vDash \alpha$ if and only if $\alpha = \beta$.
 - (c) if $\alpha \vDash \beta$ or $\alpha \vDash \gamma$ then $\alpha \vDash (\beta \vee \gamma)$.
 - (d) if $\alpha \vDash \beta$ or $\alpha \vDash \gamma$ then $\alpha \vDash (\beta \wedge \gamma)$.
 - (e) if $\alpha \vDash \beta$ and $\alpha \vDash \gamma$ then $\alpha \vDash (\beta \wedge \gamma)$.
 - (f) if $\alpha \vDash \beta$ and $\alpha \vDash \gamma$ then $\alpha \vDash (\beta \vee \gamma)$.
 - (g) if $\alpha \vDash \neg \alpha$ then $\neg \alpha$ is a tautology.
 - (h) if $\alpha, \beta \vDash \gamma$ then $\alpha \vDash \gamma$ or $\beta \vDash \gamma$.
 - (i) if $\alpha \vDash \gamma$ then $\alpha, \beta \vDash \gamma$.
 - (j) if $\alpha \vDash (\beta \vee \gamma)$ then $\alpha \vDash \beta$ or $\alpha \vDash \gamma$.
4. (a) Show that if α is unsatisfiable then $\alpha \vDash \beta$ for any formula β .

(b) Show that if β is a tautology then $\alpha \vDash \beta$ for any formula α .

3. Normal Forms

We will now describe some algebraic procedures for simplifying \mathcal{A} -formulas:

(I) ELIMINATE \supset AND \leftrightarrow .

Simply use Eq. (1.2) for each occurrence of \leftrightarrow . After all such occurrences have been eliminated, use Eq. (1.1) for each occurrence of \supset .

Assuming (I) accomplished, we move on to

(II) MOVE \neg INWARD.

For any occurrence of \neg that is not immediately to the left of an atom either

1. the occurrence immediately precedes another \neg , in which case the pair $\neg\neg$ can be eliminated using the law of double negation; or
2. the occurrence immediately precedes an \mathcal{A} -formula of the form $(\alpha \wedge \beta)$ or $(\alpha \vee \beta)$, in which case one of the De Morgan laws can be applied to move the \neg inside the parentheses.

After (II) has been applied some finite number of times, a formula will be obtained to which (II) can no longer be applied. Such a formula must have each \neg immediately preceding an atom.

As an example of the use of (I) and (II) consider the formula

$$(((p \leftrightarrow q) \supset (r \supset s)) \wedge (q \supset \neg(p \wedge r))). \quad (3.1)$$

Eliminating \leftrightarrow gives

$$(((p \supset q) \wedge (q \supset p)) \supset (r \supset s)) \wedge (q \supset \neg(p \wedge r))).$$

Eliminate \supset :

$$(\neg((\neg p \vee q) \wedge (\neg q \vee p)) \vee (\neg r \vee s)) \wedge (\neg q \vee \neg(p \wedge r)). \quad (3.2)$$

Move \neg inward:

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

Move \neg inward:

$$((p \wedge \neg q) \vee (q \wedge \neg p) \vee \neg r \vee s) \wedge (\neg q \vee \neg p \vee \neg r). \quad (3.3)$$

A formula λ is called a *literal* if either λ is an atom or λ is $\neg\alpha$, where α is an atom. Note that if $\lambda = \neg\alpha$, for α an atom, then $\neg\lambda = \neg\neg\alpha = \alpha$. For α an atom it is convenient to write $\bar{\alpha}$ for $\neg\alpha$.

With this notation (3.3) becomes

$$((p \wedge \bar{q}) \vee (q \wedge \bar{p}) \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r}). \quad (3.4)$$

The distributive laws can be used to carry out further simplification, analogous to “multiplying out” in elementary algebra. However, the fact

that there are two distributive laws available is a complication because the “multiplying out” can proceed in two directions. As we shall see, each direction gives rise to a specific so-called *normal form*.

A handy technique that makes use of the reader’s facility with elementary algebra is to actually replace the symbols \wedge , \vee by $+$, \cdot and then calculate as in ordinary algebra. Since there are two distributive laws available, correct results will be obtained either by replacing \wedge by $+$ and \vee by \cdot or vice versa. Thus, writing \cdot for \wedge (and even omitting the \cdot as in elementary algebra) and $+$ for \vee , (3.4) can be written

$$\begin{aligned}
 & (p\bar{q} + q\bar{p} + \bar{r} + s) \cdot (\bar{q} + \bar{p} + \bar{r}) \\
 &= p\bar{q}\bar{q} + p\bar{q}\bar{p} + p\bar{q}\bar{r} + q\bar{p}\bar{q} + q\bar{p}\bar{p} + q\bar{p}\bar{r} + \bar{q}\bar{r} + \bar{r}\bar{p} + \bar{r}\bar{r} \\
 &\quad + s\bar{q} + s\bar{p} + s\bar{r} \\
 &= p\bar{q} + 0 + p\bar{q}\bar{r} + 0 + q\bar{p} + q\bar{p}\bar{r} + \bar{r}\bar{q} \\
 &\quad + \bar{r}\bar{p} + \bar{r} + s\bar{q} + s\bar{p} + s\bar{r} \\
 &= (p \wedge \bar{q}) \vee (p \wedge \bar{q} \wedge \bar{r}) \vee (q \wedge \bar{p}) \vee (q \wedge \bar{p} \wedge \bar{r}) \\
 &\quad \vee (\bar{r} \wedge \bar{q}) \vee (\bar{r} \wedge \bar{p}) \vee \bar{r} \vee (s \wedge \bar{q}) \vee (s \wedge \bar{p}) \vee (s \wedge \bar{r}), \quad (3.5)
 \end{aligned}$$

where we have used the principles of contradiction and absorption. Alternatively, writing $+$ for \wedge and \cdot for \vee , (3.4) can be written

$$\begin{aligned}
 & (p + \bar{q})(q + \bar{p})\bar{r}s + \bar{q}\bar{p}\bar{r} \\
 &= (pq + p\bar{p} + \bar{q}q + \bar{q}\bar{p})\bar{r}s + \bar{q}\bar{p}\bar{r} \\
 &= (pq + 1 + 1 + \bar{q}\bar{p})\bar{r}s + \bar{q}\bar{p}\bar{r} \\
 &= pq\bar{r}s + \bar{q}\bar{p}\bar{r}s + \bar{q}\bar{p}\bar{r} \\
 &= (p \vee q \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r}). \quad (3.6)
 \end{aligned}$$

Let λ_i be a sequence of *distinct* literals, $1 \leq i \leq n$. Then the formula $\bigvee_{i \leq n} \lambda_i$ is called an \vee -clause and the formula $\bigwedge_{i \leq n} \lambda_i$ is called an \wedge -clause. A pair of literals λ, λ' are called *mates* if $\lambda' = \neg \lambda$. We have

Theorem 3.1. Let λ_i be a literal for $1 \leq i \leq n$. Then the following are equivalent:

1. $\bigvee_{i \leq n} \lambda_i$ is a tautology;
2. $\bigwedge_{i \leq n} \lambda_i$ is unsatisfiable;
3. some pair $\lambda_i, \lambda_j, 1 \leq i, j \leq n$, is a pair of mates.

Proof. If $\lambda_j = \neg \lambda_i$, then obviously, $\bigvee_{i \leq n} \lambda_i$ is a tautology and $\bigwedge_{i \leq n} \lambda_i$ is unsatisfiable. If, on the other hand, the λ_i contain no pair of mates, then there are assignments v, w such that $v(\lambda_i) = 1, w(\lambda_i) = 0$ for $1 \leq i \leq n$. Then $(\bigvee_{i \leq n} \lambda_i)^v = 0, (\bigwedge_{i \leq n} \lambda_i)^v = 1$, so that $\bigvee_{i \leq n} \lambda_i$ is not a tautology and $\bigwedge_{i \leq n} \lambda_i$ is satisfiable. ■

Let κ_i , $1 \leq i \leq n$, be a sequence of *distinct* \vee -clauses. Then the \mathcal{A} -formula $\bigwedge_{i \leq n} \kappa_i$ is said to be in *conjunctive normal form* (CNF). Dually, if κ_i , $1 \leq i \leq n$, is a sequence of *distinct* \wedge -clauses, then the \mathcal{A} -formula $\bigvee_{i \leq n} \kappa_i$ is in *disjunctive normal form* (DNF). Note that (3.6) is in CNF and (3.5) is in DNF. We say that (3.6) is a *CNF of* (3.1) and that (3.5) is a *DNF of* (3.1). It should be clear that the procedures we have been describing will yield a CNF and a DNF for each \mathcal{A} -formula. Thus we have

Theorem 3.2. There is an algorithm which will transform any given \mathcal{A} -formula α into a formula β in CNF such that $\beta = \alpha$. There is a similar (in fact, dual) algorithm for DNF.

Because of Theorem 2.1, the following result is of particular importance.

Theorem 3.3. A formula in CNF is a tautology if and only if each of its \vee -clauses is a tautology. Dually, a formula in DNF is unsatisfiable if and only if each of its \wedge -clauses is unsatisfiable.

Proof. Let $\alpha = \bigwedge_{i \leq n} \kappa_i$, where each κ_i is an \vee -clause. If each κ_i is a tautology, then for any assignment v we have $\kappa_i^v = 1$ for $1 \leq i \leq n$, so that $\alpha^v = 1$; hence α is a tautology. If some κ_i is not a tautology, then there is an assignment v such that $\kappa_i^v = 0$; hence $\alpha^v = 0$ and α is not a tautology.

The proof for DNF is similar. Alternatively, we can invoke the general principle of duality. ■

Let us try to use these methods in applying Theorem 2.1 to example (2.1). First, using Theorem 2.1(1), we wish to know whether the following formula is a tautology:

$$\begin{aligned} (((p \wedge q) \supset (r \wedge s)) \wedge ((p_1 \wedge q_1) \supset r_1)) \\ \wedge ((r_1 \wedge s) \supset s_1) \wedge p \wedge q \wedge q_1 \wedge p_1 \supset s_1. \end{aligned}$$

Use of (I) yields

$$\begin{aligned} (\neg((\neg(p \wedge q) \vee (r \wedge s)) \wedge (\neg(p_1 \wedge q_1) \vee r_1)) \\ \wedge (\neg(r_1 \wedge s) \vee s_1) \wedge p \wedge q \wedge q_1 \wedge p_1) \vee s_1. \end{aligned}$$

Use of (II) gives

$$\begin{aligned} (\neg(\neg(p \wedge q) \vee (r \wedge s)) \vee \neg(\neg(p_1 \wedge q_1) \vee r_1)) \\ \vee \neg(\neg(r_1 \wedge s) \vee s_1) \vee \neg p \vee \neg q \vee \neg q_1 \vee \neg p_1 \vee s_1. \end{aligned}$$

Use of (II) again yields

$$\begin{aligned} ((p \wedge q \wedge \neg(r \wedge s)) \vee (p_1 \wedge q_1 \wedge \neg r_1) \vee (r_1 \wedge s \wedge \neg s_1) \\ \vee \neg p \vee \neg q \vee \neg q_1 \vee \neg p_1 \vee s_1). \end{aligned}$$

One final use of (II) gives

$$\begin{aligned} ((p \wedge q \wedge (\neg r \vee \neg s)) \vee (p_1 \wedge q_1 \wedge \neg r_1) \\ \vee (r_1 \wedge s \wedge \neg s_1) \vee \neg p \vee \neg q \vee \neg q_1 \vee \neg p_1 \vee s_1). \end{aligned} \quad (3.7)$$

To apply Theorem 3.3, it is necessary to find a CNF of (3.7). So we replace \wedge by $+$ and \vee by $:$:

$$(p + q + \bar{r}\bar{s})(p_1 + q_1 + \bar{r}_1)(r_1 + s + \bar{s}_1)\bar{p}\bar{q}\bar{q}_1\bar{p}_1s_1 \quad (3.8)$$

and see that the CNF of (3.7) will consist of 27 clauses. Here are three “typical” clauses from this CNF:

$$\begin{aligned} (p \vee p_1 \vee r_1 \vee \bar{p} \vee \bar{q} \vee \bar{q}_1 \vee \bar{p}_1 \vee s_1) \\ (\bar{r} \vee \bar{s} \vee q_1 \vee r_1 \vee \bar{p} \vee \bar{q} \vee \bar{q}_1 \vee \bar{p}_1 \vee s_1) \\ (q \vee \bar{r}_1 \vee \bar{s}_1 \vee \bar{p} \vee \bar{q} \vee \bar{q}_1 \vee \bar{p}_1 \vee s_1). \end{aligned}$$

Each of these clauses contains a pair of literals that are mates: p, \bar{p} in the first (and also p_1, \bar{p}_1); q_1, \bar{q}_1 in the second; and q, \bar{q} in the third (also s_1, \bar{s}_1). The same will be true for the remaining 24 clauses. But this is clearly not the basis for a very efficient algorithm. What if we try Theorem 2.1(2) on the same example? Then we need to show that the following formula is unsatisfiable:

$$\begin{aligned} ((p \wedge q) \supset (r \wedge s)) \wedge ((p_1 \wedge q_1) \supset r_1) \wedge ((r_1 \wedge s) \supset s_1) \\ \wedge p \wedge q \wedge q_1 \wedge p_1 \wedge \neg s_1. \end{aligned} \quad (3.9)$$

Using (I) we obtain

$$\begin{aligned} ((\neg(p \wedge q) \vee (r \wedge s)) \wedge (\neg(p_1 \wedge q_1) \vee r_1) \\ \wedge (\neg(r_1 \wedge s) \vee s_1) \wedge p \wedge q \wedge q_1 \wedge p_1 \wedge \neg s_1). \end{aligned}$$

Using (II) we obtain

$$\begin{aligned} & ((\neg p \vee \neg q \vee (r \wedge s)) \wedge (\neg p_1 \vee \neg q_1 \vee r_1) \\ & \quad \wedge (\neg r_1 \vee \neg s \vee s_1) \wedge p \wedge q \wedge q_1 \wedge p_1 \wedge \neg s_1). \end{aligned} \quad (3.10)$$

To find a DNF formula equal to this we replace \wedge by \cdot and \vee by $+$, obtaining

$$(\bar{p} + \bar{q} + rs)(\bar{p}_1 + \bar{q}_1 + r_1)(\bar{r}_1 + \bar{s} + s_1)pqq_1p_1\bar{s}_1.$$

But this is exactly the same as (3.8) except that each literal has been replaced by its mate! Once again we face essentially the same 27 clauses.

Suppose we seek a formula in CNF equal to (3.10) instead of a formula in DNF. We need only replace \wedge by $+$ and \vee by \cdot :

$$\bar{p}\bar{q}(r+s) + \bar{p}_1\bar{q}_1r_1 + \bar{r}_1\bar{s}s_1 + p+q+q_1+p_1+\bar{s}_1.$$

In this manner, we get a formula in which almost all “multiplying out” has already occurred. The CNF is simply

$$\begin{aligned} & (\bar{p} \vee \bar{q} \vee r) \wedge (\bar{p} \vee \bar{q} \vee s) \wedge (\bar{p}_1 \vee \bar{q}_1 \vee r_1) \\ & \quad \wedge (\bar{r}_1 \vee \bar{s} \vee s_1) \wedge p \wedge q \wedge q_1 \wedge p_1 \wedge \bar{s}_1. \end{aligned} \quad (3.11)$$

It consists of nine short, easily obtained clauses.

A moment’s reflection will show that this situation is entirely typical. Because the formula of Theorem 2.1(2) has the form

$$(\gamma_1 \wedge \gamma_2 \wedge \cdots \wedge \gamma_n \wedge \neg \gamma),$$

we can get a CNF formula simply by obtaining a CNF for each of the (ordinarily short) formulas $\gamma_1, \gamma_2, \dots, \gamma_n, \neg \gamma$. However, to obtain a DNF, which according to Theorem 3.3 is what we really want, we will have to multiply out $(n+1)$ polynomials. If, say, each of $\gamma_1, \dots, \gamma_n, \neg \gamma$ is an \vee -clause consisting of k literals, then the DNF will consist of k^{n+1} \wedge -clauses. And the general principle of duality guarantees (as we have already seen in our particular example) that the same discouraging arithmetic will emerge should we attempt instead to use Theorem 2.1(1). In this case a DNF will generally be easy to get, whereas a CNF (which is what we really want) will require a good deal of computing time.

These considerations lead to the following problem:

Satisfiability Problem. Find an efficient algorithm for testing an \mathcal{A} -formula in CNF to determine whether it is truth-functionally satisfiable.

This problem has been of central importance in theoretical computer science, not only for the reasons already given, but also for others that will emerge in Chapter 15.

Exercises

1. Find CNF and DNF formulas equal to each of the following.
 - (a) $((p \wedge (q \vee r)) \vee (q \wedge (p \vee r)))$.
 - (b) $((\neg p \vee (p \wedge \neg q)) \wedge (r \vee (\neg p \wedge q)))$.
 - (c) $(p \supset (q \leftrightarrow r))$.
2. Find a DNF formula that has the truth table

p	q	r	
1	1	1	0
0	1	1	1
1	0	1	1
0	0	1	1
1	1	0	0
0	1	0	1
1	0	0	0
0	0	0	0

[Hint: The second row of the table corresponds to the \wedge -clause $(\neg p \wedge q \wedge r)$. Each row for which the value is 1 similarly determines an \wedge -clause.]

3. Show how to generalize Exercise 2 to obtain a DNF formula corresponding to any given truth table.
4. Describe a dual of the method of Exercise 3 which, for any formula α , gives a DNF formula β such that $\alpha = \neg \beta$. Then show how to turn $\neg \beta$ into a CNF formula γ such that $\alpha = \gamma$. Apply the method to the truth table in Exercise 2. [Hint: Each row in the truth table for which the value is 0 corresponds to an \wedge -clause which should *not* be true.]
5. Let $\mathcal{A} = \{p, q, r\}$.
 - (a) Give a DNF formula α over \mathcal{A} such that $\alpha^v = 1$ for exactly three assignments v on \mathcal{A} .
 - (b) Give a CNF formula β over \mathcal{A} such that $\beta^v = 1$ for exactly three assignments v on \mathcal{A} .

6. (a) Let α be

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

Give DNF formulas β, γ, δ with 3, 2, 1 \wedge -clauses, respectively, such that $\alpha = \beta = \gamma = \delta$.

- (b) Let α be

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

Give CNF formulas β, γ, δ with 3, 2, 1 \vee -clauses, respectively, such that $\alpha = \beta = \gamma = \delta$.

7. Give a CNF formula α with two \vee -clauses such that $\alpha \neq \beta$ for all CNF formulas β with one \vee -clause.
 8. Use a normal form to show the correctness of the inference

$$(p \supset q), (r \vee \neg q), \neg(p \wedge r) \vDash \neg p.$$

4. The Davis – Putnam Rules

In order to make it easier to state algorithms for manipulating formulas in CNF, it will be helpful to give a simple representation of such formulas as sets. From now on we use the word *clause* to mean \vee -clause. We represent the clause $\kappa = \bigvee_{j \leq m} \lambda_j$ as the set $\kappa = \{\lambda_j | j \leq m\}$, and we represent the formula $\alpha = \bigwedge_{i \leq n} \kappa_i$, where each κ_i is a clause, as the set $\alpha = \{\kappa_i | i \leq n\}$. In so doing we lose the order of the clauses and the order of the literals in each clause; however, by the commutative laws, this does not matter.

It is helpful to speak of the empty set of literals as the *empty clause*, written \square , and of the empty set of clauses as the *empty formula*, written simply \emptyset . Since it is certainly true, although vacuously so, that there is an assignment (in fact any assignment will do) which makes every clause belonging to the empty formula true, it is natural and appropriate to agree that the *empty formula* \emptyset is *satisfiable* (in fact, it is a tautology). On the other hand, there is no assignment which makes some literal belonging to the empty clause \square true (because there are no such literals). Thus, we should regard the empty clause \square as being *unsatisfiable*. Hence any formula α such that $\square \in \alpha$ will be unsatisfiable as well.

We will give some rules for manipulating formulas in CNF that are helpful in designing algorithms for testing such formulas for satisfiability. By Theorem 3.1, a clause κ is a tautology if and only if $\lambda, \neg\lambda \in \kappa$ for some literal λ . Now, if $\kappa \in \alpha$ and κ is a tautologous clause, then α is satisfiable if and only if $\alpha - \{\kappa\}$ is. Hence, we can assume that the sets of clauses with which we deal contain no clauses which are tautologies. The following terminology is helpful: a clause $\kappa = \{\lambda\}$, consisting of a single literal, is called a *unit*. If α is a set of clauses and λ is a literal, then a clause κ is called λ -*positive* if $\lambda \in \kappa$, κ is called λ -*negative* if $\neg\lambda \in \kappa$, and κ is called λ -*neutral* if κ is neither λ -positive nor λ -negative. Since tautologous clauses have been excluded, no clause can be both λ -positive and λ -negative. We write α_λ^+ for the set of λ -positive clauses of α , α_λ^- for the set of λ -negative clauses of α , and α_λ^0 for the set of λ -neutral clauses of α . Thus for every literal λ , we have the decomposition $\alpha = \alpha_\lambda^+ \cup \alpha_\lambda^- \cup \alpha_\lambda^0$. Finally, we write

$$\text{POS}_\lambda(\alpha) = \alpha_\lambda^0 \cup \{\kappa - \{\lambda\} \mid \kappa \in \alpha_\lambda^+\},$$

$$\text{NEG}_\lambda(\alpha) = \alpha_\lambda^0 \cup \{\kappa - \{\neg\lambda\} \mid \kappa \in \alpha_\lambda^-\}.$$

Our main result is

Theorem 4.1 (Splitting Rule). Let α be a formula in CNF, and let λ be a literal. Then α is satisfiable if and only if at least one of the pair $\text{POS}_\lambda(\alpha)$ and $\text{NEG}_\lambda(\alpha)$ is satisfiable.

Proof. First let α be satisfiable, say $\alpha^v = 1$. Thus $\kappa^v = 1$ for all $\kappa \in \alpha$. That is, for each $\kappa \in \alpha$, there is a literal $\mu \in \kappa$ such that $\mu^v = 1$. Now, we must have either $\lambda^v = 1$ or $\lambda^v = 0$. Suppose first that $\lambda^v = 0$. We know that for each $\kappa \in \alpha_\lambda^+$, there is a literal $\mu \in \kappa$ such that $\mu^v = 1$. Thus this μ is not λ . Thus, for $\kappa \in \alpha_\lambda^+$, $(\kappa - \{\lambda\})^v = 1$. Hence, in this case, $\text{POS}_\lambda(\alpha)^v = 1$. If, instead, $\lambda^v = 1$, we can argue similarly that for each $\kappa \in \alpha_\lambda^-$, $(\kappa - \{\neg\lambda\})^v = 1$ and hence that $\text{NEG}_\lambda(\alpha)^v = 1$.

Conversely, let $\text{POS}_\lambda(\alpha)^v = 1$ for some assignment v . Then we define the assignment w by stipulating that

$$\lambda^w = 0; \quad \mu^w = \mu^v \quad \text{for all literals } \mu \neq \lambda, \neg\lambda.$$

Now, if $\kappa \in \alpha_\lambda^0$, then $\kappa^w = \kappa^v = 1$; if $\kappa \in \alpha_\lambda^+$, then $\kappa^w = (\kappa - \{\lambda\})^w = (\kappa - \{\lambda\})^v = 1$; finally, if $\kappa \in \alpha_\lambda^-$, then $\kappa^w = 1$ because $(\neg\lambda)^w = 1$. Thus, $\alpha^w = 1$.

If, instead, $\text{NEG}_\lambda(\alpha)^v = 1$ for some assignment v , we define w by

$$\lambda^w = 1; \quad \mu^w = \mu^v \quad \text{for all literals } \mu \neq \lambda, \neg\lambda.$$

Then if $\kappa \in \alpha_\lambda^0$, we have $\kappa^w = \kappa^v = 1$; if $\kappa \in \alpha_\lambda^-$, then $\kappa^w = (\kappa - \{\neg \lambda\})^w = (\kappa - \{\neg \lambda\})^v = 1$; finally, if $\kappa \in \alpha_\lambda^+$, then $\kappa^w = 1$ because $\lambda^w = 1$. Thus again $\alpha^w = 1$. ■

This theorem has the virtue of eliminating one literal, but at the price of considering two formulas instead of one. For this reason, it is of particular interest to find special cases in which we do not need to consider both $\text{POS}_\lambda(\alpha)$ and $\text{NEG}_\lambda(\alpha)$.

Thus, suppose that $\alpha_\lambda^- = \emptyset$. Then $\text{NEG}_\lambda(\alpha) = \alpha_\lambda^0 \subseteq \text{POS}_\lambda(\alpha)$. Hence, in this case, for any assignment v we have $\text{POS}_\lambda(\alpha)^v = 1$ implies $\text{NEG}_\lambda(\alpha)^v = 1$. Therefore, we conclude

Corollary 4.2 (Pure Literal Rule). If $\alpha_\lambda^- = \emptyset$, then α is satisfiable if and only if $\text{NEG}_\lambda(\alpha) = \alpha_\lambda^0$ is satisfiable.

For another useful special case, suppose that the unit clause $\{\lambda\} \in \alpha$. Then, since $\{\lambda\} - \{\lambda\} = \square$, we conclude that $\square \in \text{POS}_\lambda(\alpha)$. Hence, $\text{POS}_\lambda(\alpha)$ is unsatisfiable, and we have

Corollary 4.3 (Unit Rule). If $\{\lambda\} \in \alpha$, then α is satisfiable if and only if $\text{NEG}_\lambda(\alpha)$ is satisfiable.

To illustrate this last corollary by an example, let α be (3.11), which is a CNF of (3.9). Using the set representation,

$$\alpha = \{\{\bar{p}, \bar{q}, r\}, \{\bar{p}, \bar{q}, s\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{r}_1, \bar{s}, s_1\}, \{p\}, \{q\}, \{q_1\}, \{p_1\}, \{\bar{s}_1\}\}. \quad (4.1)$$

Thus, there are nine clauses, of which five are units. Using the unit clause $\{p\}$, Corollary 4.3 tells us that α is satisfiable if and only if $\text{NEG}_p(\alpha)$ is. That is, we need to test for satisfiability the set of clauses

$$\{\{\bar{q}, r\}, \{\bar{q}, s\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{r}_1, \bar{s}, s_1\}, \{q\}, \{q_1\}, \{p_1\}, \{\bar{s}_1\}\}.$$

Using the unit rule again, this time choosing the unit clause $\{q\}$, we reduce to

$$\{\{r\}, \{s\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{r}_1, \bar{s}, s_1\}, \{q_1\}, \{p_1\}, \{\bar{s}_1\}\}.$$

Using the unit clause $\{s\}$, we get

$$\{\{r\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{r}_1, s_1\}, \{q_1\}, \{p_1\}, \{\bar{s}_1\}\}.$$

Successive uses of the unit clauses $\{q_1\}$, $\{p_1\}$, $\{\bar{s}_1\}$ yield

$$\begin{aligned} & \{\{r\}, \{\bar{p}_1, r_1\}, \{\bar{r}_1, s_1\}, \{p_1\}, \{\bar{s}_1\}\}; \\ & \{\{r\}, \{r_1\}, \{\bar{r}_1, s_1\}, \{\bar{s}_1\}\}; \\ & \{\{r\}, \{r_1\}, \{\bar{r}_1\}\}. \end{aligned}$$

This last, containing the unit clauses $\{r_1\}$ and $\{\bar{r}_1\}$, is clearly unsatisfiable. Or, alternatively, applying the unit rule one last time, we obtain

$$\text{NEG}_{r_1}(\{\{r\}, \{r_1\}, \{\bar{r}_1\}\}) = \{\{r\}, \square\},$$

which is unsatisfiable because it contains the empty clause \square .

So we have shown by this computation that (4.1), and therefore (3.9), is unsatisfiable. And by Theorem 2.1, we then can conclude (once again) that the tautological inference (2.1) is valid.

A slight variant of this computation would begin by applying Corollary 4.2, the pure literal rule, to (4.1), using the literal r . This has the effect of simply deleting the first clause. The rest of the computation might then go as previously, but with the initial clause deleted at each stage.

For another example, recall (3.6), which was obtained as a CNF of (3.1). Written as a set of clauses this becomes

$$\beta = \{\{p, q, \bar{r}, s\}, \{\bar{q}, \bar{p}, \bar{r}, s\}, \{\bar{q}, \bar{p}, \bar{r}\}\}. \quad (4.2)$$

Here the pure literal rule can be applied using either of the literals \bar{r}, s . Thus, we have that β is satisfiable if and only if β_r^0 is satisfiable, if and only if β_s^0 is satisfiable. And we have

$$\beta_r^0 = \emptyset, \quad \beta_s^0 = \{\{\bar{q}, \bar{p}, \bar{r}\}\}.$$

From the first we see at once that β is satisfiable; if we wish to use the second, we can note by inspection that $(\beta_s^0)^v = 1$, where $v(q) = v(p) = v(r) = 0$, or we can use the pure literal rule a second time (using any of the three available literals) and once again arrive at the empty formula \emptyset .

We next turn to an example that has no unit clauses and to which the pure literal rule is not applicable:

$$\alpha = \{\{\bar{q}, p\}, \{r, p\}, \{\bar{p}, \bar{q}\}, \{\bar{p}, s\}, \{q, \bar{r}\}, \{q, \bar{s}\}\}.$$

Thus we are led to use the splitting rule forming, say,

$$\text{POS}_p(\alpha) = \{\{\bar{q}\}, \{r\}, \{q, \bar{r}\}, \{q, \bar{s}\}\},$$

$$\text{NEG}_p(\alpha) = \{\{\bar{q}\}, \{s\}, \{q, \bar{r}\}, \{q, \bar{s}\}\}.$$

Applying the pure literal rule once and then the unit rule twice to $\text{POS}_p(\alpha)$, we obtain successively

$$\{\{\bar{q}\}, \{r\}, \{q, \bar{r}\}\}, \quad \{\{r\}, \{\bar{r}\}\}, \quad \{\square\},$$

so that $\text{POS}_p(\alpha)$ is unsatisfiable. Doing the same to $\text{NEG}_p(\alpha)$ we obtain successively

$$\{\{\bar{q}\}, \{s\}, \{q, \bar{s}\}\}, \quad \{\{s\}, \{\bar{s}\}\}, \quad \{\square\},$$

so that $\text{NEG}_p(\alpha)$ is likewise unsatisfiable. By Theorem 4.1, we can thus conclude that α is unsatisfiable.

These examples suggest a rather systematic recursive procedure (sometimes known as the Davis–Putnam procedure) for testing a given formula α in CNF for satisfiability. The procedure as we shall describe it will not be completely deterministic; there will be situations in which one of a number of literals is to be selected. We will write the recursive procedure using two variables, γ for a set of clauses and \mathcal{S} for a stack of sets of clauses. We write $\text{TOP}(\mathcal{S})$ for the set of clauses at the top of the stack \mathcal{S} , $\text{POP}(\mathcal{S})$ for \mathcal{S} after $\text{TOP}(\mathcal{S})$ has been removed, $\text{PUSH}(\beta, \mathcal{S})$ for the stack obtained by putting β on the top of \mathcal{S} , and \emptyset for the empty stack. The procedure is as follows:

```

 $\gamma \leftarrow \alpha; \mathcal{S} \leftarrow \emptyset;$ 
while  $\gamma \neq \emptyset$  and ( $\square \in \gamma$  or  $\mathcal{S} \neq \emptyset$ )
  if  $\square \in \gamma$ 
    then  $\gamma \leftarrow \text{TOP}(\mathcal{S}); \mathcal{S} \leftarrow \text{POP}(\mathcal{S});$ 
  else if  $\gamma_\lambda^- = \emptyset$ 
    then  $\gamma \leftarrow \gamma_\lambda^0;$ 
  else if  $\{\lambda\} \in \gamma$ 
    then  $\gamma \leftarrow \text{NEG}_\lambda(\gamma);$ 
  else  $\mathcal{S} \leftarrow \text{PUSH}(\text{NEG}_\lambda(\gamma), \mathcal{S}); \gamma \leftarrow \text{POS}_\lambda(\gamma);$ 
end while
if  $\gamma = \emptyset$  then return SATISFIABLE
else return UNSATISFIABLE

```

Thus, this procedure will terminate returning SATISFIABLE whenever γ is the empty formula \emptyset , whether or not the stack \mathcal{S} is empty. (This is all right because the original formula will be satisfiable if any one of the formulas obtained by repeated uses of the splitting rule is satisfiable, and, of course, \emptyset is satisfiable.) The procedure will terminate returning UNSATISFIABLE if $\square \in \gamma$ and $\mathcal{S} = \emptyset$. (Here, γ is unsatisfiable, and no formulas remain in \mathcal{S} as the result of uses of the splitting rule.) If neither

of these termination conditions is satisfied, the algorithm will first test for $\square \in \gamma$. If $\square \in \gamma$, it rejects (since γ is unsatisfiable) and “pops” the stack. Otherwise it attempts to apply first the pure literal rule and then the unit rule. If both attempts fail, it chooses (nondeterministically) some literal λ , takes $\text{POS}_\lambda(\gamma)$ as the new formula to work on, and “pushes” $\text{NEG}_\lambda(\gamma)$ onto the stack for future reference.

It is not difficult to see that the algorithm just given must always terminate. Let us say that a set of clauses α *reduces* to a set of clauses β if for each clause κ in β there is a clause $\bar{\kappa}$ in α such that $\kappa \subseteq \bar{\kappa}$. Then, at the beginning of each pass through the *while* loop, γ is a set of clauses to which α reduces and the stack consists of a list of sets of clauses to each of which α reduces. Since, for a given α , there are only a finite number of distinct configurations of this kind, and none can be repeated, the algorithm must eventually terminate.

Exercises

- Let α be $\{\{p, q, r\}, \{p, \bar{q}\}, \{\bar{p}, \bar{r}\}\}$. For $\lambda = p, q, r, \bar{p}, \bar{q}, \bar{r}$, give $\alpha_\lambda^+, \alpha_\lambda^-, \alpha_\lambda^0, \text{POS}_\lambda(\alpha), \text{NEG}_\lambda(\alpha)$. Which of these sets are necessarily equal?
- Use the Davis–Putnam rules to show the correctness of the inference in Exercise 3.8.
- Use the Davis–Putnam rules to show the correctness of the following inference.

If John went swimming, then he lost his glasses and did not go to the movies. If John ate too much meat and did not go to the movies, then he will suffer indigestion. Therefore, if John ate too much meat and went swimming, then he will suffer indigestion.

- Test the following set of clauses for satisfiability:

$$\begin{aligned} & \{p, \bar{q}, r, s\} \\ & \{\bar{p}, q, \bar{r}\} \\ & \{\bar{r}, s\} \\ & \{\bar{q}, r\} \\ & \{p, \bar{s}\}. \end{aligned}$$

- Modify the Davis–Putnam procedure so that when the answer is SATISFIABLE on input α , it returns an assignment v such that $\alpha^v = 1$.

6. How many distinct computations can be performed by the Davis–Putnam procedure on input $\{\{p, q, r\}, \{p, \bar{q}\}, \{\bar{p}, \bar{r}\}\}$?
7. Let α be a CNF formula with n distinct atoms.
 - (a) What is the maximum number of formulas that can be on the stack in the Davis–Putnam procedure at any given time?
 - (b) Suppose that α is satisfiable. Show that if the Davis–Putnam procedure always makes the right choice of λ at each stage, the *while* loop executes no more than n times.
 - (c) How many times must the *while* loop execute on input

$$\{\{p, q\}, \{p, \bar{q}\}, \{\bar{p}, q\}, \{\bar{p}, \bar{q}\}\}?$$

On input

$$\{\{p, q, r\}, \{p, q, \bar{r}\}, \{p, \bar{q}, r\}, \{p, \bar{q}, \bar{r}\},$$

$$\{\bar{p}, q, r\}, \{\bar{p}, q, \bar{r}\}, \{\bar{p}, \bar{q}, r\}, \{\bar{p}, \bar{q}, \bar{r}\}\}?$$

5. Minimal Unsatisfiability and Subsumption

We begin with

Theorem 5.1. Let the clauses κ_1, κ_2 satisfy $\kappa_1 \subset \kappa_2$. Then if α is a formula in CNF such that $\kappa_1, \kappa_2 \in \alpha$, then α is satisfiable if and only if $\alpha - \{\kappa_2\}$ is satisfiable.

Proof. Clearly, if α is satisfiable, so is $\alpha - \{\kappa_2\}$.

Conversely, if $(\alpha - \{\kappa_2\})^v = 1$, then $\kappa_1^v = 1$, so that also $\kappa_2^v = 1$. Hence, $\alpha^v = 1$. ■

Thus, if in fact $\kappa_1, \kappa_2 \in \alpha$ and $\kappa_1 \subset \kappa_2$, we may simply drop κ_2 and test $\alpha - \{\kappa_2\}$ for satisfiability. The operation of dropping κ_2 in such a case is called *subsumption*. Unfortunately, there is no efficient algorithm known for testing a large set of clauses for the possibility of applying *subsumption*.

Definition. A finite set of clauses α is called *minimally unsatisfiable* if

1. α is unsatisfiable, and
2. for all $\beta \subset \alpha$, β is satisfiable.

Definition. A finite set of clauses α is said to be *linked* if whenever $\lambda \in \kappa_1$ and $\kappa_1 \in \alpha$, there is a clause $\kappa_2 \in \alpha$ such that $\neg \lambda \in \kappa_2$. That is, each literal in a clause of α has a *mate* in another clause of α .

Then it is very easy to prove

Theorem 5.2. Let α be minimally unsatisfiable. Then

1. for no $\kappa_1, \kappa_2 \in \alpha$ can we have $\kappa_1 \subset \kappa_2$, and
2. α is linked.

Proof. Condition 1 is an immediate consequence of Theorem 5.1. To verify 2, suppose that α is minimally unsatisfiable but not linked. Then, there is a literal λ in a clause $\kappa \in \alpha$ such that the literal $\neg\lambda$ occurs in none of the clauses of α , i.e., $\alpha_{\lambda}^- = \emptyset$. Thus, by the pure literal rule, α_{λ}^0 is unsatisfiable. But since $\alpha_{\lambda}^0 \subset \alpha$, this is a contradiction. ■

Exercise

1. Give a minimally unsatisfiable CNF formula with four clauses.

6. Resolution

Let κ_1, κ_2 be clauses such that $\lambda \in \kappa_1$ and $\neg\lambda \in \kappa_2$. Then we write

$$\text{res}_{\lambda}(\kappa_1, \kappa_2) = (\kappa_1 - \{\lambda\}) \cup (\kappa_2 - \{\neg\lambda\}).$$

The clause $\text{res}_{\lambda}(\kappa_1, \kappa_2)$ is then called the *resolvent of κ_1, κ_2 with respect to the literal λ* . The operation of forming resolvents has been the basis of a very large number of computer programs designed to perform logical deductions. We have

Theorem 6.1. Let λ be an atom and let κ_1, κ_2 be clauses such that $\lambda \in \kappa_1$, $\neg\lambda \in \kappa_2$. Then

$$\kappa_1, \kappa_2 \models \text{res}_{\lambda}(\kappa_1, \kappa_2).$$

Proof. Let v be an assignment such that $\kappa_1^v = \kappa_2^v = 1$. Now if $\lambda^v = 1$, then $(\kappa_2 - \{\neg\lambda\})^v = 1$, while if $\lambda^v = 0$, then $(\kappa_1 - \{\lambda\})^v = 1$. In either case, therefore, $\text{res}_{\lambda}(\kappa_1, \kappa_2)^v = 1$. ■

Let α be a finite set of clauses. A sequence of clauses $\kappa_1, \kappa_2, \dots, \kappa_n = \kappa$ is called a *resolution derivation of κ from α* if for each i , $1 \leq i \leq n$, either $\kappa_i \in \alpha$ or there are $j, k < i$ and a literal λ such that $\kappa_i = \text{res}_{\lambda}(\kappa_j, \kappa_k)$. A resolution derivation of \square from α is called a *resolution refutation of α* . We define

$$\text{RES}_{\lambda}(\alpha) = \alpha_{\lambda}^0 \cup \{\text{res}_{\lambda}(\kappa_1, \kappa_2) \mid \kappa_1 \in \alpha_{\lambda}^+, \kappa_2 \in \alpha_{\lambda}^-\}.$$

We have

Theorem 6.2. Let α be a formula in CNF and let λ be a literal. Then α is satisfiable if and only if $\text{RES}_\lambda(\alpha)$ is satisfiable.

Proof. First let $\alpha^v = 1$. Then if $\kappa \in \alpha_\lambda^0$, we have also $\kappa \in \alpha$, so that $\kappa^v = 1$. Furthermore, if $\kappa = \text{res}_\lambda(\kappa_1, \kappa_2)$, with $\kappa_1 \in \alpha_\lambda^+$, $\kappa_2 \in \alpha_\lambda^-$, then $\kappa_1^v = 1$, $\kappa_2^v = 1$, so that, by Theorem 6.1, $\kappa^v = 1$. Since for all $\kappa \in \text{RES}_\lambda(\alpha)$, we have $\kappa^v = 1$, it follows that $\text{RES}_\lambda(\alpha)^v = 1$.

Conversely, let $\text{RES}_\lambda(\alpha)^v = 1$. We claim that either $\text{POS}_\lambda(\alpha)^v = 1$ or $\text{NEG}_\lambda(\alpha)^v = 1$. For, suppose that $\text{POS}_\lambda(\alpha)^v = 0$. Since $\alpha_\lambda^0 \subseteq \text{RES}_\lambda(\alpha)$, we have $(\alpha_\lambda^0)^v = 1$. So for some $\kappa_1 \in \alpha_\lambda^+$, we must have $(\kappa_1 - \{\lambda\})^v = 0$. However, for all $\kappa_2 \in \alpha_\lambda^-$ and this κ_1 , we must have $\text{res}_\lambda(\kappa_1, \kappa_2)^v = [(\kappa_1 - \{\lambda\}) \cup (\kappa_2 - \{\neg \lambda\})]^v = 1$. Thus, for all $\kappa_2 \in \alpha_\lambda^-$ we have $(\kappa_2 - \{\neg \lambda\})^v = 1$, i.e., $\text{NEG}_\lambda(\alpha)^v = 1$. This proves our claim that either $\text{POS}_\lambda(\alpha)$ or $\text{NEG}_\lambda(\alpha)$ must be satisfiable. By Theorem 4.1, i.e., the splitting rule, α is satisfiable. ■

Theorem 6.2 suggests another procedure for testing a formula α in CNF for satisfiability. As with the Davis–Putnam rules, seek a literal of α to which the pure literal or unit rule can be applied. If none is to be found, choose a literal λ of α and compute $\text{RES}_\lambda(\alpha)$. Continue recursively.

As with the Davis–Putnam procedure, this procedure must eventually terminate in $\{\square\}$ or \emptyset ; this is because the number of literals is successively diminished. This procedure has the advantage of not requiring a stack of formulas, but the disadvantage that the problem may get considerably larger because of the use of the $\text{RES}_\lambda(\alpha)$ operation. Unfortunately, the present procedure is also called the Davis–Putnam procedure in the literature. To add to the confusion, it seems that computer implementations of the “Davis–Putnam procedure” have been almost exclusively of the procedure introduced in Section 4, whereas theoretical analyses of the computational complexity of the “Davis–Putnam procedure” have tended to deal with the procedure we have just introduced.

Theorem 6.3. Let α be a formula in CNF and suppose that there is a resolution derivation of the clause κ from α . Then $\alpha^v = 1$ implies $\kappa^v = 1$.

Proof. Let $\kappa_1, \kappa_2, \dots, \kappa_n = \kappa$ be a resolution derivation of κ from α . We shall prove that $\kappa_i^v = 1$ for $1 \leq i \leq n$, which will prove the result. To prove this by induction, we assume that $\kappa_j^v = 1$ for all $j < i$. (Of course for the case $i = 1$, this induction hypothesis is true vacuously.) Now, there are two cases. If $\kappa_i \in \alpha$, then $\kappa_i^v = 1$. Otherwise $\kappa_i = \text{res}_\lambda(\kappa_j, \kappa_k)$, where

$j, k < i$. Hence, by the induction hypothesis, $\kappa_j^v = \kappa_k^v = 1$. So by Theorem 6.1, $\kappa_i^v = 1$. ■

Theorem 6.4 (Ground Resolution Theorem). The formula α in CNF is unsatisfiable if and only if there is a resolution refutation of α .

Proof. First let there be a resolution refutation of α , but suppose that nevertheless $\alpha^v = 1$. Then, by Theorem 6.3, $\square^v = 1$, which is impossible.

Conversely, let α be unsatisfiable. Let $\lambda_1, \lambda_2, \dots, \lambda_k$ be a list of all the atoms that occur in α . Let

$$\alpha_0 = \alpha, \quad \alpha_i = \text{RES}_{\lambda_i}(\alpha_{i-1}), \quad i = 1, 2, \dots, k.$$

Clearly each α_i contains only the atoms λ_j for which $i < j \leq k$. Hence α_k contains no atoms at all, and must be either \emptyset or $\{\square\}$. On the other hand, by Theorem 6.2, we have that α_i is unsatisfiable for $0 \leq i \leq k$. Hence $\alpha_k = \{\square\}$. Now, let the sequence $\kappa_1, \kappa_2, \dots, \kappa_m$ of clauses consist, first, of all of the clauses of $\alpha_0 = \alpha$, then, all of the clauses of α_1 , and so on through all of the clauses of α_k . But this last means that $\kappa_m = \square$. Moreover, it is clear from the definition of the RES_λ operation that $\kappa_1, \kappa_2, \dots, \kappa_m$ is a resolution derivation. ■

To illustrate the ground resolution theorem, we apply it to (4.1) to show, once again, that (3.9) is unsatisfiable. Here then is a resolution refutation of the formula α of (4.1):

$$\begin{aligned} &\{\bar{p}, \bar{q}, s\}, \{\bar{r}_1, \bar{s}, s_1\}, \{\bar{p}, \bar{q}, \bar{r}_1, s_1\}, \{p\}, \{\bar{q}, \bar{r}_1, s_1\}, \{q\}, \{\bar{r}_1, s_1\}, \\ &\{\bar{s}_1\}, \{\bar{r}_1\}, \{\bar{p}_1, \bar{q}_1, r_1\}, \{\bar{p}_1, \bar{q}_1\}, \{q_1\}, \{\bar{p}_1\}, \{p_1\}, \square. \end{aligned}$$

Exercises

1. (a) Use the resolution method to answer Exercise 1.1.
(b) Do the same for Exercise 2.1.
2. Give a resolution refutation that shows the correctness of the inference of Exercise 3.8.
3. Do the same for the inference of Exercise 4.3.
4. Let $\alpha_0, \dots, \alpha_n$ be CNF formulas, and let β be a DNF formula $\bigvee_{i \leq m} \beta_i$. Show that $\alpha_0, \dots, \alpha_n \models \beta$ if and only if there is a resolution refutation of $\bigcup_{i \leq n} \alpha_i \cup \{\neg \beta_i \mid i \leq m\}$.

5. Let λ be an atom and let κ_1, κ_2 be clauses such that $\lambda \in \kappa_1$, $\neg \lambda \in \kappa_2$. Prove or disprove the following.
 - (a) $\text{res}_\lambda(\kappa_1, \kappa_2) \vDash (\kappa_1 \vee \kappa_2)$.
 - (b) $\text{res}_\lambda(\kappa_1, \kappa_2) \vDash \kappa_1$ or $\text{res}_\lambda(\kappa_1, \kappa_2) \vDash \kappa_2$.
 - (c) $\text{res}_\lambda(\kappa_1, \kappa_2) \vDash (\kappa_1 \wedge \kappa_2)$.
 - (d) if $\text{res}_\lambda(\kappa_1, \kappa_2)$ is valid, then $(\kappa_1 \wedge \kappa_2)$ is valid.
6. Let α be a formula in CNF and let λ be a literal. Prove or disprove that α is valid if and only if $\text{RES}_\lambda(\alpha)$ is valid.

7. The Compactness Theorem

Now, we will prove a theorem relating infinite sets of \mathcal{A} -formulas to their finite subsets.

Definition. A set Ω of \mathcal{A} -formulas is called *finitely satisfiable* if for every finite set $\Delta \subseteq \Omega$, the set Δ is truth-functionally satisfiable.

We have

Theorem 7.1. Let Ω be finitely satisfiable and let α be an \mathcal{A} -formula. Then either $\Omega \cup \{\alpha\}$ or $\Omega \cup \{\neg \alpha\}$ is finitely satisfiable.

Proof. Suppose to the contrary that Ω is finitely satisfiable but that neither $\Omega \cup \{\alpha\}$ nor $\Omega \cup \{\neg \alpha\}$ is finitely satisfiable. Then there are finite sets $\Delta_1, \Delta_2 \subseteq \Omega$ such that $\Delta_1 \cup \{\alpha\}$ and $\Delta_2 \cup \{\neg \alpha\}$ are both truth-functionally unsatisfiable. But $\Delta_1 \cup \Delta_2$ is a finite subset of Ω and hence there must be an assignment v such that for each $\beta \in \Delta_1 \cup \Delta_2$, we have $\beta^v = 1$. Now, either $\alpha^v = 1$ or $\alpha^v = 0$. In the first case $\Delta_1 \cup \{\alpha\}$ is satisfiable, and in the second case $\Delta_2 \cup \{\neg \alpha\}$ is satisfiable. This is a contradiction. ■

Now we will need to use a general property of infinite languages.

Enumeration Principle. Let L be an infinite subset of A^* , where A is an alphabet (and therefore is finite). Then there is an infinite sequence or *enumeration* w_0, w_1, w_2, \dots which consists of all the words in L each listed exactly once.

The truth of this enumeration principle can be seen in many ways. One is simply to imagine the elements of L written in order of increasing length, and to order words of the same length among themselves like the entries in a dictionary. Alternatively, one can regard the strings on A as notations

for numbers in some base (as in Chapter 5) and arrange the elements of L in numerical order. (Actually, as it is not difficult to see, these two methods yield the same enumeration.) Of course, no claim is made that there is an algorithm for computing w_i from i . Such an algorithm can only exist if the language L is r.e.

Now, let $\alpha_0, \alpha_1, \alpha_2, \dots$ be an enumeration of the set of all \mathcal{A} -formulas. (By the enumeration principle, such an enumeration must exist.) Let Ω be a given finitely satisfiable set of \mathcal{A} -formulas. We define the sequence

$$\Omega_0 = \Omega$$

$$\Omega_{n+1} = \begin{cases} \Omega_n \cup \{\alpha_n\} & \text{if this set is finitely satisfiable} \\ \Omega_n \cup \{\neg \alpha_n\} & \text{otherwise.} \end{cases}$$

By Theorem 7.1, we have

Lemma 1. Each Ω_n is finitely satisfiable.

Let $\tilde{\Omega} = \bigcup_{n=0}^{\infty} \Omega_n$. Then, we have

Lemma 2. $\tilde{\Omega}$ is finitely satisfiable.

Proof. Let us be given a finite set $\Delta \subseteq \tilde{\Omega}$. For each $\gamma \in \Delta$, $\gamma \in \Omega_n$ for some n . Hence $\Delta \subseteq \Omega_m$, where m is the maximum of those n . By Lemma 1, Δ is truth-functionally satisfiable. ■

Lemma 3. For each \mathcal{A} -formula α either $\alpha \in \tilde{\Omega}$ or $\neg \alpha \in \tilde{\Omega}$, but not both.

Proof. Let $\alpha = \alpha_n$. Then $\alpha \in \Omega_{n+1}$ or $\neg \alpha \in \Omega_{n+1}$, so that α or $\neg \alpha$ belongs to $\tilde{\Omega}$. If $\alpha, \neg \alpha \in \tilde{\Omega}$, then by Lemma 2, the finite set $\{\alpha, \neg \alpha\}$ would have to be truth-functionally satisfiable. But this is impossible. ■

Now we define an assignment v by letting $v(\lambda) = 1$ if $\lambda \in \tilde{\Omega}$ and $v(\lambda) = 0$ if $\lambda \notin \tilde{\Omega}$ for every atom λ . We have

Lemma 4. For each \mathcal{A} -formula α , $\alpha^v = 1$ if and only if $\alpha \in \tilde{\Omega}$.

Proof. As we already know, it suffices to restrict ourselves to formulas using the connectives \neg, \vee, \wedge . And, in fact, the De Morgan relation

$$(\beta_1 \vee \beta_2) = \neg(\neg \beta_1 \wedge \neg \beta_2)$$

shows that we can restrict ourselves even further, to the connectives \neg, \wedge . So, we assume that α is an \mathcal{A} -formula expressed in terms of the connectives \neg, \wedge . Our proof will be by induction on the total number of occurrences of these connectives in α .

If this total number is 0, then α is an atom, and the result follows from our definition of v . Otherwise we must have either $\alpha = \neg\beta$ or $\alpha = (\beta \wedge \gamma)$, where by the induction hypothesis we can assume the desired result for β and γ .

Case 1. $\alpha = \neg\beta$

Then, using Lemma 3,

$$\begin{aligned}\alpha^v = 1 &\quad \text{if and only if} \quad \beta^v \neq 1 \\ &\quad \text{if and only if} \quad \beta^v \notin \tilde{\Omega} \\ &\quad \text{if and only if} \quad \alpha \in \tilde{\Omega}.\end{aligned}$$

Case 2. $\alpha = (\beta \wedge \gamma)$

If $\alpha^v = 1$, then $\beta^v = \gamma^v = 1$, so by the induction hypothesis, $\beta, \gamma \in \tilde{\Omega}$. If $\alpha \notin \tilde{\Omega}$, then by Lemma 3, $\neg\alpha \in \tilde{\Omega}$. But the finite set $\{\beta, \gamma, \neg\alpha\}$ is not satisfiable, contradicting Lemma 2. Thus, $\alpha \in \tilde{\Omega}$.

Conversely, if $\alpha \in \tilde{\Omega}$, then neither $\neg\beta$ nor $\neg\gamma$ can belong to $\tilde{\Omega}$, because the finite sets $\{\alpha, \neg\beta\}$, $\{\alpha, \neg\gamma\}$ are not satisfiable. Thus, by Lemma 3, $\beta, \gamma \in \tilde{\Omega}$. By the induction hypothesis $\beta^v = \gamma^v = 1$. Therefore, $\alpha^v = 1$. ■

Now, since $\Omega \subseteq \tilde{\Omega}$, we see that $\alpha^v = 1$ for each $\alpha \in \Omega$. Hence, Ω is truth-functionally satisfiable. Since we began with an arbitrary finitely satisfiable set of \mathcal{A} -formulas Ω , we have proved

Theorem 7.2 (Compactness Theorem for Propositional Calculus). Let Ω be a finitely satisfiable set of \mathcal{A} -formulas. Then Ω is truth-functionally satisfiable.

Exercises

1. Is the set of clauses

$$\{(p_i \vee \neg p_{i+1}) \mid i = 1, 2, 3, \dots\}$$

satisfiable? Why?

2. The same for the set

$$\{(p_i \vee \neg p_{i+1}), (\neg p_i \vee p_{i+1}) \mid i = 1, 2, 3, \dots\}.$$

- 3.* Let us be given a plane map containing infinitely many countries. Suppose there is no way to color this map with k colors so that

adjacent countries are colored with different colors. Prove that there is a finite submap for which the same is true.

- 4.* Let Γ be a (not necessarily finite) set of \mathcal{A} -formulas, and let α be an \mathcal{A} -formula. We can generalize the notion of tautological consequence by writing $\Gamma \vDash \alpha$ to mean this: for every assignment v on \mathcal{A} such that $\gamma^v = 1$ for all $\gamma \in \Gamma$, we also have $\alpha^v = 1$.
- (a) Show that $\Gamma \vDash \alpha$ if and only if $\gamma_1, \dots, \gamma_n \vDash \alpha$ for some $\gamma_1, \dots, \gamma_n \in \Gamma$.
 - (b) Show that if Γ is an r.e. set, then $\{\alpha \mid \Gamma \vDash \alpha\}$ is also r.e.
 - (c) Give an r.e. set Γ such that $\{\alpha \mid \Gamma \vDash \alpha\}$ is not recursive.
 - (d) Let Γ be an r.e. set of \mathcal{A} -formulas such that for some \mathcal{A} -formula α , both $\Gamma \vDash \alpha$ and $\Gamma \vDash \neg \alpha$. Show that $\{\alpha \mid \Gamma \vDash \alpha\}$ is recursive.
 - (e) Let Γ be an r.e. set of \mathcal{A} -formulas such that for every \mathcal{A} -formula α , either $\Gamma \vDash \alpha$ or $\Gamma \vDash \neg \alpha$ but not both. Show that $\{\alpha \mid \Gamma \vDash \alpha\}$ is recursive.

13

Quantification Theory

1. The Language of Predicate Logic

Although a considerable part of logical inference is contained in the propositional calculus, it is only with the introduction of the apparatus of *quantifiers* that one can encompass the full scope of logical deduction as it occurs in mathematics, and in science generally. We begin with an alphabet called a *vocabulary* consisting of two kinds of symbols, *relation symbols* and *function symbols*. Let \mathbf{W} be a vocabulary. For each symbol $t \in \mathbf{W}$, we assume there is an integer $\delta(t)$ called the *degree of* t . For t a function symbol, $\delta(t) \geq 0$, while for t a relation symbol, $\delta(t) > 0$. A function symbol t whose degree is 0 is also called a *constant symbol*. We assume that \mathbf{W} contains at least one relation symbol. (What we are calling a *vocabulary* is often called a *language* in the literature of mathematical logic. Obviously this terminology is not suitable for a book on theoretical computer science.) In addition to \mathbf{W} we shall use the alphabet

$$Q = \{\neg, \wedge, \vee, \supset, \Leftrightarrow, \forall, \exists, (,), x, y, z, u, v, w, I, ,\},$$

where the boldface comma $,$ is one of the symbols that belong to Q . The words that belong to the language

$$\{x^{[i]}, y^{[i]}, z^{[i]}, u^{[i]}, v^{[i]}, w^{[i]} \mid i \in N\}$$

are called *variables*. Again we think of strings of the form $\mathbf{I}^{[i]}$, $i > 0$, as subscripts, e.g., writing x_5 for $x\mathbf{I}\mathbf{I}\mathbf{I}\mathbf{I}$. By a **W-term** (or when the vocabulary **W** is understood, simply a *term*) we mean an element of $(Q \cup W)^*$ that either is a constant symbol $c \in W$ or a variable, or is obtained from constant symbols and variables by repeated application of the operation on $(Q \cup W)^*$ that transforms $\mu_1, \mu_2, \dots, \mu_n$ into

$$f(\mu_1, \mu_2, \dots, \mu_n),$$

where f is a function symbol in **W** and $\delta(f) = n > 0$.

An *atomic W-formula* is an element of $(Q \cup W)^*$ of the form

$$r(\mu_1, \mu_2, \dots, \mu_n),$$

where $r \in W$ is a relation symbol, $\delta(r) = n$, and $\mu_1, \mu_2, \dots, \mu_n$ are terms. Finally, a **W-formula** (or simply a *formula*) is either an atomic **W-formula** or is obtained from atomic **W-formulas** by repeated application of the following operations on $(Q \cup W)^*$:

1. transform α into $\neg \alpha$;
2. transform α and β into $(\alpha \wedge \beta)$;
3. transform α and β into $(\alpha \vee \beta)$;
4. transform α and β into $(\alpha \supset \beta)$;
5. transform α and β into $(\alpha \leftrightarrow \beta)$;
6. transform α into $(\forall b)\alpha$, where b is a variable;
7. transform α into $(\exists b)\alpha$, where b is a variable.

If b is a variable, the expressions

$$(\forall b) \quad \text{and} \quad (\exists b)$$

are called *universal quantifiers* and *existential quantifiers*, respectively.

Let b be a variable, let λ be a formula or a term, and suppose that we have the decomposition $\lambda = rbs$, where the leftmost symbol of s is not **I**. (This means that b is not part of a longer variable. In fact, because λ is a formula or a term, s will have to begin either with $,$ or with $)$.) Then we say that the variable b *occurs in* λ . If more than one such decomposition is possible for a given variable b we speak, in an obvious sense, of the *first occurrence of b in λ* , *the second occurrence of b in λ* , etc., reading from left to right.

Next suppose that α is a formula and that we have the decomposition

$$\alpha = r(\forall b)\beta s \quad \text{or} \quad \alpha = r(\exists b)\beta s,$$

where β is itself a formula. Then *the occurrence of b in the quantifiers shown, as well as all occurrences of b in β , are called bound occurrences of b in α* . Any occurrence of b in α that is not bound is called a **free**

occurrence of b in α . A **W-formula** α containing no free occurrences of variables is called a **W-sentence**, or simply a *sentence*. Any occurrence of a variable in a term is considered to be a *free occurrence*.

Thus, in the formula

$$(r(x) \supset (\exists y)s(u, y)),$$

x and u each have one occurrence, and it is free; y has two occurrences, and they are both bound. The formula

$$(\forall x)(\exists u)(r(x) \supset (\exists y)s(u, y))$$

is a sentence.

Exercises

- Let $\mathbf{W} = \{0, s, <\}$, where $0, s$ are function symbols with $\delta(0) = 0$, $\delta(s) = 1$, and $<$ is a relation symbol with $\delta(<) = 2$. Describe the set of **W-terms** and the set of atomic **W-formulas**.
- (a) Define the *height* of a **W-term** t , denoted $\text{Ht}(t)$, as follows:

$$\text{Ht}(x) = 1 \quad \text{for all variables } x$$

$$\text{Ht}(c) = 1 \quad \text{for all constant symbols } c$$

$$\text{Ht}(f(t_1, \dots, t_n)) = \max\{\text{Ht}(t_i) \mid 1 \leq i \leq n\} + 1.$$

Show by induction on height that all **W-terms** have an equal number of left and right parentheses.

- (b) Do the same for **W-formulas**.

2. Semantics

In analogy with the propositional calculus, we wish to associate the truth values, 1 and 0, with sentences. To do this for a given sentence α will require an “interpretation” of the function and relation symbols in α .

By an *interpretation I of a vocabulary W*, we mean a nonempty set D , called the *domain of I*, together with the following:

- an element c_I of D , for each constant symbol $c \in \mathbf{W}$;
- a function f_I from $D^{\delta(f)}$ into¹ D , for each function symbol $f \in \mathbf{W}$ for which $\delta(f) > 0$; and
- a function r_I from $D^{\delta(r)}$ into $\{0, 1\}$, for each relation symbol $r \in \mathbf{W}$.

¹ Recall from Chapter 1, Section 1, that D^n is the set of n -tuples of elements of D .

Let λ be a *term* or a *formula* and let b_1, b_2, \dots, b_n be a *list of distinct variables which includes all the variables that have free occurrences in λ* . Then, we write $\lambda = \lambda(b_1, \dots, b_n)$ as a *declaration* of our intention to regard b_1, \dots, b_n as acting like parameters taking on values. In such a case, if t_1, \dots, t_n are terms containing no occurrences of variables that have bound occurrences in λ , we write $\lambda(t_1, \dots, t_n)$ for the term or formula obtained from λ by *simultaneously* replacing b_1 by t_1 , b_2 by t_2, \dots, b_n by t_n .

Now let t be a **W-term**, $t = t(b_1, b_2, \dots, b_n)$, and let I be an interpretation of **W**, with domain **D**. Then we shall define a value $t'[d_1, d_2, \dots, d_n] \in D$ for all $d_1, d_2, \dots, d_n \in D$. For the case $n = 0$, we write simply t' . We define this notion recursively as follows:

1. If $t = t(b_1, b_2, \dots, b_n)$ and t is a variable, then t must be b_i for some $i, 1 \leq i \leq n$, and we define $t'[d_1, d_2, \dots, d_n] = d_i$;
2. If $t = t(b_1, b_2, \dots, b_n)$ and t is a constant symbol c in **W**, then we define $t'[d_1, d_2, \dots, d_n] = c$;
3. If $t = t(b_1, b_2, \dots, b_n) = g(t_1, t_2, \dots, t_m)$, where g is a function symbol in **W**, $\delta(g) = m > 0$, then we first set $t_i = t_i(b_1, b_2, \dots, b_n)$, $i = 1, 2, \dots, m$, and we let $s_i = t'_i[d_1, d_2, \dots, d_n]$, $i = 1, 2, \dots, m$. Finally, we define

$$t'[d_1, d_2, \dots, d_n] = g_I(s_1, s_2, \dots, s_m).$$

Continuing, if α is a **W-formula**, $\alpha = \alpha(b_1, b_2, \dots, b_n)$, and I is an interpretation of **W** with domain **D**, we shall define a value $\alpha'[d_1, d_2, \dots, d_n] \in \{0, 1\}$, for all $d_1, d_2, \dots, d_n \in D$. Again, in the particular case $n = 0$ (which can happen only if α is a sentence), we simply write α' . The recursive definition is as follows:

1. If $\alpha = \alpha(b_1, b_2, \dots, b_n) = r(t_1, t_2, \dots, t_m)$, where r is a relation symbol in **W**, $\delta(r) = m$, then we first set $t_i = t_i(b_1, b_2, \dots, b_n)$, $i = 1, 2, \dots, m$, and then let $s_i = t'_i[d_1, d_2, \dots, d_n]$, $i = 1, 2, \dots, m$. Finally, we define $\alpha'[d_1, d_2, \dots, d_n] = r_I(s_1, s_2, \dots, s_m)$.

In 2–6 which follow, let $\beta = \beta(b_1, \dots, b_n)$, $\gamma = \gamma(b_1, \dots, b_n)$, where we assume that $\beta'[d_1, \dots, d_n] = k$, $\gamma'[d_1, \dots, d_n] = l$ with $k, l \in \{0, 1\}$, are already defined for all $d_1, d_2, \dots, d_n \in D$:

2. If α is $\neg \beta$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha'[d_1, \dots, d_n] = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k = 1. \end{cases}$$

3. If α is $(\beta \wedge \gamma)$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha' [d_1, \dots, d_n] = \begin{cases} 1 & \text{if } k = l = 1 \\ 0 & \text{otherwise.} \end{cases}$$

4. If α is $(\beta \vee \gamma)$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha' [d_1, \dots, d_n] = \begin{cases} 0 & \text{if } k = l = 0 \\ 1 & \text{otherwise.} \end{cases}$$

5. If α is $(\beta \supset \gamma)$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha' [d_1, \dots, d_n] = \begin{cases} 0 & \text{if } k = 1 \text{ and } l = 0 \\ 1 & \text{otherwise.} \end{cases}$$

6. If α is $(\beta \Leftrightarrow \gamma)$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha' [d_1, \dots, d_n] = \begin{cases} 1 & \text{if } k = l \\ 0 & \text{otherwise.} \end{cases}$$

In 7 and 8 let $\beta = \beta(b_1, \dots, b_n, b)$, where we assume that $\beta' [d_1, \dots, d_n, e]$ is already defined for all $d_1, \dots, d_n, e \in D$:

7. If α is $(\forall b)\beta$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha' [d_1, \dots, d_n] = \begin{cases} 1 & \text{if } \beta' [d_1, \dots, d_n, e] = 1 \text{ for all } e \in D \\ 0 & \text{otherwise.} \end{cases}$$

8. If α is $(\exists b)\beta$, $\alpha = \alpha(b_1, \dots, b_n)$, then we define

$$\alpha' [d_1, \dots, d_n] = \begin{cases} 1 & \text{if } \beta' [d_1, \dots, d_n, e] = 1 \text{ for some } e \in D \\ 0 & \text{otherwise.} \end{cases}$$

It is important to be aware of the entirely nonconstructive nature of 7 and 8 of this definition. When the set D is infinite, the definition provides no algorithm for carrying out the required searches, and, indeed, in many important cases no such algorithm exists.

Let us consider some simple examples.

EXAMPLE 1. $W = \{c, r, s\}$, where c is a constant symbol, and r and s are relation symbols, $\delta(r) = 3$, $\delta(s) = 2$. Let I have the domain $D = \{0, 1, 2, 3, \dots\}$, let $c_I = 0$, and let

$$r_I(x, y, z) = \begin{cases} 1 & \text{if } x + y = z \\ 0 & \text{otherwise,} \end{cases} \quad s_I(x, y) = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise.} \end{cases}$$

If α is the sentence

$$(\forall x)(\forall y)(\forall z)(r(x, y, z) \supset s(x, z)),$$

then it is easy to see that $\alpha' = 1$. For if $u, v, w \in D$ and $r_I(u, v, w) = 1$, then $u + v = w$, so that $u \leq w$ and therefore $s_I(u, w) = 1$. So if $\gamma = \gamma(x, y, z)$ is the formula $(r(x, y, z) \supset s(x, z))$, then $\gamma'[u, v, w] = 1$.

On the other hand, if β is the sentence

$$(\forall x)(\exists y)r(x, y, c),$$

then $\beta' = 0$. This is because $r_I(1, v, 0) = 0$ for all $v \in D$. Therefore

$$r(x, y, c)'[1, v] = 0$$

for all $v \in D$. Thus, $(\exists y)r(x, y, c)'[1] = 0$, and therefore, finally, $\beta' = 0$.

EXAMPLE 2. \mathbf{W} , α , β are as in Example 1. I has the domain

$$\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\},$$

the set of all integers. c_I, r_I, s_I are defined as in Example 1. In this case, it is easy to see that $\alpha' = 0$ and $\beta' = 1$.

An interpretation I of the vocabulary \mathbf{W} is called a *model* of a \mathbf{W} -sentence α if $\alpha' = 1$; I is called a *model* of the set Ω of \mathbf{W} -sentences if I is a model of each $\alpha \in \Omega$. Ω is said to be *satisfiable* if it has at least one model. An individual \mathbf{W} -sentence α is called *satisfiable* if $\{\alpha\}$ is satisfiable, i.e., if α has a model. α is called *valid* if every interpretation of \mathbf{W} is a model of α .

If $\alpha = \alpha(b_1, \dots, b_n)$, $\beta = \beta(b_1, \dots, b_n)$ are \mathbf{W} -formulas, we write $\alpha = \beta$ to mean that α and β are *semantically equivalent*, that is,

$$\alpha'[d_1, \dots, d_n] = \beta'[d_1, \dots, d_n]$$

for all interpretations I of \mathbf{W} and all $d_1, \dots, d_n \in D$, the domain of I . Then, as is readily verified, all of the equations from Section 1 of Chapter 12 hold true as well in the present context. We also note the *quantificational De Morgan laws*:

$$\neg(\forall b)\alpha = (\exists b)\neg\alpha; \neg(\exists b)\alpha = (\forall b)\neg\alpha. \quad (2.1)$$

Again, as in the case of the propositional calculus, we may eliminate the connectives \supset and \leftrightarrow by using appropriate equations from Chapter 12, Section 1. Once again, there is a “general principle of duality,” but we omit the details.

Now, let $\beta = \beta(b_1, \dots, b_n, b)$, and let the variable a have no occurrences in β . Then it is quite obvious that

$$\begin{aligned} (\exists b)\beta(b_1, \dots, b_n, b) &= (\exists a)\beta(b_1, \dots, b_n, a), \\ (\forall b)\beta(b_1, \dots, b_n, b) &= (\forall a)\beta(b_1, \dots, b_n, a). \end{aligned} \quad (2.2)$$

Continuing to assume that a has no occurrences in β , we have

$$\begin{aligned} ((\forall a)\alpha \wedge \beta) &= (\forall a)(\alpha \wedge \beta), \\ ((\exists a)\alpha \wedge \beta) &= (\exists a)(\alpha \wedge \beta), \\ ((\forall a)\alpha \vee \beta) &= (\forall a)(\alpha \vee \beta), \\ ((\exists a)\alpha \vee \beta) &= (\exists a)(\alpha \vee \beta). \end{aligned} \quad (2.3)$$

Exercises

- Let \mathbf{W} be as in Example 1. For each of the following \mathbf{W} -sentences give an interpretation that is a model of the sentence as well as one that is not.
 - $(\forall x)(\exists y)(\forall z)(s(x, c) \supset r(x, y, z))$.
 - $(\exists y)(\forall x)(\forall z)(s(x, c) \supset r(x, y, z))$.
 - $(\forall x)(\forall y)(s(x, y) \supset s(y, x))$.
- Give an interpretation that is a model of (a) in Exercise 1 but not of (b).
- Let $\mathbf{W} = \{ca, cb, cat, eq\}$, let interpretation I have domain $\{a, b\}^*$, and let $ca_I = a$, $cb_I = b$, $cat_I(u, v) = \widehat{uv}$, and

$$eq_I(u, v) = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{otherwise.} \end{cases}$$

For each of the following formulas α , calculate α' .

- $(\forall x)(\exists y)eq(cat(ca, x), y)$.
 - $(\exists y)(\forall x)eq(cat(ca, x), y)$.
 - $(\forall x)(\exists y)eq(cat(x, y), x)$.
 - $(\forall x)(\exists y)(eq(cat(ca, y), x) \vee eq(cat(cb, y), x))$.
 - $(\exists x)eq(cat(ca, x), cat(x, cb))$.
- For each of the following formulas, tell whether it is (i) satisfiable, (ii) valid, (iii) unsatisfiable.
 - $((\exists x)p(x) \wedge (\forall y)\neg p(y))$.
 - $(\forall x)(\exists y)r(f(a), b)$.

- (c) $((\forall x)(\exists y)r(x, y) \supset (\exists y)(\forall x)r(x, y)).$
 (d) $((\exists y)(\forall x)r(x, y) \supset (\forall x)(\exists y)r(x, y)).$
 (e) $(\exists x)(\forall y) < (x, y).$
 (f) $((\exists x)p(x) \supset (\exists x)(\forall y)p(x)).$
5. Let $\mathbf{W} = \{0, s, +, eq\}$, let interpretation I have domain N , and let $0_I = 0$, s_I be the successor function, $+_I$ be the addition function, and eq_I be equality (as in Exercise 3). For each of the following sets S , give a formula α such that
- $$S = \{(m_1, \dots, m_n) \in N^n \mid \alpha^I[m_1, \dots, m_n] = 1\}.$$
- (a) $S = N.$
 (b) $S = \{(x, y, z) \in N^3 \mid x + y = z\}.$
 (c) $S = \{(x, y) \in N^2 \mid x \leq y\}.$
 (d) $S = \{(x, y, z) \in N^3 \mid z - y = x\}.$
 (e) $S = \{x \in N \mid x \text{ is even}\}.$
6. For a set of sentences Ω , let $Mod(\Omega)$ be the collection of all models of Ω . Prove that

$$\Omega_1 \subseteq \Omega_2 \quad \text{implies} \quad Mod(\Omega_2) \subseteq Mod(\Omega_1).$$

3. Logical Consequence

We are now ready to use the *semantics* just developed to define the notion of *logical consequence*. Let \mathbf{W} be a vocabulary, let Γ be a set of \mathbf{W} -sentences, and let γ be a \mathbf{W} -sentence. Then we write

$$\Gamma \vDash \gamma$$

and call γ a *logical consequence* of the *premises* Γ if every model of Γ is also a model of γ . If $\Gamma = \{\gamma_1, \dots, \gamma_n\}$, then we omit the braces $\{ , \}$, and write simply

$$\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma.$$

Note that $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \gamma$ if and only if for every interpretation I of \mathbf{W} for which

$$\gamma_1^I = \gamma_2^I = \cdots = \gamma_n^I = 1,$$

we also have $\gamma^I = 1$. (Intuitively, we may think of the various interpretations as “possible worlds.”) Then our definition amounts to saying that γ is a logical consequence of some premises if γ is true in every possible world

in which the premises are all true.) As in the case of the propositional calculus, logical consequence can be determined by considering a single sentence. The proof of the corresponding theorem is virtually identical to that of Theorem 2.1 in Chapter 12 and is omitted.

Theorem 3.1. The relation $\gamma_1, \gamma_2, \dots, \gamma_n \models \gamma$ is equivalent to each of the following:

1. the sentence $((\gamma_1 \wedge \dots \wedge \gamma_n) \supset \gamma)$ is valid;
2. the sentence $(\gamma_1 \wedge \dots \wedge \gamma_n \wedge \neg \gamma)$ is unsatisfiable.

Once again we are led to a problem of satisfiability. We will focus our efforts on computational methods for demonstrating the unsatisfiability of a given sentence. We begin by showing how to obtain a suitable normal form for any given sentence.

As in Chapter 12, Section 3, we begin with the procedures

- (I) ELIMINATE \supset and \leftrightarrow .
- (II) MOVE \neg INWARD.

Procedure (I) is carried out exactly as in Chapter 12. For (II), we also need to use the quantificational De Morgan laws (2.1). Ultimately all \neg s will come to immediately precede relation symbols.

(III) RENAME VARIABLES.

Rename bound variables as necessary to ensure that no variables occur in two different quantifiers, using (2.2). Thus, the sentence

$$((\forall x)(\forall y)r(x, y) \vee ((\forall x)s(x) \wedge (\exists y)s(y)))$$

might become

$$(\forall x)(\forall y)r(x, y) \vee ((\forall u)s(u) \wedge (\exists v)s(v)).$$

(IV) PULL QUANTIFIERS

Using (2.3), bring all quantifiers to the left of the sentence. *Where possible, do so with existential quantifiers preceding universal quantifiers.* Thus, to continue our example, we would get successively

$$\begin{aligned} & (\forall x)(\forall y)r(x, y) \vee (\exists v)((\forall u)s(u) \wedge s(v)) \\ &= (\exists v)((\forall x)(\forall y)r(x, y) \vee ((\forall u)s(u) \wedge s(v))) \\ &= (\exists v)(\forall x)(\forall y)(\forall u)(r(x, y) \vee (s(u) \wedge s(v))). \end{aligned}$$

After applying (IV) as many times as possible, we obtain a sentence consisting of *a string of quantifiers followed by a formula containing no*

quantifiers. Such a sentence is called a *prenex sentence*. A prenex sentence is also said to be in *prenex normal form*.

Let γ be a sentence of the form

$$(\forall b_1) \cdots (\forall b_n)(\exists b)\alpha,$$

where $n \geq 0$, and $\alpha = \alpha(b_1, b_2, \dots, b_n, b)$. Let g be a function symbol which is not in α with $\delta(g) = n$. If necessary, we enlarge the vocabulary \mathbf{W} to include this new symbol g . Then we write

$$\gamma_s = (\forall b_1) \cdots (\forall b_n)\alpha(b_1, b_2, \dots, b_n, g(b_1, b_2, \dots, b_n)).$$

γ_s is called the *Skolemization* of γ . [In the case $n = 0$, g is a constant symbol and the term

$$g(b_1, b_2, \dots, b_n)$$

is to be simply understood as standing for g .] Skolemization is important because of the following theorem.

Theorem 3.2. Let γ be a \mathbf{W} -sentence and let γ_s be its Skolemization. Then

1. every model of γ_s is a model of γ ;
2. if γ has a model, then so does γ_s ;
3. γ is satisfiable if and only if γ_s is satisfiable.

Proof. Condition 3 obviously follows from 1 and 2.

To prove 1, let α, γ be as previously and let

$$\beta = \beta(b_1, \dots, b_n) = \alpha(b_1, \dots, b_n, g(b_1, \dots, b_n)).$$

Let I be a model of γ_s so that $\gamma_s^I = 1$, and let the domain of I be D . Then, if d_1, \dots, d_n are arbitrary elements of D , we have $\beta^I[d_1, \dots, d_n] = 1$. Let $e = g_I(d_1, \dots, d_n)$. Thus $\alpha^I[d_1, \dots, d_n, e] = 1$, so that

$$(\exists b)\alpha(b_1, \dots, b_n, b)^I[d_1, \dots, d_n] = 1.$$

Hence finally, $\gamma^I = 1$.

To prove 2, let $\gamma^I = 1$, where I has the domain D . Again let d_1, \dots, d_n be any elements of D . Then, writing β for the formula $(\exists b)\alpha$, so that we may write $\beta = \beta(b_1, \dots, b_n)$, we have $\beta^I[d_1, \dots, d_n] = 1$. Thus, there is an element $e \in D$ such that $\alpha^I[d_1, \dots, d_n, e] = 1$. Hence, we have shown that for each $d_1, \dots, d_n \in D$, there is at least one element $e \in D$ such that $\alpha^I[d_1, \dots, d_n, e] = 1$. Thus, we may extend the interpretation I to

the new function symbol g by defining $g_I(d_1, \dots, d_n)$ to be such an element² e , for each $d_1, \dots, d_n \in D$. Thus, for all $d_1, d_2, \dots, d_n \in D$, we have

$$\beta'[d_1, \dots, d_n] = \alpha'[d_1, \dots, d_n, g_I(d_1, \dots, d_n)] = 1.$$

Hence, finally, $\gamma_s^I = 1$. ■

Since Theorem 3.2 shows that the leftmost existential quantifier in a prenex formula may be eliminated without affecting satisfiability, we can, by iterated Skolemization, obtain a sentence containing no existential quantifiers. We write this

(V) ELIMINATE EXISTENTIAL QUANTIFIERS.

In the example discussed under (IV), this would yield simply

$$(\forall x)(\forall y)(\forall u)(r(x, y) \vee (s(u) \wedge s(c))), \quad (3.1)$$

where c is a constant symbol.

For another example consider the sentence

$$(\forall x)(\exists u)(\forall y)(\forall z)(\exists v)r(x, y, z, u, v),$$

where r is a relation symbol, $\delta(r) = 5$. Then two Skolemizations yield

$$(\forall x)(\forall y)(\forall z)r(x, y, z, g(x), h(x, y, z)). \quad (3.2)$$

A sentence α is called *universal* if it has the form $(\forall b_1)(\forall b_2) \cdots (\forall b_n)\gamma$, where the formula γ contains no quantifiers. We may summarize the procedure (I)–(V) in

Theorem 3.3. There is an algorithm that will transform any given sentence β into a universal sentence α such that β is satisfiable if and only if α is satisfiable. Moreover, any model of α is also a model of β .

In connection with our procedure (I)–(V) consider the example

$$((\forall x)(\exists y)r(x, y) \wedge (\forall u)(\exists v)s(u, v)),$$

where r and s are relation symbols. By varying the order in which the quantifiers are pulled, we can obtain the prenex sentences

1. $(\forall x)(\exists y)(\forall u)(\exists v)(r(x, y) \wedge s(u, v)),$
2. $(\forall u)(\exists v)(\forall x)(\exists y)(r(x, y) \wedge s(u, v)),$

² Here we are using a nonconstructive set-theoretic principle known as the *axiom of choice*.

$$3. (\forall x)(\forall u)(\exists y)(\exists v)(r(x, y) \wedge s(u, v)).$$

Skolemizations will then yield the corresponding universal sentences:

1. $(\forall x)(\forall u)(r(x, g(x)) \wedge s(u, h(x, u))),$
2. $(\forall u)(\forall x)(r(x, g(u, x)) \wedge s(u, h(u))),$
3. $(\forall x)(\forall u)(r(x, g(x, u)) \wedge s(u, h(x, u))).$

But, for this example, one would expect that y should “depend” only on x and v only on u . In other words, we would expect to be able to use a universal sentence such as

$$4. (\forall x)(\forall u)(r(x, g(x)) \wedge s(u, h(u))).$$

As we shall see, it is important to be able to justify such simplifications.

Proceeding generally, let γ be a sentence of the form

$$\delta \wedge (\forall b_1) \cdots (\forall b_n)(\exists b)\alpha,$$

where $n \geq 0$ and $\alpha = \alpha(b_1, b_2, \dots, b_n, b)$. Let g be a function symbol which does not occur in γ with $\delta(g) = n$. Then we write

$$\gamma_s = \delta \wedge (\forall b_1) \cdots (\forall b_n)\alpha(b_1, \dots, b_n, g(b_1, \dots, b_n)).$$

γ_s is called a *generalized Skolemization* of γ . Then we have the following generalization of Theorem 3.2.

Theorem 3.4. Let γ be a W-sentence and let γ_s be a generalized Skolemization of γ . Then we have 1–3 of Theorem 3.2.

Proof. Again we need verify only 1 and 2. Let α, γ, δ be as above. To prove 1, let I be a model of γ_s with domain D . Let β be defined as in the proof of Theorem 3.2. Then $\delta^I = 1$ and $(\forall b_1) \cdots (\forall b_n)\beta^I = 1$. As in the proof of Theorem 3.2, we conclude that

$$(\forall b_1) \cdots (\forall b_n)(\exists b)\alpha^I = 1,$$

and so $\gamma^I = 1$.

Conversely, let $\gamma^I = 1$, where I has domain D . Then $\delta^I = 1$ and

$$(\forall b_1) \cdots (\forall b_n)(\exists b)\alpha^I = 1.$$

Precisely as in the proof of Theorem 3.2, we can extend the interpretation I to the symbol g in such a way that

$$(\forall b_1) \cdots (\forall b_n)\beta^I = 1.$$

Hence, $\gamma_s^I = 1$. ■

Henceforth we will consider the steps (IV) PULL QUANTIFIERS and (V) ELIMINATE EXISTENTIAL QUANTIFIERS to permit the use of generalized Skolemizations. Moreover, as we have seen, Theorem 3.3 remains correct if the universal sentence is obtained using generalized Skolemizations.

Exercises

1. Consider the inference

$$\begin{aligned} & (\forall x)(p(x) \supset (\forall y)(s(y, x) \supset u(x))), \\ & (\exists x)(p(x) \wedge (\exists y)(s(y, x) \wedge h(y, x))) \\ & \models (\exists x)(\exists y)(u(x) \wedge h(y, x) \wedge s(y, x)). \end{aligned}$$

- (a) Find a universal sentence whose unsatisfiability is equivalent to the correctness of this inference. Can you do this so that Skolemization introduces only constant symbols?
 - (b) Using (a), show that the inference is correct.
2. (a) Using generalized Skolemization find a universal sentence whose unsatisfiability is equivalent to the correctness of the inference

$$(\exists x)(\forall y)r(x, y) \models (\forall y)(\exists x)r(x, y).$$

- (b) Show that the inference is correct.
3. The same as Exercise 2(a) for the inference
- $$\begin{aligned} & (\forall x)(\forall y)(\forall z)(\forall u)(\forall v)(\forall w)((P(x, y, u) \wedge P(y, z, v) \wedge P(x, v, w)) \\ & \quad \supset P(u, z, w)), \\ & (\forall x)(\forall y)(\exists z)P(z, x, y), \\ & (\forall x)(\forall y)(\exists z)P(x, z, y) \models (\exists x)(\forall y)P(y, x, y). \end{aligned}$$

4. Prove Theorem 3.1.
5. For each sentence α in Exercise 2.1, perform the following.
- (a) Transform α into a prenex normal form sentence.
 - (b) Give the Skolemization γ_s of γ .
 - (c) Give a model I of α .
 - (d) Extend I to a model of γ_s .
6. Let γ be a W-sentence, for some vocabulary W , and let γ_s be its Skolemization. Prove or disprove each of the following statements.
- (a) If γ is valid then γ_s is valid.
 - (b) If γ_s is valid then γ is valid.

7. Let \mathbf{W} be a vocabulary, Γ a set of \mathbf{W} -sentences, and α, β \mathbf{W} -sentences. Prove each of the following statements.
- (Deduction Theorem) $\Gamma \cup \{\alpha\} \vDash \beta$ if and only if $\Gamma \vDash (\alpha \supset \beta)$.
 - (Contraposition) $\Gamma \cup \{\alpha\} \vDash \neg \beta$ if and only if $\Gamma \cup \{\beta\} \vDash \neg \alpha$.
 - (Reductio ad absurdum) $\Gamma \cup \{\alpha\} \vDash (\beta \wedge \neg \beta)$ if and only if $\Gamma \vDash \neg \alpha$.

4. Herbrand's Theorem

We have seen that the problem of logical inference is reducible to the problem of satisfiability, which in turn is reducible to the problem of satisfiability of universal sentences. In this section, we will prove Herbrand's theorem, which can be used together with algorithms for truth-functional satisfiability (discussed in Chapter 12) to develop procedures for this purpose.

Let α be a *universal* \mathbf{W} -sentence for some vocabulary \mathbf{W} , where we assume that α contains all the symbols in \mathbf{W} . If α contains at least one constant symbol, we call the set of all constant symbols in α the *constant set* of α . If α contains no constant symbols, we let a be some new constant symbol, which we add to \mathbf{W} , and we call $\{a\}$ the *constant set* of α . Then the language which consists of all \mathbf{W} -terms containing no variables is called the *Herbrand universe* of α . The set \mathcal{A} of atomic \mathbf{W} -formulas containing no variables is called the *atom set* of α . We will work with the set of *propositional formulas over \mathcal{A}* , i.e., of \mathcal{A} -formulas in the sense of Chapter 12, Section 1. *Each of these \mathcal{A} -formulas is also a \mathbf{W} -sentence that contains no quantifiers.*

Returning to the universal sentence (3.1), we see that its constant set is $\{c\}$, its Herbrand universe is likewise $\{c\}$, and its atom set is $\{r(c, c), s(c)\}$.

Next, examining the universal sentence (3.2), its constant set is $\{a\}$, but its Herbrand universe is infinite:

$$\mathbf{H} = \{a, g(a), h(a, a, a), g(g(a)), g(h(a, a, a)), h(a, a, g(a)), \dots\}.$$

Its atom set is likewise infinite:

$$\mathcal{A} = \{r(t_1, t_2, t_3, t_4, t_5) | t_1, t_2, t_3, t_4, t_5 \in \mathbf{H}\}.$$

Theorem 4.1. Let $\zeta = \zeta(b_1, b_2, \dots, b_n)$ be a \mathbf{W} -formula containing no quantifiers, so that the sentence

$$\gamma = (\forall b_1)(\forall b_2) \cdots (\forall b_n)\zeta$$

is universal. Let \mathbf{H} be the Herbrand universe of γ and let \mathcal{A} be its atom set. Then, γ is satisfiable if and only if the set

$$\Omega = \{\zeta(t_1, t_2, \dots, t_n) | t_1, t_2, \dots, t_n \in \mathbf{H}\} \quad (4.1)$$

of \mathcal{A} -formulas is truth-functionally satisfiable.

Proof. First let γ be satisfiable, say, $\gamma^I = 1$, and let D be the domain of I . We now define an assignment v on \mathcal{A} . Let r be a relation symbol of \mathbf{W} , $\delta(r) = m$, so that $r(t_1, \dots, t_m) \in \mathcal{A}$ for all $t_1, \dots, t_m \in \mathbf{H}$. Then we define

$$v(r(t_1, \dots, t_m)) = r_I(t_1^I, \dots, t_m^I).$$

We have

Lemma 1. For all \mathcal{A} -formulas α , $\alpha^I = \alpha^v$.

Proof. As in Chapter 12, we may assume that α contains only the connectives \neg , \wedge . Proceeding by induction, we see that if α is an atom, the result is obvious from our definition of v . Thus, we may suppose that $\alpha = \neg\beta$ or $\alpha = (\beta \wedge \gamma)$, where the result is known for β or, for β and γ , respectively.

In the first case, we have

$$\begin{array}{lll} \alpha^I = 1 & \text{if and only if} & \beta^I = 0 \\ & \text{if and only if} & \beta^v = 0 \\ & \text{if and only if} & \alpha^v = 1. \end{array}$$

Similarly, in the second case

$$\begin{array}{lll} \alpha^I = 1 & \text{if and only if} & \beta^I = \gamma^I = 1 \\ & \text{if and only if} & \beta^v = \gamma^v = 1 \\ & \text{if and only if} & \alpha^v = 1. \end{array} \blacksquare$$

Returning to the proof of the theorem, we wish to show that for all $\alpha \in \Omega$, $\alpha^v = 1$. By Lemma 1, it will suffice to show that $\alpha^I = 1$ for $\alpha \in \Omega$. Now, since $\gamma^I = 1$, we have

$$\zeta^I[d_1, \dots, d_n] = 1 \quad \text{for all } d_1, \dots, d_n \in D.$$

But clearly, for $t_1, \dots, t_n \in \mathbf{H}$,

$$\zeta(t_1, \dots, t_n)^I = \zeta^I[t_1^I, \dots, t_n^I] = 1.$$

We conclude that Ω is truth-functionally satisfiable.

Conversely, let us be given an assignment v on \mathcal{A} such that $\alpha^v = 1$ for all $\alpha \in \Omega$. We shall use v to construct an interpretation I of \mathbf{W} . The domain of I is simply the Herbrand universe \mathbf{H} . Furthermore,

1. If $c \in \mathbf{W}$ is a constant symbol, then $c_I = c$. (That is, a constant symbol is interpreted as itself.)
2. If $f \in \mathbf{W}$ is a function symbol, $\delta(f) = n > 0$, and $t_1, t_2, \dots, t_n \in \mathbf{H}$, then

$$f_I(t_1, t_2, \dots, t_n) = f(t_1, t_2, \dots, t_n) \in \mathbf{H}.$$

(Note carefully the use of boldface.)

3. If $r \in \mathbf{W}$ is a relation symbol, $\delta(r) = n$, and $t_1, t_2, \dots, t_n \in \mathbf{H}$, then

$$r_I(t_1, t_2, \dots, t_n) = v(r(t_1, t_2, \dots, t_n)).$$

(Note that the assignment v is only used in 3.) We have

Lemma 2. For every $t \in \mathbf{H}$, $t^I = t$.

Proof. Immediate from 1 and 2. ■

Lemma 3. For every \mathbf{W} -formula $\alpha = \alpha(b_1, \dots, b_n)$ containing no quantifiers, and all $t_1, \dots, t_n \in \mathbf{H}$, we have

$$\alpha^I[t_1, \dots, t_n] = v(\alpha(t_1, \dots, t_n)).$$

Proof. If α is an atom, the result follows at once from 3 and Lemma 2. For the general case it now follows because the same recursive rules are used for the propositional connectives, whether we are evaluating interpretations or assignments. ■

Returning to the proof of the theorem, we wish to show that $\gamma^I = 1$. For this, recalling that \mathbf{H} is the domain of I , it suffices to show that

$$\zeta^I[t_1, \dots, t_n] = 1 \quad \text{for all } t_1, \dots, t_n \in \mathbf{H}.$$

By Lemma 3, this amounts to showing that

$$v(\zeta(t_1, \dots, t_n)) = 1 \quad \text{for all } t_1, \dots, t_n \in \mathbf{H}.$$

But this last is precisely what we have assumed about v . ■

The usefulness of the theorem we have just proved results from combining it with the compactness theorem (Theorem 7.2 in Chapter 12).

Theorem 4.2 (Herbrand's Theorem). Let ζ , γ , \mathbf{H} , \mathcal{A} , and Ω be as in Theorem 4.1. Then γ is unsatisfiable if and only if there is a truth-

functionally unsatisfiable **W**-formula of the form $\bigwedge_{\beta \in \Sigma} \beta$ for some finite subset Σ of Ω .

Proof. If there is a truth-functionally unsatisfiable \mathcal{A} -formula $\bigwedge_{\beta \in \Sigma} \beta$, where $\Sigma \subseteq \Omega$, then for every assignment v on \mathcal{A} , there is some $\beta \in \Sigma$ such that $\beta^v = 0$. Hence Σ , and therefore also Ω , is not truth-functionally satisfiable; hence by Theorem 4.1, γ is unsatisfiable.

Conversely, if γ is unsatisfiable, then by Theorem 4.1, Ω is not truth-functionally satisfiable. Thus, by the compactness theorem (Theorem 7.2 in Chapter 12), Ω is not finitely satisfiable; i.e., there is a finite set $\Sigma \subseteq \Omega$ such that Σ is not truth-functionally satisfiable. Then, the sentence $\bigwedge_{\beta \in \Sigma} \beta$ is truth-functionally unsatisfiable. ■

This theorem leads at once to a family of procedures for demonstrating the unsatisfiability of a universal sentence γ . Write $\Omega = \bigcup_{n=0}^{\infty} \Sigma_n$, where $\Sigma_0 = \emptyset$, $\Sigma_n \subseteq \Sigma_{n+1}$, the Σ_n are all finite, and where there is an algorithm that transforms each Σ_n into Σ_{n+1} . (This can easily be managed, e.g., by simply writing the elements of Ω as an infinite sequence.) Then we have the procedure

```

 $n \leftarrow 0$ 
WHILE  $\bigwedge_{\beta \in \Sigma_n} \beta$  IS TRUTH-FUNCTIONALLY SATISFIABLE DO
     $n \leftarrow n + 1$ 
END

```

If γ is unsatisfiable, the procedure will eventually terminate; otherwise it will continue forever. The test for truth-functional satisfiability of $\bigwedge_{\beta \in \Sigma_n} \beta$ can be performed using the methods of Chapter 12, e.g., the Davis–Putnam rules. Using this discussion, we are able to conclude

Theorem 4.3. For every vocabulary **W** the set of unsatisfiable sentences is recursively enumerable. Likewise the set of valid sentences is r.e.

Proof. Given a sentence α , we apply our algorithms to obtain a universal sentence γ that is satisfiable if and only if α is. We then apply the preceding procedure based on Herbrand's theorem. It will ultimately halt if and only if α is unsatisfiable. This procedure shows that the set of unsatisfiable sentences is r.e.

Since a sentence α is valid if and only if $\neg \alpha$ is unsatisfiable, the same procedure shows that the set of valid sentences is r.e. ■

One might have hoped that the set of unsatisfiable **W**-sentences would in fact be recursive. But as we shall see later (Theorem 8.1), this is not the

case. Thus, as we shall see, we *cannot hope* for an algorithm that, beginning with sentences $\gamma_1, \gamma_2, \dots, \gamma_n, \gamma$ as input, will return YES if $\gamma_1, \gamma_2, \dots, \gamma_n \models \gamma$, and NO otherwise. The best we can hope for is a general procedure that will halt and return YES whenever the given logical inference is correct, but that may fail to terminate otherwise. And in fact, using Theorem 3.1 and an algorithm of the kind used in the proof of Theorem 4.3, we obtain just such a procedure.

Now let us consider what is involved in testing the truth-functional satisfiability of $\bigwedge_{\beta \in \Sigma} \beta$, where Σ is a finite subset of the set Ω defined in (4.1). If we wish to use the methods developed in Chapter 12, we need to obtain a CNF of $\bigwedge_{\beta \in \Sigma} \beta$. But, if for each $\beta \in \Sigma$, we have a CNF formula β^0 such that $\beta = \beta^0$, then $\bigwedge_{\beta \in \Sigma} \beta^0$ is clearly a CNF of $\bigwedge_{\beta \in \Sigma} \beta$. This fact makes CNF useful in this context.

In fact we can go further. We can apply the algorithms of Chapter 12, Section 3, to obtain CNF formulas directly for $\zeta = \zeta(b_1, \dots, b_n)$. When we do this we are in effect enlarging the set of formulas to which we apply the methods of Chapter 12, by allowing atoms that contain variables. Each formula can then be thought of as representing all of the W-formulas obtained by replacing each variable by an element of the Herbrand universe \mathbf{H} . In this context formulas containing no variables are called *ground formulas*. We also speak of *ground literals*, *ground clauses*, etc.

If the CNF formula obtained in this manner from $\zeta(b_1, \dots, b_n)$ is given by the set of clauses

$$\{\kappa_i(b_1, \dots, b_n) \mid i = 1, 2, \dots, r\}, \quad (4.2)$$

then each $\beta \in \Sigma$ will have a CNF

$$\{\kappa_i(t_1, \dots, t_n) \mid i = 1, 2, \dots, r\},$$

where t_1, \dots, t_n are suitable elements of \mathbf{H} . Hence, there will be a CNF of $\bigwedge_{\beta \in \Sigma} \beta$ representable in the form

$$\{\kappa_i(t_1^j, \dots, t_n^j) \mid i = 1, \dots, r, j = 1, \dots, s\}, \quad (4.3)$$

where $t_1^j, \dots, t_n^j \in \mathbf{H}$, $j = 1, 2, \dots, s$. Thus, what we are seeking is an unsatisfiable set of clauses of the form (4.3). Of course, such a set can be unsatisfiable without being *minimally* unsatisfiable in the sense of Chapter 12, Section 5. In fact, there is no reason to expect a minimally unsatisfiable set of clauses which contains, say, $\kappa_1(t_1, \dots, t_n)$ to also contain $\kappa_2(t_1, \dots, t_n)$. Thus, we are led to treat the clauses in the set (4.2) independently of one another, seeking substitutions of elements of \mathbf{H} for the variables b_1, \dots, b_n so as to obtain a truth-functionally inconsistent set

R of clauses. Each of the clauses in (4.2) can give rise by substitution to one or more of the clauses of **R**.

Let us consider some examples.

EXAMPLE 1. Consider this famous inference: *All men are mortal; Socrates is a man; therefore, Socrates is mortal.* An appropriate vocabulary would be $\{m, t, s\}$, where m, t are relation symbols of degree 1 (which we think of as standing for the properties of *being a man*, and of *being mortal*, respectively), and s is a constant symbol (which we think of as naming Socrates). The inference becomes

$$(\forall x)(m(x) \supset t(x)), m(s) \vDash t(s).$$

Thus, we wish to prove the unsatisfiability of the sentence

$$((\forall x)(m(x) \supset t(x)) \wedge m(s) \wedge \neg t(s)).$$

Going to prenex form, we see that no Skolemization is needed:

$$(\forall x)((\neg m(x) \vee t(x)) \wedge m(s) \wedge \neg t(s)).$$

The Herbrand universe is just $\{s\}$. In this simple case, Herbrand's theorem tells us that we have to prove the truth-functional unsatisfiability of

$$((\neg m(s) \vee t(s)) \wedge m(s) \wedge \neg t(s));$$

that is, we are led directly to a ground formula in CNF. Using the set representation of Chapter 12, Section 4, we are dealing with the set of clauses

$$\{\{\bar{m}(s), t(s)\}, \{m(s)\}, \{\bar{t}(s)\}\}.$$

Using the Davis–Putnam rules (or, in this case equivalently, resolution), we obtain successively

$$\{\{t(s)\}, \{\bar{t}(s)\}\}, \quad \text{and} \quad \{\square\};$$

hence the original inference was valid.

EXAMPLE 2. Another inference: *Every shark eats a tadpole; all large white fish are sharks; some large white fish live in deep water; any tadpole eaten by a deep water fish is miserable; therefore, some tadpoles are miserable.*

Our vocabulary is $\{s, b, t, r, m, e\}$, where all of these are relation symbols of degree 1, except e , which is a relation symbol of degree 2. $e(x, y)$ is to represent “ x eats y .” s stands for the property of *being a shark*, b of *being a large white fish*, t of *being a tadpole*, r of *living in deep water*, and m of

being miserable. The inference translates as

$$\begin{aligned}
 & (\forall x)(s(x) \supset (\exists y)(t(y) \wedge e(x, y))), \\
 & (\forall x)(b(x) \supset s(x)), \\
 & (\exists x)(b(x) \wedge r(x)), \\
 & (\forall x)(\forall y)((r(x) \wedge t(y) \wedge e(x, y)) \supset m(y)) \vDash (\exists y)(t(y) \wedge m(y)).
 \end{aligned}$$

Thus, we need to demonstrate the unsatisfiability of the sentence

$$\begin{aligned}
 & ((\forall x)(s(x) \supset (\exists y)(t(y) \wedge e(x, y)))) \\
 & \quad \wedge (\forall x)(b(x) \supset s(x)) \\
 & \quad \wedge (\exists x)(b(x) \wedge r(x)) \\
 & \quad \wedge (\forall x)(\forall y)((r(x) \wedge t(y) \wedge e(x, y)) \supset m(y)) \\
 & \quad \wedge \neg(\exists y)(t(y) \wedge m(y))).
 \end{aligned}$$

We proceed as follows.

I. ELIMINATE \supset :

$$\begin{aligned}
 & ((\forall x)(\neg s(x) \vee (\exists y)(t(y) \wedge e(x, y)))) \\
 & \quad \wedge (\forall x)(\neg b(x) \vee s(x)) \\
 & \quad \wedge (\exists x)(b(x) \wedge r(x)) \\
 & \quad \wedge (\forall x)(\forall y)(\neg(r(x) \wedge t(y) \wedge e(x, y)) \vee m(y)) \\
 & \quad \wedge \neg(\exists y)(t(y) \wedge m(y))).
 \end{aligned}$$

II. MOVE \neg INWARD:

$$\begin{aligned}
 & ((\forall x)(\neg s(x) \vee (\exists y)(t(y) \wedge e(x, y)))) \\
 & \quad \wedge (\forall x)(\neg b(x) \vee s(x)) \\
 & \quad \wedge (\exists x)(b(x) \wedge r(x)) \\
 & \quad \wedge (\forall x)(\forall y)(\neg r(x) \vee \neg t(y) \vee \neg e(x, y) \vee m(y)) \\
 & \quad \wedge (\forall y)(\neg t(y) \vee \neg m(y))).
 \end{aligned}$$

III. RENAME VARIABLES;

$$\begin{aligned}
 & ((\forall x)(\neg s(x) \vee (\exists y_1)(t(y_1) \wedge e(x, y_1)))) \\
 & \quad \wedge (\forall z)(\neg b(z) \vee s(z)) \\
 & \quad \wedge (\exists u)(b(u) \wedge r(u)) \\
 & \quad \wedge (\forall v)(\forall w)(\neg r(v) \vee \neg t(w) \vee \neg e(v, w) \vee m(w)) \\
 & \quad \wedge (\forall y)(\neg t(y) \vee \neg m(y))).
 \end{aligned}$$

IV. PULL QUANTIFIERS (trying to pull existential quantifiers first):

$$\begin{aligned}
 & (\exists u)(\forall x)(\exists y_1)(\forall z)(\forall v)(\forall w)(\forall y) \\
 & ((\neg s(x) \vee (t(y_1) \wedge e(x, y_1))) \\
 & \quad \wedge (\neg b(z) \vee s(z)) \\
 & \quad \wedge b(u) \wedge r(u) \\
 & \quad \wedge (\neg r(v) \vee \neg t(w) \vee \neg e(v, w) \vee m(w)) \\
 & \quad \wedge (\neg t(y) \vee \neg m(y))).
 \end{aligned}$$

V. ELIMINATE EXISTENTIAL QUANTIFIERS:

$$\begin{aligned}
 & (\forall x)(\forall z)(\forall v)(\forall w)(\forall y) \\
 & ((\neg s(x) \vee (t(g(x)) \\
 & \quad \wedge e(x, g(x)))) \\
 & \quad \wedge (\neg b(z) \vee s(z)) \\
 & \quad \wedge b(c) \wedge r(c) \\
 & \quad \wedge (\neg r(v) \vee \neg t(w) \vee \neg e(v, w) \vee m(w)) \\
 & \quad \wedge (\neg t(y) \vee \neg m(y))).
 \end{aligned}$$

Thus we are led to the clauses

$$\begin{aligned}
 & \{\bar{s}(x), t(g(x))\}, \\
 & \{\bar{s}(x), e(x, g(x))\}, \\
 & \{\bar{b}(z), s(z)\}, \\
 & \{b(c)\}, \\
 & \{r(c)\}, \\
 & \{\bar{r}(v), \bar{t}(w), \bar{e}(v, w), m(w)\}, \\
 & \{\bar{t}(y), \bar{m}(y)\}.
 \end{aligned}$$

The Herbrand universe is

$$\mathbf{H} = \{c, g(c), g(g(c)), \dots\}.$$

To find substitutions for the variables in \mathbf{H} , we have recourse to Theorem 5.2 (2) in Chapter 12. To search for a minimally unsatisfiable set of ground clauses, we should seek substitutions that will lead to every literal having a mate (in another clause). By inspection, we are led to the substitution

$$x = c, \quad z = c, \quad v = c, \quad w = g(c), \quad y = g(c).$$

We thus obtain the set of ground clauses

$$\begin{aligned} & \{\bar{s}(c), t(g(c))\}, \\ & \{\bar{s}(c), e(c, g(c))\}, \\ & \{\bar{b}(c), s(c)\}, \\ & \{b(c)\}, \\ & \{r(c)\}, \\ & \{\bar{r}(c), \bar{t}(g(c)), \bar{e}(c, g(c)), m(g(c))\}, \\ & \{\bar{t}(g(c)), \bar{m}(g(c))\}. \end{aligned}$$

Although this set of clauses is linked, we must still test for satisfiability. Using the Davis–Putnam rules we obtain, first using the unit rule on $\{b(c)\}$,

$$\begin{aligned} & \{\bar{s}(c), t(g(c))\}, \\ & \{\bar{s}(c), e(c, g(c))\}, \\ & \{s(c)\}, \\ & \{r(c)\}, \\ & \{\bar{r}(c), \bar{t}(g(c)), \bar{e}(c, g(c)), m(g(c))\}, \\ & \{\bar{t}(g(c)), \bar{m}(g(c))\}. \end{aligned}$$

Using the unit rule on $\{s(c)\}$ and then on $\{r(c)\}$ gives

$$\begin{aligned} & \{t(g(c))\}, \\ & \{e(c, g(c))\}, \\ & \{\bar{t}(g(c)), \bar{e}(c, g(c)), m(g(c))\}, \\ & \{\bar{t}(g(c)), \bar{m}(g(c))\}. \end{aligned}$$

Using the unit rule on $\{t(g(c))\}$ and then on $\{e(c, g(c))\}$ gives

$$\begin{aligned} & \{m(g(c))\}, \\ & \{\bar{m}(g(c))\}. \end{aligned}$$

Finally, we obtain the set of clauses consisting of the empty clause:

□.

In Examples 1 and 2 each clause of (4.2) gave rise to just one clause in the truth-functionally unsatisfiable set of clauses obtained. That is, we

obtain a truth-functionally unsatisfiable set of clauses of the form (4.3) with $s = 1$. Our next example will be a little more complicated.

EXAMPLE 3. We consider the inference

$$\begin{aligned} & (\forall x)(\exists y)(r(x, y) \vee r(y, x)), \\ & (\forall x)(\forall y)(r(x, y) \supset r(y, y)) \vDash (\exists z)r(z, z). \end{aligned}$$

Thus, we wish to demonstrate the unsatisfiability of the sentence

$$\begin{aligned} & (\forall x)(\exists y)(r(x, y) \vee r(y, x)) \\ & \quad \wedge (\forall x)(\forall y)(r(x, y) \supset r(y, y)) \wedge \neg(\exists z)r(z, z). \end{aligned}$$

We proceed as follows:

I, II, III. ELIMINATE \supset ; MOVE \neg INWARD; RENAME VARIABLES:

$$\begin{aligned} & (\forall x)(\exists y)(r(x, y) \vee r(y, x)) \\ & \quad \wedge (\forall u)(\forall v)(\neg r(u, v) \vee r(v, v)) \wedge (\forall z)\neg r(z, z). \end{aligned}$$

IV. PULL QUANTIFIERS:

$$\begin{aligned} & (\forall x)(\exists y)(\forall u)(\forall v)(\forall z)((r(x, y) \vee r(y, x)) \\ & \quad \wedge (\neg r(u, v) \vee r(v, v)) \wedge \neg r(z, z)). \end{aligned}$$

V. ELIMINATE EXISTENTIAL QUANTIFIERS:

$$\begin{aligned} & (\forall x)(\forall u)(\forall v)(\forall z)((r(x, g(x)) \vee r(g(x), x)) \\ & \quad \wedge (\neg r(u, v) \vee r(v, v)) \wedge \neg r(z, z)). \end{aligned}$$

We thus obtain the set of clauses

$$\begin{aligned} & \{r(x, g(x)), r(g(x), x)\}, \\ & \{\bar{r}(u, v), r(v, v)\}, \\ & \{\bar{r}(z, z)\}. \end{aligned}$$

The Herbrand universe is

$$\mathbf{H} = \{a, g(a), g(g(a)), \dots\}.$$

How can we find a mate for $r(x, g(x))$? Not by using $\bar{r}(z, z)$ —whichever element $t \in \mathbf{H}$ we substitute for x , $r(x, g(x))$ will become $r(t, g(t))$, which cannot be obtained from $r(z, z)$ by replacing z by any element of \mathbf{H} .

Thus the only potential mate for $r(x, g(x))$ is $\bar{r}(u, v)$. We tentatively set $u = x$, $v = g(x)$ so that the second clause becomes

$$\{\bar{r}(x, g(x)), r(g(x), g(x))\}.$$

But now, $\bar{r}(u, v)$ is also the only available potential mate for $r(g(x), x)$. Thus, we are led to also substitute $v = x$, $u = g(x)$ in the second clause, obtaining

$$\{\bar{r}(g(x), x), r(x, x)\}.$$

Both $r(g(x), g(x))$ and $r(x, x)$ can be matched with $\bar{r}(z, z)$ to produce mates. We thus arrive at the set of clauses

$$\{r(x, g(x)), r(g(x), x)\},$$

$$\{\bar{r}(x, g(x)), r(g(x), g(x))\},$$

$$\{\bar{r}(g(x), x), r(x, x)\},$$

$$\{\bar{r}(x, x)\},$$

$$\{\bar{r}(g(x), g(x))\}.$$

Now we can replace x by any element of \mathbf{H} to obtain a linked set of ground clauses. For example, we can set $x = a$; but any other substitution for x will do. Actually, it is just as easy to work with the nonground clauses as listed, since the propositional calculus processing is quite independent of which element of \mathbf{H} we substitute for x . In fact after four applications of the unit rule (or of resolution) we obtain \square , which shows that the original inference was correct.

Exercises

1. Describe the Herbrand universe and the atom set of the universal sentence obtained in Exercise 3.1.
2. Do the same for Exercise 3.2.
3. Do the same for Exercise 3.3.
4. Let $\mathbf{W} = \{c, f, p\}$, where c is a constant symbol, f is a function symbol with $\delta(f) = 1$, and p is a relation symbol with $\delta(p) = 1$. Show that $\{(\exists x)p(x), \neg p(c), \neg p(f(c)), \neg p(f(f(c))), \dots\}$ is satisfiable.

5. Unification

We continue our consideration of Example 3 of the previous section. Let us analyze what was involved in attempting to “mate” our literals. Suppose we want to mate $r(x, g(x))$ with $\bar{r}(z, z)$. The first step is to observe that both literals have the same relation symbol r , and that r is negated in one and only one of the two literals. Next we were led to the equations

$$x = z, \quad g(x) = z.$$

The first equation is easily satisfied by setting $x = z$. But then the second equation becomes $g(z) = z$, and clearly no substitution from the Herbrand universe can satisfy this equation. Thus, we were led to consider instead the pair of literals $r(x, g(x))$, $\bar{r}(u, v)$. The equations we need to solve are then

$$x = u, \quad g(x) = v.$$

Again we satisfy the first equation by letting $x = u$; the second equation becomes $g(u) = v$, which can be satisfied by letting $v = g(u)$. So the literals become $r(u, g(u))$ and $\bar{r}(u, g(u))$.

This example illustrates the so-called *unification algorithm* for finding substitutions which will transform given literals $r(\lambda_1, \dots, \lambda_n)$, $\bar{r}(\mu_1, \dots, \mu_n)$ into mates of one another. The procedure involves comparing two terms μ, λ and distinguishing four cases:

1. One of μ, λ (say, μ) is a variable and λ does not contain this variable. Then replace μ by λ throughout.
2. One of μ, λ (say, μ) is a variable, $\lambda \neq \mu$, but λ contains μ . Then report: NOT UNIFIABLE.
3. μ, λ both begin with function symbols, but not with the same function symbol. Again report: NOT UNIFIABLE.
4. μ, λ begin with the same function symbol, say

$$\mu = g(\nu_1, \dots, \nu_k), \quad \lambda = g(\eta_1, \dots, \eta_k).$$

Then use this same procedure recursively on the pairs

$$\nu_1 = \eta_1, \quad \nu_2 = \eta_2, \quad \dots, \quad \nu_k = \eta_k.$$

In applying the unification algorithm to

$$r(\lambda_1, \dots, \lambda_n), \quad \bar{r}(\mu_1, \dots, \mu_n),$$

we begin with the pairs of terms

$$\lambda_1 = \mu_1, \quad \lambda_2 = \mu_2, \quad \dots, \quad \lambda_n = \mu_n$$

and apply the preceding procedure to each. Naturally, substitutions called for by step 1 must be made in all of the terms before proceeding.

To see that the process always terminates, it is necessary to note only that whenever step 1 is applied, the total number of variables present decreases.

EXAMPLE Let us attempt to unify

$$r(g(x), y, g(g(z))) \quad \text{with} \quad \bar{r}(u, g(u), g(v)).$$

We are led to the equations

$$g(x) = u, \quad y = g(u), \quad g(g(z)) = g(v).$$

The first equation leads to letting

$$u = g(x),$$

and the remaining equations then become

$$y = g(g(x)) \quad \text{and} \quad g(g(z)) = g(v).$$

The second is satisfied by letting

$$y = g(g(x)),$$

which does not affect the third equation. The third equation leads recursively to

$$g(z) = v,$$

which is satisfied by simply setting v equal to the left side of this equation. The final result is

$$r(g(x), g(g(x)), g(g(z))), \quad \bar{r}(g(x), g(g(x)), g(g(z))).$$

Numerous systematic procedures for showing sentences to be unsatisfiable based on the unification algorithm have been studied. These procedures work directly with clauses containing variables and do not require that substitutions from the Herbrand universe actually be carried out. In particular, there are *linked conjunct procedures* that are based on searches for a linked set of clauses, followed by a test for truth-functional unsatisfiability. However, most computer implemented procedures have been based on *resolution*. In these procedures, when a pair of literals have been mated by an appropriate substitution, they are immediately eliminated by resolution. We illustrate the use of resolution on Examples 2 and 3 of the previous section.

Beginning with the clauses of Example 2, applying the unification algorithm to the pair of literals $s(z), \bar{s}(x)$, and then using resolution, we get

$$\begin{aligned} & \{\bar{b}(x), t(g(x))\}, \\ & \{\bar{b}(x), e(x, g(x))\}, \\ & \{b(c)\}, \\ & \{r(c)\}, \\ & \{\bar{r}(v), \bar{i}(w), \bar{e}(v, w), m(w)\}, \\ & \{\bar{i}(y), \bar{m}(y)\}. \end{aligned}$$

Next, unifying

$$e(x, g(x)) \quad \text{and} \quad \bar{e}(v, w)$$

and using resolution, we get

$$\begin{aligned} & \{\bar{b}(x), t(g(x))\}, \\ & \{b(c)\}, \\ & \{r(c)\}, \\ & \{\bar{b}(x), \bar{r}(x), \bar{i}(g(x)), m(g(x))\}, \\ & \{\bar{i}(y), \bar{m}(y)\}. \end{aligned}$$

Another stage of unification and resolution yields

$$\begin{aligned} & \{t(g(c))\}, \\ & \{r(c)\}, \\ & \{\bar{r}(c), \bar{i}(g(c)), m(g(c))\}, \\ & \{\bar{i}(y), \bar{m}(y)\}, \end{aligned}$$

and then

$$\begin{aligned} & \{r(c)\}, \\ & \{\bar{r}(c), m(g(c))\}, \\ & \{\bar{m}(g(c))\}. \end{aligned}$$

Finally, we get

$$\begin{aligned} & \{r(c)\}, \\ & \{\bar{r}(c)\}, \end{aligned}$$

and, then, to complete the proof,

□.

The combination of unification with resolution can be thought of as a single step constituting a kind of generalized resolution. Thus, resolution in the sense of Chapter 12, that is, resolution involving only ground clauses, will now be called *ground resolution*, while the unmodified word *resolution* will be used to represent this more general operation. In the ground case we used the notation $\text{res}_\lambda(\kappa_1, \kappa_2)$ for the *resolvent* of κ_1, κ_2 with respect to the literal λ , namely,

$$(\kappa_1 - \{\lambda\}) \cup (\kappa_2 - \{\neg \lambda\}).$$

In the general case, let $\lambda \in \kappa_2, \neg \mu \in \kappa_2$, where the unification algorithm can be successfully applied to λ and $\neg \mu$. Thus, there are substitutions for the variables which yield new clauses $\tilde{\kappa}_1, \tilde{\kappa}_2$ such that if the substitutions transform λ into $\tilde{\lambda}$, they also transform $\neg \mu$ into $\neg \tilde{\lambda}$. Then we write

$$\text{res}_{\lambda, \mu}(\kappa_1, \kappa_2) = (\tilde{\kappa}_1 - \{\tilde{\lambda}\}) \cup (\tilde{\kappa}_2 - \{\neg \tilde{\lambda}\}).$$

Let α be a finite set of clauses. Then a sequence of clauses $\kappa_1, \kappa_2, \dots, \kappa_n$ is called a *resolution derivation of $\kappa_n = \kappa$ from α* if for each $i, 1 \leq i \leq n$, either $\kappa_i \in \alpha$ or there are $j, k < i$ and literals λ, μ such that $\kappa_i = \text{res}_{\lambda, \mu}(\kappa_j, \kappa_k)$. As in Chapter 12, a resolution derivation of \square from α is called a *resolution refutation of α* . The key theorem is

Theorem 5.1 (J. A. Robinson's General Resolution Theorem). Let $\zeta = \zeta(b_1, \dots, b_n)$ be a W-formula containing no quantifiers, and let ζ be in CNF. Let

$$\gamma = (\forall b_1)(\forall b_2) \cdots (\forall b_n)\zeta.$$

Then, the sentence γ is unsatisfiable if and only if there is a resolution refutation of the clauses of ζ .

We shall not prove this theorem here, but will content ourselves with showing how it applies to Example 3 of the previous section. The clauses were

1. $\{r(x, g(x)), r(g(x), x)\}$
2. $\{\bar{r}(u, v), r(v, v)\}$
3. $\{\bar{r}(z, z)\}$.

A resolution refutation is obtained as follows:

4. $\{r(g(x), x), r(g(x), g(x))\}$ (resolving 1 and 2);
5. $\{r(x, x), r(g(x), g(x))\}$ (resolving 2 and 4);
6. $\{r(g(x), g(x))\}$ (resolving 3 and 5);
7. \square (resolving 3 and 6).

Exercises

1. Indicate which of the following pairs of terms are unifiable.
 - (a) $x, g(y)$.
 - (b) $x, g(x)$.
 - (c) $f(x), g(y)$.
 - (d) $f(x, h(a)), f(g(y), h(y))$.
 - (e) $f(x, x), f(g(y), a)$.
 - (f) $f(x, y, z), f(g(w, w), g(x, x), g(y, y))$.
2. Prove the correctness of the inferences of Exercises 3.1–3.3 by obtaining minimally unsatisfiable sets of clauses.
3. Prove the correctness of the inferences of Exercises 3.1–3.3 by obtaining resolution refutations.
4. (a) Prove that the problem of the validity of the sentence

$$(\exists x)(\exists y)(\forall z)((r(x, y) \supset (r(y, z) \wedge r(z, z))) \\ \wedge ((r(x, y) \wedge s(x, y)) \supset (s(x, z) \wedge s(z, z))))$$

leads to the list of clauses

$$\{r(x, y)\}, \\ \{s(x, y), \bar{r}(y, h(x, y)), \bar{r}(h(x, y), h(x, y))\}, \\ \{\bar{r}(y, h(x, y)), \bar{r}(h(x, y), h(x, y)), \\ \bar{s}(x, h(x, y)), \bar{s}(h(x, y), h(x, y))\}.$$

[Hint: Use Theorem 5.1 in Chapter 12.]

- (b) Prove the validity of the sentence in (a) by giving a resolution refutation.
- 5.* A conventional notation for describing a substitution is $\{x_1/t_1, \dots, x_n/t_n\}$, where x_1, \dots, x_n are distinct variables and t_1, \dots, t_n are terms. If λ is a term or a formula and θ is a substitution, then $\lambda\theta$ denotes the result of simultaneously replacing each occurrence of x_i in λ by t_i , $1 \leq i \leq n$. A *unifier* of two terms or formulas λ, μ is a substitution θ such that $\lambda\theta$ and $\mu\theta$ are identical. Modify the unification algorithm so that if λ, μ are unifiable, it returns a unifier of λ, μ . Apply the modified algorithm to Exercise 1.
- 6.* An \vee -clause with at most one literal that is not negated is called a *Horn clause*. Horn clauses are the basis of *logic programming languages* such as Prolog. Horn clauses of the form λ or $(\neg \lambda_1 \vee \dots \vee \neg \lambda_n \vee \lambda)$, where the latter is sometimes written $(\lambda_1 \wedge \dots \wedge \lambda_n \supset \lambda)$, are

called *program clauses*, and a *Horn program* is a set (or conjunction) of program clauses. The input to a Horn program \mathcal{P} is a clause of the form $(\neg \lambda_1 \vee \dots \vee \neg \lambda_n)$, called a *goal clause*, and the output is a substitution θ , called an *answer substitution*, such that

$$(\forall x_1) \dots (\forall x_l) \mathcal{P} \vDash (\forall y_1) \dots (\forall y_k) [(\lambda_1 \wedge \dots \wedge \lambda_n) \theta],$$

where x_1, \dots, x_l are all of the variables which occur free in \mathcal{P} and y_1, \dots, y_k are all of the variables which occur free in $(\lambda_1 \wedge \dots \wedge \lambda_n) \theta$. (If there is no such answer substitution then the program can either stop and return NO or it can run forever.) If $(\lambda_1 \wedge \dots \wedge \lambda_n) \theta$ has no free variable occurrences, then θ is a *ground answer substitution*.

- (a) Let θ be a substitution such that $(\lambda_1 \vee \dots \vee \lambda_n) \theta$ has no free variable occurrences. Show that θ is a ground answer substitution if and only if

$$(\forall x_1) \dots (\forall x_l) [\mathcal{P} \cup \{(\neg \lambda_1 \vee \dots \vee \neg \lambda_n) \theta\}]$$

is unsatisfiable.

- (b) Let \mathcal{P} be the Horn program with clauses

$$\begin{aligned} &\{\text{edge}(a, b), \text{edge}(b, c), \text{edge}(x, y) \supset \text{connected}(x, y), \\ &\quad \text{edge}(x, y) \wedge \text{connected}(y, z) \supset \text{connected}(x, z)\}. \end{aligned}$$

For each of the following goal clauses, use resolution and the modified unification algorithm from Exercise 5 to find all possible answer substitutions.

- (i) $\neg \text{edge}(a, y).$
- (ii) $\neg \text{edge}(x, a).$
- (iii) $\neg \text{edge}(x, y).$
- (iv) $\neg \text{connected}(b, y).$
- (v) $\neg \text{connected}(a, y).$

6. Compactness and Countability

In this section we give two applications of the circle of ideas surrounding Herbrand's theorem that are extremely important in mathematical logic. It will be interesting to see if they have a role to play in the application of logic to computer science.

Theorem 6.1 (Compactness Theorem for Predicate Logic). Let Ω be a set of \mathbf{W} -sentences each finite subset of which is satisfiable. Then Ω is satisfiable.

Proof. If Ω is finite, there is nothing to prove. If Ω is infinite, we can use the enumeration principle from Chapter 12, Section 7, to obtain an enumeration $\beta_0, \beta_1, \beta_2, \dots$ of the elements of Ω . Let us write

$$\gamma_n = \bigwedge_{i \leq n} \beta_i, \quad n = 0, 1, 2, \dots$$

Let steps (I)–(V) of Section 3 be applied to each of $\beta_0, \beta_1, \beta_2, \dots$ to obtain universal sentences

$$\alpha_i = (\forall b_1^{(i)}) \cdots (\forall b_{m_i}^{(i)}) \zeta_i(b_1^{(i)}, \dots, b_{m_i}^{(i)}).$$

Then by Theorem 3.3, for each i , α_i is satisfiable if and only if β_i is satisfiable, and moreover any model of α_i is also a model of β_i . Now let us apply the same steps (I)–(V) to the sentence γ_n . We see that if we use generalized Skolemization we can do this in such a way that the universal sentence δ_n we obtain, corresponding to γ_n in the sense of Theorem 3.3, consists of universal quantifiers followed by the formula

$$\bigwedge_{i \leq n} \zeta_i.$$

Now, by hypothesis, each γ_n is satisfiable. Hence, by Theorem 3.3, so is each δ_n . For each n , let \mathbf{H}_n be the Herbrand universe of δ_n . Thus,

$$\mathbf{H}_0 \subseteq \mathbf{H}_1 \subseteq \mathbf{H}_2 \cdots.$$

Let $\mathbf{H} = \bigcup_{n \in N} \mathbf{H}_n$. By Theorem 4.1, the sets

$$\Sigma_n = \left\{ \bigwedge_{i \leq n} \zeta_i(t_1^{(i)}, \dots, t_{m_i}^{(i)}) \mid t_1^{(i)}, \dots, t_{m_i}^{(i)} \in \mathbf{H}_n, i = 0, 1, \dots, n \right\}$$

are truth-functionally satisfiable. We wish to show that the set

$$\Gamma = \{ \zeta_i(t_1, \dots, t_{m_i}) \mid t_1, t_2, \dots \in \mathbf{H} \}$$

is itself truth-functionally satisfiable. By the compactness theorem for propositional calculus (Theorem 7.2 in Chapter 12) it suffices to prove this for every finite subset Δ of Γ . But for any finite subset Δ of Γ , there is a largest value of the subscript i which occurs, and all the t_j which occur are in some \mathbf{H}_k . Let l be the larger of this subscript k and this largest value of i . Then Δ is itself a subset of

$$\Lambda_l = \{ \zeta_i(t_1, \dots, t_{m_i}) \mid t_1, t_2, \dots \in \mathbf{H}_l, 0 \leq i \leq l \}.$$

Moreover, since Σ_l is truth-functionally satisfiable, so is Λ_l , and therefore Δ . This shows that Γ is truth-functionally satisfiable.

Now, let \mathcal{A} be the set of all atoms which occur in the formulas that belong to Γ . Let v be an assignment on \mathcal{A} such that $\beta^v = 1$ for all $\beta \in \Gamma$. Then we use v to construct an interpretation I of \mathbf{W} with domain \mathbf{H} precisely as in the proof of Theorem 4.1. Then Lemmas 2 and 3 of that proof hold and precisely as in that case we have

$$\zeta'_i[t_1, \dots, t_{m_i}] = 1 \quad \text{for all } t_1, \dots, t_{m_i} \in \mathbf{H} \text{ and } i \in N.$$

Hence, $\alpha'_i = 1$ for all $i \in N$. Since any model of α_i is also a model of β_i , we have $\beta'_i = 1$ for all $i \in N$. Thus, I is a model of Ω . ■

Now let us begin with a set Ω of \mathbf{W} -sentences which has a model I . Then of course I is a model of every finite subset of Ω . Thus, the *method of proof* of the previous theorem can be applied to Ω . Of course, this would be pointless if our aim were merely to obtain a model of Ω ; we already have a model I of Ω . But the method of proof of Theorem 6.1 gives us a model of Ω whose domain \mathbf{H} is a language on an alphabet. Thus, we have proved

Theorem 6.2 (Skolem–Löwenheim Theorem). Let Ω be a satisfiable set of \mathbf{W} -sentences. Then Ω has a model whose domain is a language on some alphabet.

What makes this important and interesting is that any language satisfies the enumeration principle of Chapter 12, Section 7. Infinite sets that possess an enumeration are called *countably infinite*. This brings us to the usual form of the Skolem–Löwenheim theorem.

Corollary 6.3. Let Ω be a satisfiable set of \mathbf{W} -sentences. Then Ω has a model whose domain is countably infinite.

Many infinite sets that occur in mathematics are *not* countable. In fact, the diagonal method, which was used in obtaining unsolvability results in Part 1 of this book, was originally developed by Cantor to prove that the set of real numbers is not countable. What the Skolem–Löwenheim theorem shows is that no set of sentences can characterize an infinite uncountable set in the sense of excluding countable models.

We close this section with another useful form of the compactness theorem.

Theorem 6.4. If $\Gamma \vDash \gamma$, then there is a finite subset Δ of Γ such that $\Delta \vDash \gamma$.

Proof. Since every model of Γ is a model of γ , the set $\Gamma \cup \{\neg \gamma\}$ has no models; that is, it is not satisfiable. Thus, by Theorem 6.1, there is a finite

subset Δ of Γ such that $\Delta \cup \{\neg \gamma\}$ is unsatisfiable. Thus every model of Δ is a model of γ , i.e., $\Delta \models \gamma$. ■

Exercises

- Let Ω_1, Ω_2 be sets of sentences such that $\Omega_1 \cup \Omega_2$ is unsatisfiable. Prove that there is a sentence α such that $\Omega_1 \models \alpha$, and $\Omega_2 \models \neg \alpha$.
- Show that if a set Ω of sentences has models with arbitrarily large finite domains, then it has a model with an infinite domain. [Hint: Show that $\Omega \cup \{(\exists x_1) \cdots (\exists x_n) \wedge_{1 \leq i < j \leq n} x_i \neq x_j \mid n \in N\}$ is satisfiable.]
- Let \mathbf{W} be the vocabulary $\{0, c, s, >\}$, where $0, c$ are constant symbols, s is a function symbol with $\delta(s) = 1$, and $>$ is a relation symbol with $\delta(>) = 2$. Use the compactness theorem to show that the set of sentences $\{c > 0, c > s(0), c > s(s(0)), \dots\}$ is satisfiable.

*7. Gödel's Incompleteness Theorem

Let Γ be a recursive set of \mathbf{W} -sentences for some given vocabulary \mathbf{W} . We think of Γ as being considered for use as a set of “axioms” for some part of mathematics. The requirement that Γ be recursive is natural, because, by Church’s thesis, it simply amounts to requiring that there be some algorithmic method of determining whether or not an alleged “axiom” really is one. Often Γ will be finite. We define $T_\Gamma = \{\gamma \mid \Gamma \models \gamma\}$ and call T_Γ the *axiomatizable theory* on \mathbf{W} whose axioms are the sentences belonging to the set Γ . Of course, it is quite possible to have different sets of axioms which define the same theory.

If T is an axiomatizable theory, we write

$$\vdash_T \gamma$$

(read: “ T proves γ ”) to mean that $\gamma \in T$. We also write $\not\vdash_T \gamma$ to mean that $\gamma \notin T$. The most important fact about axiomatizable theories is given by the following theorem.

Theorem 7.1. An axiomatizable theory is r.e.

Proof. By Theorems 3.1 and 6.4, $\gamma \in T_\Gamma$ if and only if

$$(\gamma_1 \wedge \gamma_2 \wedge \cdots \wedge \gamma_n \wedge \neg \gamma)$$

is unsatisfiable for some $\gamma_1, \gamma_2, \dots, \gamma_n \in \Gamma$. Since Γ is recursive, it is certainly r.e. Thus, by Theorem 4.11 in Chapter 4, there is a recursive function g on N whose range is Γ . For a given sentence γ , let

$$\delta(n, \gamma) = (g(0) \wedge g(1) \wedge \cdots \wedge g(n) \wedge \neg\gamma)$$

for all $n \in N$. Clearly, $\delta(n, \gamma)$ is a recursive function of n and γ . Moreover, the sentence γ belongs to T_Γ if and only if there is an $n \in N$ such that $\delta(n, \gamma)$ is unsatisfiable. But by Theorem 4.3, the set of unsatisfiable **W**-sentences is r.e. Hence there is a partially computable function h which is defined for a given input if and only if that input is an unsatisfiable **W**-sentence. Let h be computed by program \mathcal{P} and let $p = \#(\mathcal{P})$. Then the following “dovetailing” program halts if and only if the input γ belongs to T_Γ , thereby showing that T_Γ is r.e.:

```
[ A ]   Z ← δ(l(T), γ)
          T ← T + 1
          IF ~ STP(1)(Z, p, r(T)) GOTO A
```

■

We shall see in the next section that there is a Γ such that T_Γ is not recursive.

Now let **W** be some vocabulary intended for use in expressing properties of the natural numbers. By a *numeral system* for **W**, we mean a *recursive function* ν on N such that for each $n \in N$, $\nu(n)$ is a **W**-term containing no variables, and such that for all $n, m \in N$, $n \neq m$ implies $\nu(n) \neq \nu(m)$. When ν can be understood from the context, we write \bar{n} for $\nu(n)$. \bar{n} is called the *numeral* corresponding to n and may be thought of as a notation for n using the vocabulary **W**. A popular choice is

$$\bar{n} = S(S(\cdots S(0) \cdots)),$$

where S is a function symbol of degree 1, 0 is a constant symbol, and the number of occurrences of S is n .

Let $\alpha = \alpha(b)$ be a **W**-formula and let **T** be an axiomatizable theory on **W**. Then, given a numeral system for **W**, we can associate with α the set

$$U = \{n \in N \mid \vdash_T \alpha(\bar{n})\}. \quad (7.1)$$

In this case, we say that the formula α *represents the set U in T*. If we begin with a set $U \subseteq N$, we can ask the question: is there a **W**-formula α which represents U in **T**? We have

Theorem 7.2. If there is a formula α which represents the set U in an axiomatizable theory **T**, then U is r.e.

Proof. Let \mathbf{T} be an axiomatizable theory, and let α represent U in \mathbf{T} . By Theorem 7.1, we know that there is a program \mathcal{P} that will halt for given input γ if and only if $\vdash_{\mathbf{T}} \gamma$. Given $n \in N$, we need only compute $\alpha(\bar{n})$ [which we can do because $\nu(n) = \bar{n}$ is recursive], and feed it as input to \mathcal{P} . The new program thus defined halts for given input $n \in N$ if and only if $\vdash_{\mathbf{T}} \alpha(\bar{n})$. By (7.1), U is r.e. ■

In fact, there are many axiomatizable theories in which all r.e. sets are representable. To see the negative force of Theorem 7.2, we rewrite it as follows.

Corollary 7.3. Let \mathbf{T} be an axiomatizable theory. Then if $U \subseteq N$ is not r.e., there is no formula which represents U in \mathbf{T} .

This corollary is a form of Gödel's incompleteness theorem. To obtain a more striking form of the theorem, let us say that the formula α *quasi-represents the set U in \mathbf{T}* if

$$\{n \in N \mid \vdash_{\mathbf{T}} \alpha(\bar{n})\} \subseteq U. \quad (7.2)$$

We can think of such a formula α as intended to express the proposition " $n \in U$ " using the vocabulary \mathbf{W} . Comparing (7.1) and (7.2) and considering Corollary 7.3, we have

Corollary 7.4. Let \mathbf{T} be an axiomatizable theory and let $U \subseteq N$ be a set that is not r.e. Let the formula α quasi-represent U in \mathbf{T} . Then, there is a number n_0 such that $n_0 \in U$ but $\not\vdash_{\mathbf{T}} \alpha(\bar{n}_0)$.

As we can say loosely, the sentence $\alpha(\bar{n}_0)$ is "true" but not provable. Corollary 7.4 is another form of Gödel's incompleteness theorem. We conclude with our final version.

Theorem 7.5. Let \mathbf{T} be an axiomatizable theory, and let S be an r.e. set that is not recursive. Let $\alpha = \alpha(x)$ be a formula such that α represents S in \mathbf{T} , and $\neg \alpha$ quasi-represents \bar{S} in \mathbf{T} . Then there is a number n_0 such that $\not\vdash_{\mathbf{T}} \alpha(\bar{n}_0)$ and $\not\vdash_{\mathbf{T}} \neg \alpha(\bar{n}_0)$.

Proof. We take $U = \bar{S}$ in Corollary 7.4 to obtain a number n_0 such that $n_0 \in \bar{S}$, but $\not\vdash_{\mathbf{T}} \neg \alpha(\bar{n}_0)$. Since $n_0 \notin S$ and α represents S in \mathbf{T} , we must also have $\not\vdash_{\mathbf{T}} \alpha(\bar{n}_0)$. ■

In this last case, it is usual to say that $\alpha(\bar{n}_0)$ is undecidable in \mathbf{T} .

Exercises

- Let Γ be an r.e. set of \mathbf{W} -sentences for some vocabulary \mathbf{W} . Show that $\{\gamma \mid \Gamma \vDash \gamma\}$ is r.e.

2. Let T be an axiomatizable theory on some vocabulary W . T is *consistent* if there is no W -sentence α such that both $\vdash_T \alpha$ and $\vdash_T \neg \alpha$, and T is *inconsistent* otherwise.
 - (a) Show that if T is inconsistent then $\vdash_T \alpha$ for all W -sentences α .
 - (b) Show that if there is a formula which represents some nonrecursive set in T , then T is consistent.
 - (c) Show that if T is consistent and the formula α represents some r.e. set U in T , then $\neg \alpha$ quasi-represents \bar{U} in T .
3. An axiomatizable theory T on vocabulary W is *complete* if for all W -sentences α , either $\vdash_T \alpha$ or $\vdash_T \neg \alpha$. Show that if T is complete then it is recursive. [See also Exercise 2.]
4. An axiomatizable theory T on some vocabulary W is ω -*consistent* if the following holds for all W -formulas $\alpha(b)$: If $\vdash_T \neg \alpha(\bar{n})$ for all $n \in N$, then $\not\vdash_T (\exists x)\alpha(x)$. Show that if T is ω -consistent then it is consistent. [See Exercise 2 for the definition of consistency.]
5. A function $f(x_1, \dots, x_n)$ is *representable* in an axiomatizable theory T if there is a formula $\alpha(b_1, \dots, b_n, b)$ such that if $f(m_1, \dots, m_n) = k$ then

$$\vdash_T \alpha(\bar{m}_1, \dots, \bar{m}_n, \bar{k}) \quad \text{and} \quad \vdash_T (\forall y)(\alpha(\bar{m}_1, \dots, \bar{m}_n, y) \supset y = \bar{k}).$$

We say that α *represents* $f(x_1, \dots, x_n)$ in T . Let T be a consistent axiomatizable theory [see Exercise 2] such that $\vdash_T \bar{0} \neq \bar{1}$ and such that every primitive recursive function is representable in T .

- (a) Let $\alpha(x, y, t, z)$ represent the function $\text{STP}^{(1)}(x, y, t)$ in T , and for every r.e. set W_m , let $\beta_m(x)$ be the formula $(\exists t)\alpha(x, \bar{m}, t, \bar{1})$. Show that if $n \in W_m$ then $\vdash_T \beta_m(\bar{n})$.
- (b) Show that if $n \notin W_m$ then $\vdash_T \neg \alpha(\bar{n}, \bar{m}, \bar{t}, \bar{1})$ for all $t \in N$.
- (c) Show that if T is ω -consistent then $n \notin W_m$ implies $\not\vdash_T \beta_m(\bar{n})$. [See Exercise 4.]
- (d) Conclude that if T is an ω -consistent axiomatizable theory in which every primitive recursive function is representable and if $\vdash_T \bar{0} \neq \bar{1}$, then T has an undecidable sentence.

*8. Unsolvability of the Satisfiability Problem in Predicate Logic

In 1928, the great mathematician David Hilbert called the problem of finding an algorithm for testing a given sentence to determine whether it is

satisfiable “the main problem of mathematical logic.” This was because experience had shown that all of the inferences in mathematics could be expressed within the logic of quantifiers. Thus, an algorithm meeting Hilbert’s requirements would have provided, in principle, algorithmic solutions to all the problems in mathematics. So, when unsolvable problems were discovered in the 1930s, it was only to be expected that Hilbert’s satisfiability problem would also turn out to be unsolvable.

Theorem 8.1 (Church–Turing). There is a vocabulary \mathbf{W} such that there is no algorithm for testing a given \mathbf{W} -sentence to determine whether it is satisfiable.

Proof. Our plan will be to translate the word problem for a Thue process into predicate logic in such a way that a solution to Hilbert’s satisfiability problem would also yield a solution to the word problem for the given process.

Thus, using Theorem 3.5 in Chapter 7, let Π be a Thue process on the alphabet $\{a, b\}$ with an unsolvable word problem. Let Π have the productions $g_i \rightarrow h_i, i = 1, 2, \dots, K$, together with their inverses, where we may assume that for each i , $g_i, h_i \neq 0$ (recall Theorem 3.5 in Chapter 7). We introduce the vocabulary $\mathbf{W} = \{\mathbf{a}, \mathbf{b}, \bullet, \equiv\}$, where \mathbf{a}, \mathbf{b} are constant symbols, \bullet is a function symbol, and \equiv is a relation symbol, with $\delta(\bullet) = \delta(\equiv) = 2$. We will make use of the interpretation I with domain $\{a, b\}^* - \{0\}$ which is defined as follows:

$$\begin{aligned}\mathbf{a}_I &= a, \\ \mathbf{b}_I &= b, \\ \bullet_I(u, v) &= uv, \\ \equiv_I(u, v) &= 1 \quad \text{if and only if } u \xrightarrow[\Pi]^* v.\end{aligned}$$

For ease of reading, we shall write \bullet and \equiv in “infix” position. Thus, we shall write, for example,

$$((x \bullet a) \equiv y) \quad \text{instead of} \quad \equiv (\bullet(x, a), y).$$

For each word $w \in \{a, b\}^* - \{0\}$, we now define a \mathbf{W} -term $w^\#$ as follows:

$$\begin{aligned}a^\# &= \mathbf{a}, & b^\# &= \mathbf{b}, \\ (ua)^\# &= (u^\# \bullet a), & (ub)^\# &= (u^\# \bullet b).\end{aligned}\tag{8.1}$$

We have

Lemma 1. For every word $w \in \{a, b\}^* - \{0\}$, we have $(w^\#)^I = w$.

Proof. The proof is by an easy induction on $|w|$, using (8.1) and the definition of the interpretation I . ■

Let Γ be the set of **W**-sentences obtained by prefixing the appropriate universal quantifiers to each **W**-formula in the following list:

1. $(x \equiv x)$,
2. $((x \equiv y) \supset (y \equiv x))$,
3. $((x \equiv y) \wedge (y \equiv z)) \supset (x \equiv z))$,
4. $((x \equiv y) \wedge (u \equiv v)) \supset ((x \bullet u) \equiv (y \bullet v)))$,
5. $((x \bullet y) \bullet z) \equiv (x \bullet (y \bullet z)))$,
- 5 + i. $(g_i^\# \equiv h_i^\#)$, $1 \leq i \leq K$.

We have

Lemma 2. The interpretation I is a model of the set of sentences Γ .

Proof. The sentences of Γ all express in logical notation basic facts about concatenation of strings and about derivations in Thue processes. Detailed verification is left to the reader. ■

Lemma 3. If $\Gamma \vDash (u^\# \equiv v^\#)$, then $u \xrightarrow[\Pi]{} v$.

Proof. By the definition of logical inference and Lemma 2, we have $(u^\# \equiv v^\#)^I = 1$. Hence

$$u = (u^\#)^I \xrightarrow[\Pi]{} (v^\#)^I = v. \quad \blacksquare$$

We next wish to establish the converse of Lemma 3. For this it will suffice to show that if $u \xrightarrow[\Pi]{} v$, then the sentence

$$\bigwedge_{\alpha \in \Gamma} \alpha \wedge \neg(u^\# \equiv v^\#)$$

is unsatisfiable (recall Theorem 3.1). The Herbrand universe is

$$H = \{a, b, a \bullet a, a \bullet b, b \bullet a, b \bullet b, a \bullet (a \bullet a), \dots\}.$$

Let us call a **W**-sentence α a *Herbrand instance* of a **W**-formula β if α can be obtained from β by replacing each of its free variables by an element of **H**. α is said to be *rooted* if it is a tautological consequence of the sentences 5 + i together with Herbrand instances of the formulas listed in 1–5. Obviously, if the sentence β is rooted, then $\Gamma \vDash \beta$.

Lemma 4. If $w = uv$, where $u \neq 0$ and $v \neq 0$, then

$$(w^\# \cong (u^\# \bullet v^\#)) \quad (8.2)$$

is rooted.

Proof. The proof is by induction on $|v|$. If $|v| = 1$, we can assume without loss of generality that $v = a$. But in this case, the sentence (8.2) is a Herbrand instance of formula 1.

Supposing the result known for v , we need to establish it for va and vb . We give the proof for va , that for vb being similar. So let $w = uv$, where we can assume that (8.2) is rooted. We need to show that the sentence

$$((wa)^\# \cong (u^\# \bullet (va)^\#))$$

is likewise rooted. By (8.1) this amounts to showing that

$$((w^\# \bullet a) \cong (u^\# \bullet (v^\# \bullet a)))$$

is rooted. But this follows from the induction hypothesis, noting that the following sentences are rooted. (For each of these sentences, the number of the corresponding formula of which it is a Herbrand instance is given.)

$$(a \cong a) \quad (1)$$

$$(((w^\# \cong (u^\# \bullet v^\#)) \wedge (a \cong a)) \supset ((w^\# \bullet a) \cong ((u^\# \bullet v^\#) \bullet a))) \quad (4)$$

$$(((u^\# \bullet v^\#) \bullet a) \cong (u^\# \bullet (v^\# \bullet a))) \quad (5)$$

$$\begin{aligned} (((((w^\# \bullet a) \cong ((u^\# \bullet v^\#) \bullet a)) \wedge (((u^\# \bullet v^\#) \bullet a) \cong (u^\# \bullet (v^\# \bullet a)))))) \\ \supset (((w^\# \bullet a) \cong (u^\# \bullet (v^\# \bullet a))). \end{aligned} \quad (3)$$

■

Lemma 5. If $u \underset{\Pi}{\Rightarrow} v$, then $(u^\# \cong v^\#)$ is rooted.

Proof. For some i , $1 \leq i \leq K$, we have either $u = pg_i q$, $v = ph_i q$, or $u = ph_i q$, $v = pg_i q$, where $p, q \in \{a, b\}^*$. We may assume that in fact $u = pg_i q$, $v = ph_i q$, because in the other case we could use the following Herbrand instance of formula 2:

$$((v^\# \cong u^\#) \supset (u^\# \cong v^\#)).$$

The proof now divides into three cases.

Case I. $p = q = 0$. Then the sentence $(u^\# \cong v^\#)$ is just $5 + i$ and is therefore in Γ .

Case II. $p = 0, q \neq 0$. Using $5 + i$ and the following Herbrand instance of formula 4:

$$(((g_i^\# \cong h_i^\#) \wedge (q^\# \cong q^\#)) \supset ((g_i^\# \bullet q^\#) \cong (h_i^\# \bullet q^\#))),$$

we see that the sentence

$$((g_i^\# \bullet q^\#) \cong (h_i^\# \bullet q^\#))$$

is rooted. Using Lemma 4 and Herbrand instances of formulas 2 and 3 we obtain the result.

Case III. $p, q \neq 0$. Using Case II, the sentence $((g_i q)^\# \cong (h_i q)^\#)$ is rooted. Using the Herbrand instance of formula 4:

$$\begin{aligned} (((p^\# \cong p^\#) \wedge ((g_i q)^\# \cong (h_i q)^\#)) \\ \supset ((p^\# \bullet (g_i q)^\#) \cong (p^\# \bullet (h_i q)^\#))), \end{aligned}$$

we see that

$$((p^\# \bullet (g_i q)^\#) \cong (p^\# \bullet (h_i q)^\#))$$

is rooted. The result now follows using Lemma 4 and Herbrand instances of formulas 2 and 3. ■

Lemma 6. If $u \xrightarrow{*}_{\Pi} v$, then $(u^\# \cong v^\#)$ is rooted.

Proof. The proof is by induction on the length of a derivation of v from u . If this length is 1, then $v = u$, and we may use a Herbrand instance of formula 1. To complete the proof, we may assume that $u \xrightarrow{*}_{\Pi} w \xrightarrow{*}_{\Pi} v$, where it is known that $(u^\# \cong w^\#)$ is rooted. By Lemma 5, $(w^\# \cong v^\#)$ is rooted. We then get the result by using the following Herbrand instance of formula 3:

$$(((u^\# \cong w^\#) \wedge (w^\# \cong v^\#)) \supset (u^\# \cong v^\#)).$$

Combining Lemmas 3 and 6, we obtain

Lemma 7. $u \xrightarrow{*}_{\Pi} v$ if and only if $\Gamma \vDash (u^\# \cong v^\#)$.

Now it is easy to complete the proof of our theorem. If we possessed an algorithm for testing a given W-sentence for satisfiability, we could use it to test the sentence

$$\bigwedge_{\alpha \in \Gamma} \alpha \wedge \neg (u^\# \cong v^\#)$$

and therefore, by Theorem 3.1, to test the correctness of the logical inference $\Gamma \models (u^* \equiv v^*)$. This would in turn lead to an algorithm for solving the word problem for Π , which we know is unsolvable. ■

A final remark: We really have been working with the axiomatizable theory T_Γ . Thus what Lemma 7 states is just that

$$u \stackrel{*}{\Pi} v \quad \text{if and only if} \quad \vdash_\Gamma (u^* \equiv v^*). \quad (8.3)$$

Hence we conclude that the theory T_Γ is not recursive. [If it were, we could use (8.3) to solve the word problem for Π .] Thus we have proved

Theorem 8.2. There are axiomatizable theories that are not recursive.

Exercises

1. Prove Lemma 2.
2. Let W be the vocabulary used in this section. Show that for every deterministic Turing machine \mathcal{M} there is a finite set Γ of W -sentences and a computable function $f(x)$ such that for any string w , \mathcal{M} accepts w if and only if $\Gamma \models f(w)$. [Hint: See Theorems 3.3 and 3.4 in Chapter 7.]

Part 4

Complexity

14

Abstract Complexity

1. The Blum Axioms

In this chapter we will develop an abstract theory of the amount of resources needed to carry out computations. In practical terms resources can be measured in various ways: storage space used, time, some weighted average of central processor time and peripheral processor time, some combinations of space and time used, or even monetary cost. The theorems proved in this chapter are quite independent of which of these “measures” we use. We shall work with two very simple assumptions known as the *Blum axioms* after Manuel Blum, who introduced them in his doctoral dissertation. These assumptions are satisfied by any of the “measures” mentioned above (if given precise definitions in any natural manner) as well as by many others.

Definition. A 2-ary partial function C on N is called a *complexity measure* if it satisfies the *Blum axioms*:

1. $C(x, i) \downarrow$ if and only if $\Phi_i(x) \downarrow$;
2. The predicate $C(x, i) \leq y$ is recursive. (This predicate is of course false if $C(x, i) \uparrow$.)

We write $C_i(x) = C(x, i)$. We think of $C_i(x)$ as the *complexity of the computation* that occurs when the program whose number is i is fed the

input x . It is not very difficult to see that various natural ways of measuring complexity of computation do satisfy the Blum axioms. What is remarkable is that some very interesting and quite nontrivial results can be derived from such meager assumptions.

Let us examine some examples of proposed complexity measures:

1. $C_i(x) = \text{the number of steps in a computation by program number } i \text{ on input } x$. The first axiom is clearly satisfied; the second follows from the computability of the step-counter predicate $\text{STP}^{(1)}$.
2. $M_i(x) = \text{the largest value assumed by any variable in program number } i \text{ when this program is given input } x, \text{ if } \Phi_i(x) \downarrow; M_i(x) \uparrow \text{ otherwise}$. The definition forces the first axiom to be true. The truth of the second axiom is a more subtle matter. The key observation is that, for a given program, there are only finitely many *different* snapshots¹ in which all variables have values less than or equal to a given number y . Hence, given numbers i, x, y we can test the condition $M_i(x) \leq y$ by “running” program number i on the input x until one of the following occurs:
 - I. *A snapshot is reached in which some variable has a value $> y$.* Then we return the value FALSE.
 - II. *The computation halts with all variables having values $\leq y$.* Then we return the value TRUE.
 - III. *The same snapshot is reached twice.* (By the pigeon-hole principle this must happen eventually if neither I nor II occurs.) Then, recognizing that the computation is in an “infinite” loop and so will never terminate, we return the value FALSE. (The reader should note that this algorithm in no way contradicts the unsolvability of the halting problem. Case I can include both halting and nonhalting computations.)

We will make important use of this “maximum-space” complexity measure, and we reserve the notation $M_i(x)$ for it.

3. $C_i(x) = \Phi_i(x)$. Although the first Blum axiom is satisfied, the second is *certainly not*; namely, choose i so that

$$\Phi_i(x) = \begin{cases} 0 & \text{for } x \in S \\ \uparrow & \text{otherwise,} \end{cases}$$

where S is any given r.e. nonrecursive set. Then the condition $\Phi_i(x) \leq 0$ is equivalent to $x \in S$ and hence is not recursive.

¹ The definition of *snapshot* is in Chapter 2, Section 3.

If $P(x)$ is any predicate on N , we write

$$P(x) \quad \text{a.e.},$$

and say that $P(x)$ is true *almost everywhere*, to mean that there exists $m_0 \in N$ such that $P(x)$ is true for all $x > m_0$. Equivalently, $P(x)$ is true for all but a finite set of numbers. We may think of a partial function on N as a total function with values in the set $N \cup \{\infty\}$. That is, we write $g(x) = \infty$ to mean that $g(x) \uparrow$. We extend the meaning of $<$ so that $n < \infty$ for all $n \in N$. $x \leq y$ continues to mean $x < y$ or $x = y$, so that $n \leq \infty$ for $n \in N$ but also $\infty \leq \infty$.

The second Blum axiom can be written in the equivalent forms:

- 2'. The predicate $C_i(x) = y$ is recursive.
- 2''. The predicate $C_i(x) < y$ is recursive.

To see that 2, 2' and 2'' are all equivalent we note that

$$C_i(x) = y \Leftrightarrow (C_i(x) \leq y \ \& \ \sim(C_i(x) \leq y - 1)) \vee (y = 0 \ \& \ C_i(x) \leq y),$$

so that 2 implies 2'. 2' implies 2'' because

$$C_i(x) < y \Leftrightarrow (\exists z)_{< y} (C_i(x) = z).$$

Finally, 2'' implies 2 because

$$C_i(x) \leq y \Leftrightarrow C_i(x) < y + 1.$$

Let us call a *recursive* function $r(x)$ a *scaling factor* if

- 1. r is increasing, i.e., $r(x + 1) \geq r(x)$, and
- 2. $\lim_{x \rightarrow \infty} r(x) = \infty$, i.e. r assumes arbitrarily large values.

Condition 1 is obviously equivalent to the statement: $x \leq y$ implies $r(x) \leq r(y)$. Then we have

Theorem 1.1. Let $C_i(x)$ be a complexity measure and let $r(x)$ be a scaling factor. Let $D_i(x) = r(C_i(x))$. Then $D_i(x)$ is a complexity measure.

Proof. It is clear that D satisfies the first Blum axiom. To test $D_i(x) \leq y$, note that if $y < r(0)$ then $D_i(x) = r(C_i(x)) \geq r(0) > y$. Otherwise, find the number t for which

$$r(0) \leq r(1) \leq r(2) \leq \cdots \leq r(t) \leq y < r(t + 1).$$

We claim that $D_i(x) \leq y$ if and only if $C_i(x) \leq t$. It remains only to verify this claim. If $C_i(x) \leq t$, then

$$D_i(x) = r(C_i(x)) \leq r(t) \leq y.$$

Otherwise, if $t + 1 \leq C_i(x)$, then

$$y < r(t + 1) \leq r(C_i(x)) = D_i(x). \quad \blacksquare$$

This theorem is hardly surprising. Naturally, if $C_i(x)$ is a plausible complexity measure, we would expect $2^{C_i(x)}$ to be one as well. What is surprising is that any pair of complexity measures are related to each other in a manner not so different from C and D in Theorem 1.1.

Theorem 1.2 (Recursive Relatedness Theorem). Let C and D be arbitrary complexity measures. Then there is a recursive function $r(x, y)$ such that $r(x, y) < r(x, y + 1)$, and for all i

$$C_i(x) \leq r(x, D_i(x)) \quad \text{a.e.}$$

and

$$D_i(x) \leq r(x, C_i(x)) \quad \text{a.e.} \quad (1.1)$$

[where we let $r(x, \infty) = \infty$ for all x].

Proof. Note that by the first Blum axiom

$$\begin{aligned} C_i(x) \downarrow &\quad \text{if and only if} \\ \Phi_i(x) \downarrow &\quad \text{if and only if } D_i(x) \downarrow. \end{aligned}$$

By the second Blum axiom (in the form 2'), the predicate

$$C_i(x) = y \quad \vee \quad D_i(x) = y$$

is recursive. Hence the function h defined as follows is recursive:

$$h(i, x, y) = \begin{cases} \max(C_i(x), D_i(x)) & \text{if } C_i(x) = y \quad \vee \quad D_i(x) = y \\ 0 & \text{otherwise.} \end{cases}$$

Let

$$r(x, y) = y + \max_{j \leq x} \max_{z \leq y} h(j, x, z),$$

so that $r(x, y)$ is recursive. Then

$$\begin{aligned} r(x, y + 1) &= (y + 1) + \max_{j \leq x} \max_{z \leq y+1} h(j, x, z) \\ &> y + \max_{j \leq x} \max_{z \leq y} h(j, x, z) \\ &= r(x, y) \end{aligned}$$

since maximizing over a larger set of numbers cannot result in a smaller outcome. Moreover, using this same principle, and assuming that $x \geq i$,

$$\begin{aligned} r(x, D_i(x)) &\geq \max_{j \leq x} \max_{z \leq D_i(x)} h(j, x, z) \\ &\geq \max_{j \leq x} h(j, x, D_i(x)) \\ &\geq h(i, x, D_i(x)) \quad (\text{since } x \geq i) \\ &= \max(C_i(x), D_i(x)) \\ &\geq C_i(x). \end{aligned}$$

Thus, the inequality

$$r(x, D_i(x)) \geq C_i(x)$$

holds for all $x \geq i$ and hence *almost everywhere*. Since the definition of h is symmetric in C and D , the same argument shows that

$$r(x, C_i(x)) \geq D_i(x) \quad \text{a.e.} \quad \blacksquare$$

As we shall see, one use of the recursive relatedness theorem is in enabling us to proceed, in some cases, from the knowledge that a theorem is true for one particular complexity measure to the truth of that theorem for all complexity measures.

Exercises

1. Which of the following are complexity measures?

(a) $C_i(x) = 0$ for all i, x . (That is, all computation is “free.”)

(b) $C_i(x) = \begin{cases} M_i(x) & \text{for } i \notin A \\ 0 & \text{for } i \in A, \end{cases}$

where A is some given finite set such that Φ_i is total for all $i \in A$. (That is, the programs whose numbers belong to A can be run “free.”)

(c) $C_i(x) = 2^{\Phi_i(x)}$.

(d) $C_i(x) = \begin{cases} M_i(x) & \text{if } i \text{ is even} \\ \text{the number of steps in computing } \Phi_i(x) & \text{if } i \text{ is odd.} \end{cases}$

2. Prove that if C is a complexity measure and

$$D_i(x) = \begin{cases} C_i(x) & \text{for } i \notin A \\ 0 & \text{for } i \in A, \end{cases}$$

where A is as in Exercise 1(b), then D is a complexity measure.

3. Let $C_i(x)$ be the number of steps in the computation on input x by \mathcal{S} program \mathcal{P} , where $\#(\mathcal{P}) = i$. For some fixed $n > 0$, let $D_i(x)$ be the number of steps in the computation on input x by \mathcal{S}_n program \mathcal{P}' , where $\#(\mathcal{P}') = i$ and \mathcal{P}' is constructed from \mathcal{P} as in Section 3 of Chapter 5, by treating each \mathcal{S} instruction as a macro in \mathcal{S}_n .
- (a) Show that D is a complexity measure.
 - (b) Give a function $r(x, y)$ that satisfies the recursive relatedness theorem for C and D . [See Exercise 3.2 in Chapter 5.]
4. Let C be a complexity measure.
- (a) Show that for every i , $C_i(x)$ is partially computable.
 - (b) Show that if $\Phi_i(x)$ is total, then $C_i(x)$ is computable.
5. Let C be a complexity measure. Show that the predicate $P(i)$, defined

$$P(i) \Leftrightarrow (\forall x)(\exists y \in N)C_i(x) \leq y,$$

is not computable.

6. Let C be an arbitrary complexity measure. Show that there is a recursive function t such that

$$\Phi_i(x) \leq t(x, C_i(x)) \quad \text{a.e.}$$

[Hint: Use the complexity measure $M_i(x)$ and the recursive relatedness theorem.]

7. Can the result of the previous problem be improved so that t is a unary recursive function such that

$$\Phi_i(x) \leq t(C_i(x)) \quad \text{a.e.?}$$

Prove that your answer is correct.

8. (a) Let C be the complexity measure in Example 1. Show that for any computable function $f(x)$ there is a program number i such that $\Phi_i(x) = 0$ and $C_i(x) > f(x)$ for all x . Conclude that there are arbitrarily (with respect to computable lower bounds) slow \mathcal{S} programs that compute constant functions.

- (b) Let D be an arbitrary complexity measure. Show that for any computable function $f(x)$ there is a program number i such that $\Phi_i(x) = 0$ for all x and $D_i(x) > f(x)$ a.e. [Hint: Use (a) and the recursive relatedness theorem.]
9. Let C be a complexity measure. Show that there is no computable function $g(x, y)$ such that for all i, x , if $\Phi_i(x) \downarrow$ then $C_i(x) \leq g(x, \Phi_i(x))$. Compare with Exercise 6. [Hint: Use Exercise 8.]
10. Let C be a complexity measure. Show that for any computable function $f(x)$ there is a computable function $g(x)$ such that $g(x) \leq 1$ for all x and such that for any i , if $\Phi_i = g$ then $C_i(x) > f(x)$ for infinitely many x . Conclude that there are arbitrarily (with respect to computable lower bounds) complex “small” computable functions. [Hint: Define

$$g(x) = \begin{cases} 1 & \text{if } C_{l(x)}(x) \leq f(x) \text{ and } \Phi_{l(x)}(x) \neq 1 \\ 0 & \text{otherwise.} \end{cases}$$

11. Let C be a complexity measure. Show that for any computable function $f(x)$ there is a computable function $g(x)$ such that $g(x) \leq x$ for all x and such that for any i , if $\Phi_i = g$ then $C_i(x) > f(x)$ for all $x > i$. Compare with Exercise 10.

2. The Gap Theorem

In this section C is some given fixed complexity measure. Suppose that $t(x)$ is a complexity bound. That is, assume that we are restricted to computations for which $C_i(x) \leq t(x)$ whenever $\Phi_i(x) \downarrow$. Then, in response to our complaints, the bound is increased enormously to $g(t(x))$, where g is some recursive, rapidly increasing function, e.g., $g(x) = 2^x$ or

$$g(x) = 2^{2^{\dots^2}} \Big\} x \quad \text{or} \quad g(x) = 2^{2^{\dots^2}} \Big\} 2^x$$

Then, we can carry out far more computations. Right? Wrong! If the original function $t(x)$ is sufficiently tricky, it is possible that for every i , there are only finitely many values of x for which

$$C_i(x) \leq g(t(x)), \quad \text{but not} \quad C_i(x) \leq t(x).$$

This surprising assertion is a consequence of the gap theorem.

Theorem 2.1 (Gap Theorem). Let $g(x, y)$ be any recursive function such that $g(x, y) > y$. Then, there is a recursive function $t(x)$ such that if $x > i$ and $C_i(x) < g(x, t(x))$, then $C_i(x) \leq t(x)$. (See Fig. 2.1.)

Proof. Consider the predicate

$$P(x, y) \leftrightarrow (\forall i)_{< x} (C_i(x) \leq y \vee g(x, y) \leq C_i(x)).$$

By the second Blum axiom, the predicate $C_i(x) \leq y$ is computable. So is the predicate

$$g(x, y) \leq C_i(x) \leftrightarrow \sim (\exists z)_{< g(x, y)} (z = C_i(x)).$$

Hence, $P(x, y)$ is also recursive. We define

$$t(x) = \min_y P(x, y), \quad (2.1)$$

so that t is a partially computable function. We will show that t is total.

Let x be a given number. Consider the set $Q = \{C_i(x) \mid i < x \text{ & } \Phi_i(x) \downarrow\}$. Let $y_0 = 0$ if $Q = \emptyset$ and let y_0 be the largest element of Q otherwise. We

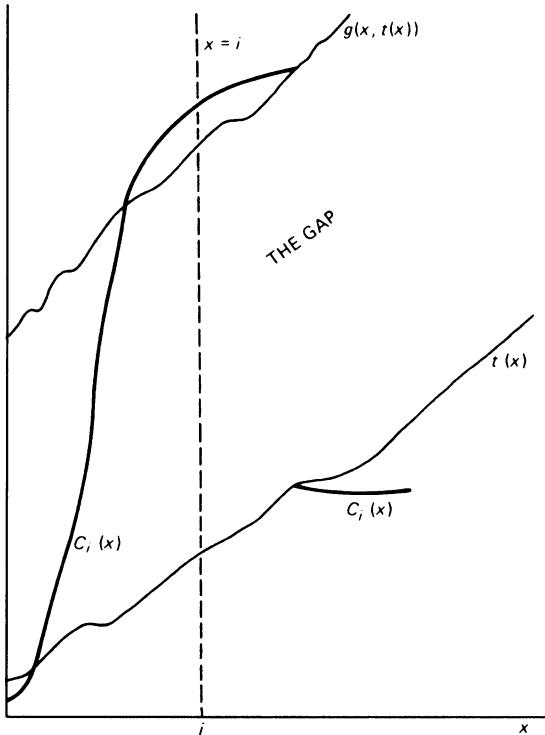


Figure 2.1. For $x > i$, $C_i(x)$ cannot enter the “gap.”

claim that $P(x, y_0)$ is true. To see this, choose $i < x$. Then if $\Phi_i(x) \downarrow$, then $C_i(x) \downarrow$ and therefore $C_i(x) \leq y_0$. If, on the other hand, $\Phi_i(x) \uparrow$ then $C_i(x) \uparrow$. Since $g(x, y_0) \downarrow$, the predicate $g(x, y_0) \leq C_i(x)$ is true. Thus, we have $P(x, y_0)$. We have shown that for every $x \in N$ there is a number y such that $P(x, y)$. Thus, $t(x)$ defined by (2.1) is total and therefore recursive.

Now let $x > i$ and $C_i(x) < g(x, t(x))$. Since $P(x, t(x))$ is true, and $i < x$, we have $C_i(x) \leq t(x) \vee g(x, t(x)) \leq C_i(x)$. But $C_i(x) < g(x, t(x))$. Hence $C_i(x) \leq t(x)$. ■

In their fine book, Machtey and Young (see “Suggestions for Further Reading”) give an amusing interpretation of the gap theorem. Let us imagine two computers, one of which is very much faster than the other. We think of each computer equipped with a reasonably efficient interpreter for our programming language \mathcal{S} so that we can speak of running a program of \mathcal{S} on one or another of the computers. Let $C_i(x)$ be the computation time of the slow computer running program number i on input x . Similarly for $D_i(x)$ and the fast computer. Clearly, C and D satisfy the Blum axioms. By the recursive relatedness theorem, there is a recursive function r satisfying (1.1). If we let $g(x, y) = r(x, y) + y + 1$, then we have $g(x, y) > y$, $g(x, y + 1) > g(x, y)$ and

$$C_i(x) \leq r(x, D_i(x)) < g(x, D_i(x)) \quad \text{a.e.}$$

Now let $t(x)$ satisfy the gap theorem for the complexity measure C with respect to this function g . And consider a program \mathcal{P} with number i such that $D_i(x) \leq t(x)$ a.e. That is, for sufficiently large inputs x , \mathcal{P} runs on the fast machine in time bounded by $t(x)$. Then on the slow computer, \mathcal{P} will run in time

$$C_i(x) < g(x, D_i(x)) \leq g(x, t(x)) \quad \text{a.e.}$$

But now the gap theorem comes into play to assure us that

$$C_i(x) \leq t(x) \quad \text{a.e.}$$

Conclusion: *Any program that runs in time $t(x)$ on the fast computer also runs in time $t(x)$ (for sufficiently large x) on the slow computer!*

Exercises

- Let C be a complexity measure. Does the gap theorem imply that there is no program number i such that $|x|^2 \leq C_i(x) \leq |x|^3$ a.e.? Explain.

2. Let C be a complexity measure. We will say that a total function $f(x)$ is C -constructible if there is a program number i such that $C_i(x) = f(x)$ for all x . Prove or disprove that every computable function is C -constructible.

3. Preliminary Form of the Speedup Theorem

Computer scientists often seek programs that will obtain a desired result using minimum resources. The speedup theorem, which is the deepest theorem in this chapter, tells us that it is possible for there to be no best program for this purpose. Roughly speaking, the theorem states that there exists a recursive function that is so badly behaved that for every program to compute it, there is another program that computes the same function but which uses much less resources. The proof of the speedup theorem is quite intricate. In this section we will prove a preliminary version. Then in the next section we will use this preliminary version to obtain the full speedup theorem. The proof of the speedup theorem will use the parameter theorem and the recursion theorem from Chapter 4 (Theorems 5.1 and 8.1).

We define a particular complexity measure $M_i(x)$ as follows. If $\Phi_i(x) \uparrow$, then $M_i(x) \uparrow$. If $\Phi_i(x) \downarrow$, then $M_i(x)$ is the largest value assumed by any variable in program number i when computing with input x . Thus $M_i(x)$ is just the complexity measure in Example 2 of Section 1. We will also work with $M_i^{(2)}(x_1, x_2)$, which is defined exactly like $M_i(x)$ except that program number i is given the pair of inputs x_1, x_2 . $M_i(x)$ and $M_i^{(2)}(x_1, x_2)$ are related by

Theorem 3.1. $M_i^{(2)}(x, y) = M_{S_1^1(y, i)}(x)$, where S_1^1 is the function defined in the parameter theorem.

Proof. Let $i = \#(\mathcal{P}_0)$. Then, examining the *proof* of Theorem 5.1 in Chapter 4, we see that $S_1^1(y, i) = \#(\mathcal{P})$, where \mathcal{P} is a program consisting of y copies of the instruction $X_2 \leftarrow X_2 + 1$ followed by the program \mathcal{P}_0 . The result is now obvious. ■

Our preliminary form of the speedup theorem is as follows.

Theorem 3.2. Let $g(x, y)$ be any given recursive function. Then there is a recursive function $f(x)$ such that $f(x) \leq x$ and, whenever $\Phi_i = f$, there is

a j such that

$$\Phi_j(x) = f(x) \quad \text{a.e.} \quad (3.1)$$

and

$$g(x, M_j(x)) \leq M_i(x) \quad \text{a.e.} \quad (3.2)$$

Discussion. To see the force of the theorem take $g(x, y) = 2^y$. Then, given $\Phi_i = f$, there is a j satisfying (3.1) such that

$$2^{M_j(x)} \leq M_i(x) \quad \text{a.e.,}$$

i.e.,

$$M_j(x) \leq \log_2 M_i(x) \quad \text{a.e.}$$

Thus program number j computes f a.e. and uses far less resources than program number i . In Section 4 we shall improve this preliminary version of the speedup theorem by eliminating the “a.e.” condition in (3.1) and by obtaining (3.2) for an arbitrary complexity measure, not merely for M .

The proof of Theorem 3.2 will use a diagonal argument, but one far more complex than we have encountered so far. Let us recall how a simple diagonal argument works. When we write

$$\bar{K} = \{n \in N \mid n \notin W_n\}$$

we know that \bar{K} is not r.e. because it differs from each r.e. set W_i with respect to the number i , namely, $i \in W_i$ if and only if $i \notin \bar{K}$. More generally, a diagonal argument constructs an object that is guaranteed not to belong to a given class by systematically ensuring that the object differs in some way from each member of the class. More intricate diagonal arguments often are carried out in an infinite sequence of stages; at each stage one seeks to ensure that the object being constructed is different from some particular member of the class. The proof of the speedup theorem is of this character.

Proof of Theorem 3.2. We will proceed through “stages” $x = 0, 1, 2, 3, \dots$. At each stage x and for certain $n, w \in N$, we will define a set $C(n, w, x) \subseteq N$. We think of the members of $C(n, w, x)$ as numbers of programs which are *cancelled* at stage x with respect to n and w . $C(n, w, x)$ is defined

recursively by the equation

$$\begin{aligned} C(n, w, x) = & \left\{ i \in N \mid w \leq i < x \text{ \& } i \notin \bigcup_{y < x} C(n, w, y) \right. \\ & \left. \text{ \& } M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\}. \end{aligned} \quad (3.3)$$

We think of C as a 3-ary partial function on N . (The fact that the values of C are finite subsets of N instead of numbers is of no importance. Naturally, if we wished, we could use some coding device to represent each finite subset of N by a particular number.) The three conditions in (3.3) connected by $\&$ are to be tested in order with the understanding that if the first or second condition is false, the succeeding conditions are simply not tested. Thus we have

$$w \geq x \quad \text{implies} \quad C(n, w, x) = \emptyset \quad \text{for all } n. \quad (3.4)$$

Moreover, we have obviously

Lemma 1. If $C(n, w, y) \downarrow$ for all $y < x$ and $M_n^{(2)}(x, i + 1) \downarrow$ for all i such that $w \leq i < x$, then $C(n, w, x) \downarrow$.

Indeed, when the conditions of Lemma 1 are satisfied, we can explicitly compute $C(n, w, x)$ given knowledge of $C(n, w, y)$ for $y < x$. Now clearly, when the conditions of Lemma 1 are not satisfied, $C(n, w, x) \uparrow$. Thus (3.3) can be used to give an algorithm for computing C and we may conclude that C is a partially computable function.

Lemma 2. If $i \in C(n, w, x)$, then $M_i(x) \downarrow$ and $\Phi_i(x) \downarrow$.

Proof. The truth of the condition

$$M_i(x) < g(x, M_n^{(2)}(x, i + 1))$$

implies that $M_i(x) \downarrow$, and by the Blum axioms, this implies $\Phi_i(x) \downarrow$. ■

We shall now define a 3-ary partially computable function k on N such that if $C(n, w, x) \downarrow$, then for each $i \in C(n, w, x)$, we will have $k(x, w, n) \neq \Phi_i(x)$. k is computed by using the following procedure:

Compute $C(n, w, x)$. If this computation terminates, compute $\Phi_i(x)$ for each $i \in C(n, w, x)$. [By Lemma 2, each such $\Phi_i(x) \downarrow$.] Finally, set $k(x, w, n)$ equal to the least number which is not a member of the finite set

$$\{\Phi_i(x) \mid i \in C(n, w, x)\}.$$

It is to this function k that we apply the recursion theorem. Thus, we obtain a number e such that

$$\Phi_e^{(2)}(x, w) = k(x, w, e). \quad (3.5)$$

Lemma 3. If $x \leq w$, then $k(x, w, e) = 0$.

Proof. Let $x \leq w$. By (3.4), $C(e, w, x) = \emptyset$. Hence, by definition, $k(x, w, e)$ is the least number which does not belong to \emptyset , namely, 0. ■

Lemma 4. If $k(x, w, e) \downarrow$, then $k(x, w, e) \leq x$.

Proof. The largest possible value for $k(x, w, e)$ would be obtained if the values $\Phi_i(x)$ for $i \in C(e, w, x)$ were all different and were consecutive numbers beginning with 0. In this “worst” case, there would be as many values of $\Phi_i(x)$ as in the set $C(e, w, x)$. But,

$$\begin{aligned} C(e, w, x) &\subseteq \{i \in N \mid w \leq i < x\} \\ &\subseteq \{0, 1, 2, \dots, x - 1\}. \end{aligned}$$

Thus, all the values of $\Phi_i(x)$ would be $< x$ and hence $k(x, w, e) \leq x$. ■

Lemma 5. Let $x > w$. Suppose that

$$\Phi_e^{(2)}(x, w + 1) \downarrow, \Phi_e^{(2)}(x, w + 2) \downarrow, \dots, \Phi_e^{(2)}(x, x) \downarrow \quad (3.6)$$

and

$$\Phi_e^{(2)}(0, w) \downarrow, \Phi_e^{(2)}(1, w) \downarrow, \dots, \Phi_e^{(2)}(x - 1, w) \downarrow. \quad (3.7)$$

Then, $\Phi_e^{(2)}(x, w) \downarrow$, i.e., $k(x, w, e) \downarrow$.

The reader is referred to Fig. 3.1 in connection with this lemma. In effect, Lemma 5 states that if $\Phi_e^{(2)}$ is defined along both the horizontal and vertical “pincers” shown pointing at (x, w) , then it must also be defined at (x, w) .

Proof of Lemma 5. By (3.7), $\Phi_e^{(2)}(y, w) \downarrow$ for all $y < x$. By definition of $k(y, w, e) = \Phi_e^{(2)}(y, w)$, we have that $C(e, w, y) \downarrow$ for all $y < x$. By (3.6), $\Phi_e^{(2)}(x, i + 1) \downarrow$ for all i such that $w \leq i < x$. Hence, likewise, $M_e^{(2)}(x, i + 1) \downarrow$ for these i . By Lemma 1, $C(e, w, x) \downarrow$. But now, by definition of k , $k(x, w, e) \downarrow$. ■

Lemma 6. $\Phi_e^{(2)}$ is total.

Proof. We shall prove by induction on x the assertion

$$\text{For all } w, \quad \Phi_e^{(2)}(x, w) \downarrow. \quad (3.8)$$

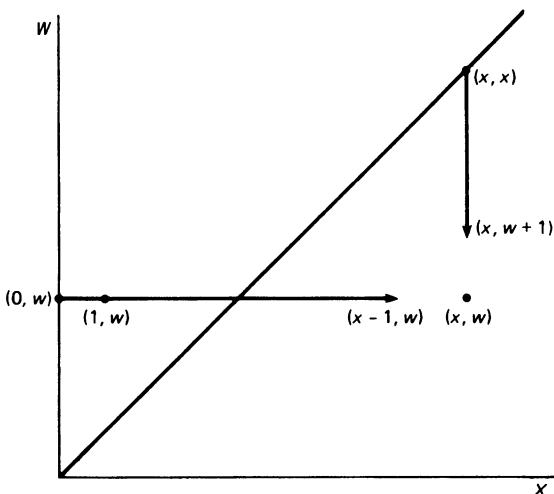


Figure 3.1. Horizontal and vertical “pincers” pointing at (x, w) . (See Lemma 5.)

By Lemma 3, we have

$$\Phi_e^{(2)}(0, w) = k(0, w, e) = 0,$$

which gives the result for $x = 0$. Suppose that $x > 0$, and it is known that

$$\Phi_e^{(2)}(y, w) \downarrow$$

for all $y < x$ and all w . We shall show that (3.8) then follows.

By Lemma 3, (3.8) holds for all $w \geq x$. Thus, we need show only that (3.8) holds for $w < x$. That is, it suffices to show that

$$\Phi_e^{(2)}(x, x - 1) \downarrow, \dots, \Phi_e^{(2)}(x, 0) \downarrow.$$

We will prove each of these in succession by using Lemma 5. That is, in Lemma 5, we successively set $w = x - 1, x - 2, \dots, 0$. In each case (3.7) (the horizontal “pincer”) is satisfied by the induction hypothesis. For $w = x - 1$, (3.6) requires only that $\Phi_e^{(2)}(x, x) \downarrow$, and this last follows at once from Lemma 3. Thus by Lemma 5, $\Phi_e^{(2)}(x, x - 1) \downarrow$. But this means that (3.6) is now satisfied with $w = x - 2$. Hence once again Lemma 5 shows that $\Phi_e^{(2)}(x, x - 2) \downarrow$. Continuing, we eventually obtain $\Phi_e^{(2)}(x, 0) \downarrow$. ■

For the remainder of the proof of Theorem 3.1, we will use the notation

$$I_w = \{i \in N \mid i < w\} = \{0, 1, \dots, w - 1\}.$$

Lemma 7. $C(n, w, x) = C(n, 0, x) - I_w$.

Proof. The proof is by induction on x . $C(n, w, 0) = \emptyset$ for all n, w . Hence the result for $x = 0$ is trivially true. Suppose the result known for all $y < x$. We obtain the result for x as follows (noting $\{i \in N \mid w \leq i < x\} \cap I_w = \emptyset$):

$$\begin{aligned} C(n, w, x) &= \left\{ i \in N \mid w \leq i < x \text{ } \& \text{ } i \notin \bigcup_{y < x} C(n, w, y) \right. \\ &\quad \left. \& M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\} \\ &= \left\{ i \in N \mid w \leq i < x \text{ } \& \text{ } i \notin \bigcup_{y < x} (C(n, 0, y) - I_w) \right. \\ &\quad \left. \& M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\} \\ &= \left\{ i \in N \mid w \leq i < x \text{ } \& \text{ } i \notin \bigcup_{y < x} C(n, 0, y) \right. \\ &\quad \left. \& M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\} \\ &= \left\{ i \in N \mid 0 \leq i < x \text{ } \& \text{ } i \notin \bigcup_{y < x} C(n, 0, y) \right. \\ &\quad \left. \& M_i(x) < g(x, M_n^{(2)}(x, i + 1)) \right\} - I_w \\ &= C(n, 0, x) - I_w. \end{aligned}$$

■

Lemma 8. For each $w \in N$, there is a number m_w such that for all $x > m_w$, we have

$$\Phi_e^{(2)}(x, w) = \Phi_e^{(2)}(x, 0).$$

Proof. By (3.3) [the definition of $C(n, w, x)$], we have $C(e, 0, x) \cap C(e, 0, y) = \emptyset$ for $x \neq y$. [Numbers in $C(e, 0, y)$ for $y < x$ are automatically excluded from $C(e, 0, x)$.] Hence each number in I_w belongs to at most one of the sets $C(e, 0, x)$. If we let m_w be the largest such value of x , then for $x > m_w$,

$$C(e, 0, x) \cap I_w = \emptyset.$$

Hence, using Lemma 7, for $x > m_w$,

$$C(e, w, x) = C(e, 0, x) - I_w = C(e, 0, x).$$

Hence, by the definition of the function k we have for $x > m_w$,

$$\Phi_e^{(2)}(x, w) = k(x, w, e) = k(x, 0, e) = \Phi_e^{(2)}(x, 0). \quad \blacksquare$$

Note that there is no claim being made that m_w is a computable function of w , and indeed it is not!

We are now ready to define the function $f(x)$ whose existence is asserted in Theorem 3.2. We set

$$f(x) = \Phi_e^{(2)}(x, 0).$$

Lemma 9. If $\Phi_i = f$ and $x > i$, then

$$g(x, M_e^{(2)}(x, i + 1)) \leq M_i(x).$$

Proof. Suppose otherwise. Choose the least value of $x > i$ with

$$g(x, M_e^{(2)}(x, i + 1)) > M_i(x). \quad (3.9)$$

Then we claim that for $y < x$, $i \notin C(e, 0, y)$. This is because

$$C(e, 0, y) = \left\{ j \in N \mid j < y \text{ } \& \text{ } j \notin \bigcup_{z < y} C(e, 0, z) \right. \\ \left. \& M_j(y) < g(y, M_e^{(2)}(y, j + 1)) \right\},$$

so that, if $i \in C(e, 0, y)$, we would have $i < y < x$, and

$$g(y, M_e^{(2)}(y, i + 1)) > M_i(y),$$

contradicting the choice of x as the least number $> i$ satisfying (3.9). Thus, we have

$$i \notin \bigcup_{y < x} C(e, 0, y).$$

Hence,

$$i \in C(e, 0, x) = \left\{ j \in N \mid j < x \text{ } \& \text{ } j \notin \bigcup_{y < x} C(e, 0, y) \right. \\ \left. \& M_j(x) < g(x, M_e^{(2)}(x, j + 1)) \right\}.$$

Now $k(x, 0, e)$ was defined to be different from all $\Phi_j(x)$ for which $j \in C(e, 0, x)$. Hence, $k(x, 0, e) \neq \Phi_i(x)$. But

$$k(x, 0, e) = \Phi_e^{(2)}(x, 0) = f(x) = \Phi_i(x),$$

This contradiction completes the proof.

Proof of Theorem 3.2 Concluded. Let $\Phi_i = f$, and set $j = S_1^1(i + 1, e)$. Then, by Theorem 3.1 and Lemma 9, we have for $x > i$,

$$g(x, M_j(x)) = g(x, M_e^{(2)}(x, i + 1)) \leq M_i(x),$$

which proves (3.2). Finally, using the parameter theorem (Theorem 5.1 in Chapter 4) and Lemma 8 we have for $x > m_{i+1}$,

$$\Phi_j(x) = \Phi_e^{(2)}(x, i + 1) = \Phi_e^{(2)}(x, 0) = f(x),$$

which proves (3.1). ■

4. The Speedup Theorem Concluded

We will begin by showing how to eliminate the a.e. from Eq. (3.1) in Theorem 3.2. The technique we will use is a general one; to change a condition

$$\Phi_j(x) = f(x) \quad \text{a.e.}$$

into an equation valid everywhere, we need only modify program number j to agree with $f(x)$ at a finite number of values. We can do this by patching in a “table look-up” program. More precisely, we have

Theorem 4.1. There is a recursive function $t(u, w)$ such that

$$\begin{aligned} \Phi_{t(u,w)}(x) &= \begin{cases} \Phi_u(x) & \text{if } x > l(w) \\ (r(w))_{x+1} & \text{if } x \leq l(w), \end{cases} \\ M_{t(u,w)}(x) &= M_u(x) \quad \text{if } x > l(w). \end{aligned}$$

Here, once again we are using the pairing functions and Gödel numbers as coding devices (Chapter 3, Section 8).

Proof. Let the numbers u, w be given. Let P_u be program number u of the language \mathcal{S} , if this program begins with a labeled statement. Otherwise let P_u be program number u modified by having its initial statement labeled by a label not otherwise occurring in the program. In either case let L be the label with which P_u begins.

Let $Q_{u,w}$ be a program of \mathcal{S} which computes the primitive recursive function $(r(w))_{x+1}$, which always terminates using a branch instruction, and which has no labels in common with P_u . Let V be a local variable that occurs neither in P_u nor in $Q_{u,w}$. Let $t(u, w)$ be the number of the program indicated in Fig. 4.1. Note that $V \leftarrow X$ is to be replaced by a

```

 $V \leftarrow X$ 
 $V \leftarrow V - 1$ 
 $V \leftarrow V - 1$ 
 $\vdots$ 
 $V \leftarrow V - 1$ 
IF  $V \neq 0$  GOTO  $L$ 
 $Q_{u,w}$ 
 $P_u$ 

```

Figure 4.1

suitable macro expansion as in Chapter 2 and that there are $l(w)$ statements $V \leftarrow V - 1$. Clearly this can all be done with t a recursive (even primitive recursive) function.

Now, let $x > l(w)$. Then after the $l(w)$ decrement instructions $V \leftarrow V - 1$ have been executed, V will have the value $x - l(w) > 0$. Hence, the branch shown will be taken and program P_u will be executed. Hence, $\Phi_{t(u,w)}(x) = \Phi_u(x)$. To compare the value of $M_{t(u,w)}(x)$ and $M_u(x)$ we need to be concerned about the maximum value assumed by variables in the macro expansion of $V \leftarrow X$. Examining this macro expansion as given in (c) in Chapter 2, Section 2, we see that the only possibility for a number $> x$ to arise is in the case $x = 0$. This is because local variables need to be incremented to 1 in this macro expansion in order to force a branch to be taken.² However, we are assuming $x > l(w) \geq 0$, so that $x \neq 0$. Hence, $M_{t(u,w)}(x) = M_u(x)$.

Finally, let $x \leq l(w)$. Then after $l(w)$ executions of $V \leftarrow V - 1$, V has the value 0. Thus $Q_{u,w}$ is executed. Hence, $\Phi_{t(u,w)}(x) = (r(w))_x$. ■

Now we can easily prove

Theorem 4.2. Let $g(x, y)$ be any given recursive function. Then there is a recursive $f(x)$ such that $f(x) \leq x$ and, whenever $\Phi_i = f$, there is a j such that

$$\Phi_j(x) = f(x)$$

and

$$g(x, M_j(x)) \leq M_i(x) \quad \text{a.e.}$$

² Actually, if each unconditional branch statement in program (c), Chapter 2, Section 2, is directly expanded, some of the local variables used in this expansion will reach values > 1 . The simplest way to get around this is to place the single statement $Z_2 \leftarrow Z_2 + 1$ at the beginning of this program and then to replace each of the four unconditional branch statements GOTO L by the corresponding conditional branch statement IF $Z_2 \neq 0$ GOTO L .

Proof. Let f be as in Theorem 3.2, and suppose $\Phi_i = f$. Then there is $j \in N$ such that (3.1) and (3.2) hold. Let $\Phi_j(x) = f(x)$ for $x > x_0$. Let

$$w = \langle x_0, [f(0), \dots, f(x_0)] \rangle.$$

Finally, let $\bar{j} = t(j, w)$. Then using Theorem 4.1,

$$\Phi_{\bar{j}}(x) = \Phi_{t(j, w)}(x) = \begin{cases} \Phi_j(x) & \text{if } x > x_0 \\ f(x) & \text{if } x \leq x_0 \end{cases},$$

i.e., $\Phi_{\bar{j}} = f$. Theorem 4.1 also implies that $M_{\bar{j}}(x) = M_j(x)$ a.e. Hence, using (3.2), we have almost everywhere

$$g(x, M_{\bar{j}}(x)) = g(x, M_j(x)) \leq M_i(x). \quad \blacksquare$$

Finally, we are ready to give the speedup theorem for arbitrary complexity measures.

Theorem 4.3 (Blum Speedup Theorem). Let $g(x, y)$ be any given recursive function and let C be any complexity measure. Then there is a recursive function $f(x)$ such that $f(x) \leq x$ and whenever $\Phi_i = f$, there is a j such that

$$\Phi_j(x) = f(x)$$

and

$$g(x, C_j(x)) \leq C_i(x) \quad \text{a.e.}$$

Proof. Using the recursive relatedness theorem (Theorem 1.2), there is a recursive function $r(x, y)$ such that

$$\begin{aligned} r(x, y) &< r(x, y + 1), \\ C_i(x) &\leq r(x, M_i(x)) \quad \text{a.e.} \\ M_i(x) &\leq r(x, C_i(x)) \quad \text{a.e.} \end{aligned}$$

Let

$$h(x, y) = \sum_{z \leq y} g(x, z),$$

so that h is recursive,

$$h(x, y) \geq g(x, y),$$

and

$$h(x, y + 1) \geq h(x, y).$$

Finally, let

$$\bar{g}(x, y) = r(x, h(x, r(x, y))).$$

Now, we apply Theorem 4.2 using \bar{g} as the given function g . Let $f(x)$ be the recursive function obtained, so that $f(x) \leq x$. Let $\Phi_i = f$. Then there is a j such that $\Phi_j = f$ and

$$\bar{g}(x, M_j(x)) \leq M_i(x) \quad \text{a.e.}$$

Hence, we have, almost everywhere,

$$\begin{aligned} r(x, g(x, C_j(x))) &\leq r(x, h(x, r(x, C_j(x)))) \\ &\leq r(x, h(x, r(x, M_j(x)))) \\ &= \bar{g}(x, M_j(x)) \\ &\leq M_i(x) \leq r(x, C_i(x)). \end{aligned}$$

Now, if $C_i(x) < g(x, C_j(x))$ for any value of x , we would have, for that value of x ,

$$r(x, C_i(x)) < r(x, g(x, C_j(x))).$$

Hence, we must have, almost everywhere,

$$g(x, C_j(x)) \leq C_i(x). \quad \blacksquare$$

Exercises

1. Show that for all $i \in N$ there is a j such that $\Phi_j(x) = M_i(x)$ and $\Phi_j(x) = M_j(x)$ for all x . Conclude that every function $M_i(x)$ has an “optimal” program with respect to complexity measure M .
2. Let L be the set of all strings that are syntactically correct Pascal programs, and let

$$P(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{otherwise.} \end{cases}$$

Does the speedup theorem imply that there is no fastest \mathcal{S} program that computes $P(x)$? Explain.

15

Polynomial–Time Computability

1. Rates of Growth

In this chapter we will be working with functions f such that $f(n) \in N$ for all sufficiently large $n \in N$, but which may be undefined or have negative values for some finite number of values of n . We refer to such functions briefly, and slightly inaccurately, as functions from N to N . These functions f will typically have the additional property

$$\lim_{n \rightarrow \infty} f(n) = \infty. \quad (1.1)$$

Examples of such functions are n^2 , 2^n , and $\lfloor \log_2 n \rfloor$. It will be important for us to understand in what sense we can say that 2^n grows faster than n^2 and that n^2 grows faster than $\lfloor \log_2 n \rfloor$. Although in practice, the definitions we are about to give are of interest only for functions that satisfy (1.1), our definitions will not assume that this is the case.

Definition. Let f, g be functions from N to N . Then, we say that $f(n) = O(g(n))$ if there are numbers c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$. If these conditions do not hold we say that $f(n) \neq O(g(n))$.

If $f(n) = O(g(n))$ and $g(n) = O(f(n))$ we say that f and g have the same rate of growth. On the other hand, if $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$, we say that $g(n)$ grows faster than $f(n)$.

An example should help clarify these notions. We have

$$n^2 = O(3n^2 - 6n + 5)$$

since

$$\frac{n^2}{3n^2 - 6n + 5} = \frac{1}{3 - 6/n + 5/n^2} \rightarrow \frac{1}{3}$$

as $n \rightarrow \infty$, and therefore there is a number n_0 such that for all $n \geq n_0$,

$$\frac{n^2}{3n^2 - 6n + 5} \leq 1.$$

Likewise $3n^2 - 6n + 5 = O(n^2)$, so that these two functions have the same rate of growth.

Clearly, it is also true that $3n^2 - 6n + 5 = O(n^3)$; however,

$$n^3 \neq O(3n^2 - 6n + 5)$$

because

$$\frac{n^3}{3n^2 - 6n + 5} = n \cdot \frac{1}{3 - 6/n + 5/n^2} \rightarrow \infty$$

as $n \rightarrow \infty$. Thus, we can say that n^3 grows faster than $3n^2 - 6n + 5$.

More generally, we can prove

Theorem 1.1. Let f, g be functions from N to N , and let

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \beta, \quad (1.2)$$

where β is a positive real number. Then $f(n) = O(g(n))$ and $g(n) = O(f(n))$, so that f and g have the same rate of growth.

If, on the other hand,

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty, \quad (1.3)$$

then $g(n) = O(f(n))$ but $f(n) \neq O(g(n))$, so that $f(n)$ grows faster than $g(n)$.

Proof. If (1.2) holds, then there is a number n_0 such that for all $n \geq n_0$,

$$\frac{f(n)}{g(n)} \leq \beta + 1.$$

Hence, $f(n) = O(g(n))$. Since (1.2) implies that

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \frac{1}{\beta},$$

the same reasoning can be used to show that $g(n) = O(f(n))$.

Next, (1.3) implies that

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0.$$

Therefore, there is a number n_0 such that $n \geq n_0$ implies

$$\frac{g(n)}{f(n)} \leq 1.$$

Hence, $g(n) = O(f(n))$. If we had also $f(n) = O(g(n))$, then for numbers c , n_0 we should have for $n \geq n_0$,

$$\frac{f(n)}{g(n)} \leq c;$$

on the other hand, (1.3) implies that there is a number n_1 such that $n \geq n_1$ implies

$$\frac{f(n)}{g(n)} > c,$$

which is a contradiction. ■

A *polynomial* is a function p from N to N that is defined by a formula of the form

$$p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_r n^r, \quad (1.4)$$

where a_0, a_1, \dots, a_{r-1} are integers, positive, negative, or zero, while a_r is a positive integer. In this case the number r is called the *degree* of the polynomial p . The degree of a polynomial determines its rate of growth in the following precise sense.

Theorem 1.2. Let p be a polynomial of degree r . Then p and n^r have the same rate of growth. Moreover, p grows faster than n^m if $m < r$, and n^m grows faster than p if $m > r$.

Proof. Letting p be as in (1.4), we have

$$\frac{p(n)}{n^r} = \frac{a_0}{n^r} + \frac{a_1}{n^{r-1}} + \cdots + \frac{a_r}{n^0} \rightarrow a_r$$

as $n \rightarrow \infty$. Also,

$$\frac{p(n)}{n^m} = \frac{p(n)}{n^r} \cdot n^{r-m},$$

so that

$$\frac{p(n)}{n^m} \rightarrow \infty \quad \text{if } r > m, \quad \text{and} \quad \frac{p(n)}{n^m} \rightarrow 0 \quad \text{if } r < m.$$

The result then follows from Theorem 1.1. ■

Next we shall see that exponential functions grow faster than any fixed power.

Theorem 1.3. The function k^n , with $k > 1$, grows faster than any polynomial.

Proof. It clearly suffices to prove that for any $r \in N$,

$$\lim_{n \rightarrow \infty} \frac{k^n}{n^r} = \infty.$$

One way to obtain this result is to use L'Hospital's rule from calculus; on differentiating the numerator and denominator of this fraction r times, a fraction is obtained whose numerator approaches infinity and whose denominator is a constant (in fact, $r!$). To obtain the result directly, we first prove the following lemma.

Lemma. Let g be a function from N to N such that

$$\lim_{n \rightarrow \infty} \frac{g(n+1)}{g(n)} = \beta > 1.$$

Then $g(n) \rightarrow \infty$ as $n \rightarrow \infty$.

Proof of Lemma. Let γ be a number strictly between 1 and β , for example, $\gamma = (1 + \beta)/2$. Then there is a number n_0 such that $n \geq n_0$ implies

$$\frac{g(n+1)}{g(n)} \geq \gamma.$$

Thus, for each m ,

$$g(n_0 + m) \geq \gamma g(n_0 + m - 1) \geq \dots \geq \gamma^m g(n_0).$$

Since $\gamma^m \rightarrow \infty$ as $m \rightarrow \infty$, the result follows. ■

Proof of Theorem 1.3 Concluded. Setting

$$g(n) = k^n/n^r,$$

we have

$$\frac{g(n+1)}{g(n)} = \frac{k}{\left(1 + \frac{1}{n}\right)^r} \rightarrow k \quad \text{as } n \rightarrow \infty,$$

which, by the lemma, gives the result. ■

Exercises

1. Suppose we have a computer that executes 1 million instructions per second.
 - (a) For each of the following functions $f(x)$, give the length of the longest string that can be processed in one hour if $f(|w|)$ instructions are required to process a string w : $f(x) = x$; $f(x) = x^2$; $f(x) = x^4$; $f(x) = 2^x$.
 - (b) For the same functions, approximately how long would it take to process w if $|w| = 100$?
2. What is the least $x \in N$ such that $10000x^2 \leq 2^x$?
3. For each of the following functions $f(x)$, give a function $g(x)$ such that some Turing machine on a two-symbol alphabet can calculate $f(x)$ in $O(g(|x|))$ steps: $f(x) = 2x$; $f(x) = x^2$; $f(x) = 2^x$; $f(x) = 2^{(2^x)}$.
4. (a) Show that if $p(n)$ is defined by (1.4), then $p(n)$ is positive for n sufficiently large, so that p is a function from N to N in the sense defined at the beginning of this chapter.
 - (b) Show that if $p(n)$ is as in (a) with $r > 0$, then $p(n) \rightarrow \infty$ as $n \rightarrow \infty$.
5. Show that n grows faster than $\lfloor \log_2 n \rfloor$.
6. Show that for any $k \geq 1$ and any polynomials $p(x)$, $q(x)$, there is a polynomial $r(x)$ such that $q(x) \cdot k^{p(x)} = O(2^{r(x)})$.

2. P versus NP

Computability theory has enabled us to distinguish clearly and precisely between problems for which there are algorithms and those for which

there are none. However, there is a great deal of difference between solvability “in principle,” with which computability theory deals, and solvability “in practice,” which is a matter of obtaining an algorithm that can be implemented to run using space and time resources likely to be available. It has become customary to speak of problems that are solvable, not only in principle but also in practice, as *tractable*; problems that may be solvable in principle but are not solvable in practice are then called *intractable*.

The satisfiability problem, discussed in Chapter 12, is an example that is illuminating in this connection and will, in fact, play a central role in this chapter. The satisfiability problem is certainly *solvable*; in Chapter 12, we discussed algorithms for testing a given formula in CNF for satisfiability based on truth tables, on converting to DNF, on resolution, and on the Davis–Putnam rules. However, we cannot claim that the satisfiability problem is *tractable* on the basis of any of these algorithms or, for that matter, on the basis of any known algorithm. As we have seen, procedures based on truth tables or DNF require a number of steps which is an *exponential* function of the length of the expression representing a given formula in CNF. It is because of the rapid growth of the exponential function that these procedures can quickly exhaust available resources. Procedures based on resolution or on the Davis–Putnam rules can be designed that work well on “typical” formulas. However, no one has succeeded in designing such a procedure for which it can be proved that exponential behavior never arises, and it is widely believed (for reasons that will be indicated later) that every possible procedure for the satisfiability problem behaves exponentially in some cases. Thus the satisfiability problem is regarded as a prime candidate for intractability, although the matter remains far from being settled.

This association of intractability with the exponential function, coupled with the fact (Theorem 1.3) that an exponential function grows faster than any polynomial function, suggests that a problem be regarded as *tractable* if there is an algorithm that solves it which requires a number of steps bounded by some polynomial in the length of the input.

To make these ideas precise, we have recourse to the Turing machine model of computation as developed in Chapter 6. In particular, we shall use the terms *configuration* and *computation* as in Chapter 6.

Definition. A language L on an alphabet A is said to be *polynomial-time decidable* if there is a Turing machine \mathcal{M} that accepts L , and a polynomial $p(n)$, such that the number of steps in an accepting computation by \mathcal{M} with input x is $\leq p(|x|)$. When the alphabet is understood, we write \mathbf{P} for the class of polynomial–time decidable languages.

Definition. A total function f on A^* , where A is an alphabet, is said to be *polynomial-time computable* if there is a Turing machine \mathcal{M} that computes f , and a polynomial $p(n)$, such that the number of steps in the computation by \mathcal{M} with input x is $\leq p(|x|)$.

With respect to both of these definitions, we note

1. It suffices that there exist a polynomial $p(n)$ such that the number of steps in the computation by \mathcal{M} with input x is $\leq p(|x|)$ for all but a finite number of input strings x . For, in such a case, to include the finite number of omitted cases as well, we let c be the largest number of steps used by \mathcal{M} in these cases, and replace $p(n)$ by the polynomial $p(n) + c$.
2. Using 1 and Theorem 1.2, it suffices that the number of steps be $O(|x|^r)$ for some $r \in N$.

The discussion leading to these definitions suggests that in analogy with Church's thesis, we consider the

Cook-Karp Thesis. The problem of determining membership of strings in a given language L is tractable if and only if $L \in \mathbf{P}$.

The evidence supporting the Cook-Karp thesis is much weaker than that supporting Church's thesis. Nevertheless, it has gained wide acceptance. Later, we shall discuss some of the reasons for this.

The following simple result is quite important.

Theorem 2.1. Let $L \in \mathbf{P}$, let f be a polynomial-time computable function on A^* , and let $Q = \{x \in A^* \mid f(x) \in L\}$. Then $Q \in \mathbf{P}$.

Proof. Let \mathcal{M} accept L using a number of steps which is $O(|x|^r)$, and let \mathcal{N} compute $f(x)$ in a number of steps which is $O(|x|^s)$. A Turing machine \mathcal{R} that accepts Q is easily constructed that, in effect, first runs \mathcal{N} on x to compute $f(x)$ and then runs \mathcal{M} on $f(x)$ to determine whether $f(x) \in L$. Since a Turing machine cannot print more symbols in the course of a computation than there are steps in that computation, we have

$$|f(x)| \leq |x| + p(|x|), \quad \text{where } p(n) = O(n^s).$$

By Theorem 1.2, it follows that $|f(x)| = O(|x|^s)$. Hence, the number of steps required by \mathcal{R} on input x is $O(|x|^{sr})$. ■

Theorem 2.2. Let f, g be polynomial-time computable functions, and let $h(x) = f(g(x))$. Then h is polynomial-time computable.

Proof. The proof is similar to that of the previous theorem. ■

It has turned out to be extremely difficult to prove that specific languages do not belong to **P**, although there are many likely candidates. An important example is the satisfiability problem discussed in Chapter 12. To make matters definite, we assume a set of atoms $\mathcal{A} = \{\alpha_1, \alpha_2, \dots\}$, where subscripts are understood as in Section 1 of Chapter 12. We use the symbols

$$\alpha_1, \alpha_2, \dots, \bar{\alpha}_1, \bar{\alpha}_2, \dots$$

for the atoms and their negations, simply using concatenation for disjunction. Finally, we use the symbol $/$ to begin a clause. Then, any string on the alphabet $C = \{\alpha, \bar{\alpha}, |, /\}$ which begins $/$ and in which $/$ is never immediately followed by $|$, stands for a CNF formula (where in the interest of simplicity we are permitting empty and tautologous clauses and repetitions of literals in a clause). Thus the CNF formula

$$(p \vee q \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r} \vee s) \wedge (\bar{q} \vee \bar{p} \vee \bar{r})$$

from Chapter 12 could be written as

$$/\alpha_1 \alpha_2 \bar{\alpha}_3 \alpha_4 / \bar{\alpha}_2 \bar{\alpha}_1 \bar{\alpha}_3 \alpha_4 / \bar{\alpha}_2 \bar{\alpha}_1 \bar{\alpha}_3 .$$

Any string in C^* which ends $/$ or in which $/$ is repeated represents a CNF formula which contains the empty clause, and hence is unsatisfiable.

Now, we write SAT for the language consisting of all elements of C^* that represent satisfiable CNF formulas. In spite of a great deal of attention to the question, it is still not known whether $SAT \in P$. The starting point of the work on computational complexity that we discuss in this chapter is the observation that the situation changes entirely when we shift our attention from deterministic to nondeterministic computation. Nondeterministically one can discover very rapidly that a formula is satisfiable; it is necessary only that the satisfying assignment be “guessed.” That is, instead of constructing an entire truth table, it suffices to construct a single row. To make these ideas precise, we have recourse to nondeterministic Turing machines as discussed in Chapter 6, Section 5.

Definition. A language L is said to belong to the class **NP** if there is a nondeterministic Turing machine \mathcal{M} that accepts L , and a polynomial $p(n)$, such that for each $x \in L$, there is an accepting computation $\gamma_1, \gamma_2, \dots, \gamma_m$ by \mathcal{M} for x with $m \leq p(|x|)$.

We then have readily

Theorem 2.3. $P \subseteq NP$. If $L \in NP$, then L is recursive.

Proof. The first inclusion is obvious, since an ordinary Turing machine is a nondeterministic Turing machine.

For the rest, let $L \in \text{NP}$, let \mathcal{M} be a nondeterministic Turing machine which accepts L , with corresponding polynomial $p(n)$. We set γ_1 to be the configuration

$$\begin{array}{c} s_0 x \\ \uparrow \\ q_1 \end{array}$$

Next, by examining the quadruples of \mathcal{M} , we find all configurations γ_2 such that $\gamma_1 \vdash \gamma_2$. Continuing in this manner, we determine all possible sequences $\gamma_1, \gamma_2, \dots, \gamma_m$ with $m \leq p(|x|)$ such that

$$\gamma_1 \vdash \gamma_2 \vdash \cdots \vdash \gamma_m.$$

Then, $x \in L$ if and only if at least one of these sequences is an accepting computation by \mathcal{M} for x . This gives an algorithm for determining whether $x \in L$, and so, invoking Church's thesis, we conclude that L is recursive. (Methods like those used in Chapter 7 could be used to prove that L is recursive without using Church's thesis.) ■

In line with our discussion of the satisfiability problem viewed nondeterministically, we can prove

Theorem 2.4. $\text{SAT} \in \text{NP}$.

Proof. Without providing all the rather messy details, we indicate how to construct a nondeterministic Turing machine \mathcal{M} that accepts SAT.

\mathcal{M} will begin by checking that a given input string $x \in C^*$ really does represent a CNF formula. Such a check requires only verifying that x begins with the symbol / and that no / is immediately followed by !. This can clearly be accomplished by \mathcal{M} in a single pass over x , and therefore it can be done in $O(|x|)$ steps.

The remainder of the computation will involve successive passes over the string x in which truth values are assigned to literals, and clauses thus satisfied are labeled as being such. When a clause has been satisfied, the symbol / that introduces it is replaced by ! (so the fact that a clause still begins / indicates that it has not yet been satisfied). Also, when a literal $\alpha|^{[i]}$ is assigned the value 1, all occurrences of the literal $\bar{\alpha}|^{[i]}$ in clauses not yet satisfied will be replaced by $\varphi|^{[i]}$ (so that they will not be assigned the value 1 in a subsequent pass). Likewise, when the literal $\alpha|^{[i]}$ is assigned the value 0, all occurrences of that literal in clauses not yet satisfied will be replaced by $\varphi|^{[i]}$.

We will speak of \mathcal{M} as being in one of two modes: *search* or *update*. After verifying that the input string x does represent a CNF formula, \mathcal{M} enters search mode. In search mode, \mathcal{M} begins by finding the first occurrence of $/$ remaining in x , starting from the left. If no $/$ remains, then the formula has been satisfied and the computation halts. Otherwise, \mathcal{M} has found an $/$ and seeks to satisfy the clause that it heads. \mathcal{M} scans the clause, moving to the right. When the symbol α or $\bar{\alpha}$ is encountered, \mathcal{M} is scanning the first symbol of a literal $\alpha|^{[i]}$ or $\bar{\alpha}|^{[i]}$, as the case may be. \mathcal{M} thus has the opportunity to satisfy the clause by making this literal true, assigning $\alpha|^{[i]}$ the value 1 in the first case and 0 in the second. \mathcal{M} nondeterministically decides whether to make this assignment. (This is the only respect in which \mathcal{M} behaves nondeterministically.) If \mathcal{M} does not make the assignment, then it continues its scan. If it reaches the end of the clause without having made an assignment, \mathcal{M} enters an infinite loop. If \mathcal{M} does make such an assignment, it enters update mode.

In update mode, \mathcal{M} begins by marking the newly assigned literal, replacing α by ρ , or $\bar{\alpha}$ by $\bar{\rho}$, respectively. \mathcal{M} then moves left to the $/$ that begins the clause and replaces it by $!$. Finally, \mathcal{M} moves to the right end of x , and then scans from right to left, checking all literals in subsequent clauses to see whether they match the newly assigned literal. This can be done by checking each block of $1s$ against the block that follows ρ (or $\bar{\rho}$). For literals that have been made true by the new assignment, the clause containing them is marked as satisfied, by replacing the $/$ at its head by $!$. For literals that have been made false, the α or $\bar{\alpha}$ is replaced by φ . When the update is complete, \mathcal{M} reenters search mode.

This completes the description of how \mathcal{M} operates. It remains to estimate the number of steps that \mathcal{M} requires for a successful computation. The number of steps between \mathcal{M} entering and leaving each of search and update mode is clearly $O(|x|)$. Since this will happen no more than $|x|$ times, we conclude that the time for the entire computation is $O(|x|^2)$. ■

It is natural to ask whether the inclusion $\mathbf{P} \subseteq \mathbf{NP}$ is proper, i.e., whether there is a language L such that $L \in \mathbf{NP} - \mathbf{P}$. As we shall see, using the notion of **NP-completeness** to be defined below, it can be shown that if there were such a language, then it would follow that $SAT \in \mathbf{NP} - \mathbf{P}$. Unfortunately, this remains an open question.

Definition.¹ Let L, Q be languages. Then we write

$$Q \leq_p L,$$

¹ For a general discussion of reducibility, see Chapter 8.

and say that Q is *polynomial-time reducible* to L , if there is a polynomial-time computable function f such that

$$x \in Q \Leftrightarrow f(x) \in L.$$

Theorem 2.5. Let $R \leq_p Q$ and $Q \leq_p L$. Then $R \leq_p L$.

Proof. This follows at once from Theorem 2.2. ■

Definition. A language L is called **NP-hard** if for every $Q \in \text{NP}$, we have $Q \leq_p L$. L is called **NP-complete** if $L \in \text{NP}$ and L is NP-hard.

The significance of NP-completeness can be appreciated from the following result.

Theorem 2.6. If there is an NP-complete language L such that $L \in \text{P}$, then $\text{NP} = \text{P}$.

Proof. We need to show that if $Q \in \text{NP}$, then $Q \in \text{P}$. Let $Q \subseteq A^*$. Since L is NP-hard, there is a polynomial-time computable function f such that

$$Q = \{x \in A^* \mid f(x) \in L\}.$$

The result now follows from Theorem 2.1. ■

Intuitively, one can thus think of the NP-complete languages as the “hardest” languages in NP. As we shall see in the next section, SAT is NP-complete. Thus, if it should turn out that $SAT \in \text{P}$, then every NP-complete problem would also be in P. It is considerations like these that have led to the tentative conclusion that NP-complete problems should be regarded as being intractable. To date, however, although very many problems are known to be NP-complete, there is no language known to be in $\text{NP} - \text{P}$, and it thus remains possible that $\text{NP} = \text{P}$.

Exercises

1. Show that Theorem 2.1 still holds when P is replaced by NP.
2. Show that if $\emptyset \subset L, M \subset A^*$ for some alphabet A , and if $L, M \in \text{P}$, then $L \leq_p M$.
3. Show that $L \leq_p M$ does not necessarily imply that $M \leq_p L$.
4. Let $L, M \in \text{P}$ be languages on some alphabet A . Show that each of the following languages are in P: $A^* - L$, $L \cap M$, $L \cup M$.
5. Let $L, M \in \text{NP}$ be languages on some alphabet A . Show that each of the following languages are in NP: $L \cap M$, $L \cup M$.

6. Show that every regular language is polynomial-time decidable. [See Chapter 9.]
7. Show that every context-free language is polynomial-time decidable. [See Chapter 10.]
8. Give a language that is not polynomial-time decidable.
9. Give a function that is not polynomial-time computable.
10. Let A be an alphabet and set

$$\text{co-}\mathbf{NP} = \{L \subseteq A^* \mid A^* - L \in \mathbf{NP}\}.$$

Show that if there is a language L such that L is **NP**-complete and $L \in \text{co-}\mathbf{NP}$, then $\mathbf{NP} = \text{co-}\mathbf{NP}$.

11. Show that Theorem 2.1 still holds when **P** is replaced by **NP**.
- 12.* Let f be a total function on N , and let A be an alphabet. A total unary function $g(x)$ on A^* is computed in **DTIME**(f) if it is computed by some Turing machine that always runs in $\leq f(|x|)$ steps on input x . A language $L \subseteq A^*$ belongs to **DTIME**(f) if L is accepted by some Turing machine that runs in $\leq f(|x|)$ steps for every $x \in L$. L belongs to **NTIME**(f) if L is accepted by some nondeterministic Turing machine that has an accepting computation with $\leq f(|x|)$ steps for every $x \in L$. For languages $L, M \subseteq A^*$, we will write $L \leq_f M$ to indicate that there is a function g computable in **DTIME**(f) such that $x \in L$ if and only if $g(x) \in M$.
 - (a) Show that $\mathbf{P} = \bigcup_{n \geq 0} \mathbf{DTIME}(x^n)$.
 - (b) Show that $\mathbf{NP} = \bigcup_{n \geq 0} \mathbf{NTIME}(x^n)$.
 - (c) Prove that if $L \in \mathbf{DTIME}(x^2)$ and $M \leq_f L$, where $f(x) = x$, then $M \in \mathbf{DTIME}(4x^2 + x)$.
 - (d) Prove that if $L \in \mathbf{NTIME}(x^2)$ and $M \leq_f L$, where $f(x) = x$, then $M \in \mathbf{NTIME}(4x^2 + x)$.
 - (e) Let $f(x) = x^2$. Give a function $g(x)$ such that if $L \in \mathbf{DTIME}(x^2)$ and $M \leq_f L$, then $M \in \mathbf{DTIME}(g)$.
- 13.* A language L belongs to **EXPTIME** if there is a Turing machine \mathcal{M} that accepts L and a polynomial $p(n)$ such that for every $x \in L$, \mathcal{M} runs for no more than $2^{p(|x|)}$ steps.
 - (a) Let \mathcal{N} be a nondeterministic Turing machine with k states. For a function $f(x)$, what is the maximum number of distinct computations that \mathcal{N} can carry out in $\leq f(x)$ steps?
 - (b) Show that $\mathbf{NP} \subseteq \mathbf{EXPTIME}$. [See Exercise 1.6.]
- 14.* A language L belongs to **PSPACE** if there is a Turing machine \mathcal{M} that accepts L and a polynomial $p(n)$ such that for every $x \in L$, \mathcal{M}

scans at most $p(|x|)$ different squares on its tape. L belongs to **NPSPACE** if there is a nondeterministic Turing machine \mathcal{N} that accepts L and a polynomial $q(n)$ such that for every $x \in L$, \mathcal{N} has some accepting computation in which at most $p(|x|)$ different tape squares are scanned.

- (a) Show that **PSPACE** = **NPSPACE**.
 - (b) Show that **NP** ⊆ **PSPACE**.
- 15.* (a) Let \mathcal{M} be a Turing machine with states q_1, \dots, q_k and alphabet $\{s_1, \dots, s_n\}$. How many distinct configurations of \mathcal{M} are there with m tape squares?
- (b) Show that **PSPACE** ⊆ **EXPTIME**. [Hint: Use the pigeon-hole principle. See the discussion in Section 1 of Chapter 14.]

3. Cook's Theorem

We now prove the main theorem of this chapter.

Theorem 3.1 (Cook's Theorem). SAT is **NP**-complete.

Proof. Since we know, by Theorem 2.4, that $SAT \in \mathbf{NP}$, it remains to show that SAT is **NP-hard**. That is, we need to show that if $L \in \mathbf{NP}$, then $L \leq_p SAT$. Thus, let $L \in \mathbf{NP}$, and let \mathcal{M} be a nondeterministic Turing machine that accepts L , with $p(n)$ the polynomial that furnishes a bound on the number of steps \mathcal{M} requires to accept an input string. Without loss of generality, we assume that $p(n) \geq n$ for all n . We must show that there is a polynomial-time computable function that translates any input string u for \mathcal{M} into a CNF formula δ_u such that u is accepted by \mathcal{M} if and only if δ_u is satisfiable. For a given input u , let $t = p(|u|)$.

We know that if \mathcal{M} accepts input u , it does so in $\leq t$ steps. Therefore, in order to determine whether \mathcal{M} accepts u , we need only run it on u for at most t steps and check to see whether the final configuration is terminal. Since at each step of the computation, \mathcal{M} can move at most one square to the left or right of the square currently being scanned, it follows that after t steps, the scanned square can be at most t squares to the left or t squares to the right of its original position. Since we have chosen the polynomial $p(n)$ so that $t \geq |u|$, for our present purposes it suffices to consider $2t + 1$ squares of tape. Thus, since we are considering only t steps of the computation, we can completely exhibit all of the information on \mathcal{M} 's tape, using a t by $(2t + 1)$ array (see Fig. 3.1).

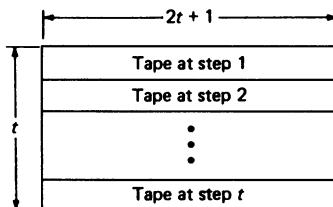


Figure 3.1

The first line of this array, corresponding to the initial tape contents, will then have the form

$$s_0^{[t+1]} u s_0^{[t-|u|]}$$

where \mathcal{M} begins in state q_1 scanning the $(t + 1)$ th symbol in this string, the s_0 immediately preceding u .

We will find it convenient, in this proof, to use the Turing machine model used in Theorem 4.2 in Chapter 6, in which acceptance of an input is by arrival in a unique accepting state q_m . We assume, therefore, that \mathcal{M} is a Turing machine of this type. Let the set of states of \mathcal{M} be $Q = \{q_1, q_2, \dots, q_m\}$ and let the set of tape symbols be $S = \{s_0, s_1, \dots, s_r\}$. It will simplify matters if we need to check only configuration number t to determine acceptance. Thus, we alter our definition of accepting computation to permit any number of repetitions of consecutive configurations; hence we may assume that our accepting computation consists of *exactly* t steps.

We will define a CNF formula δ_u that is satisfiable if and only if u is accepted by \mathcal{M} . Our set of atoms (each of length $O(t^2)$) will be

$$\mathcal{A} = \{ \rho_{h,j,k}, \sigma_{i,j,k} \mid 1 \leq h \leq m, 0 \leq i \leq r, 1 \leq j \leq 2t + 1, 1 \leq k \leq t \}.$$

We first assume that u is accepted by \mathcal{M} , so that we have an accepting computation by \mathcal{M} for u . We assume that the above t by $2t + 1$ array has been constructed correspondingly. We will construct the CNF formula δ_u so that $\delta_u^v = 1$, where v is the assignment on \mathcal{A} defined by

$$v(\rho_{h,j,k}) = \begin{cases} 1 & \text{if } \mathcal{M} \text{ is in state } q_h \text{ scanning the } j\text{th position at the } \\ & \text{ } k\text{th step of the computation} \\ 0 & \text{otherwise,} \end{cases} \quad (3.1)$$

$$v(\sigma_{i,j,k}) = \begin{cases} 1 & \text{if tape symbol } s_i \text{ is in the } j\text{th position of the } k\text{th} \\ & \text{row of the array} \\ 0 & \text{otherwise.} \end{cases}$$

In constructing δ_u , we will find the following abbreviation useful:

$$\nabla\{x_e \mid 1 \leq e \leq l\} = \bigwedge_{1 \leq e < f \leq l} (\neg x_e \vee \neg x_f) \wedge \bigvee_{1 \leq e \leq l} x_e,$$

where $\{x_e \mid 1 \leq e \leq l\}$ is a set of formulas. Thus,

$$\nabla\{x_e \mid 1 \leq e \leq l\} \quad (3.2)$$

is a formula whose value is TRUE (i.e., 1) under a given assignment if and only if *exactly* one of the formulas x_1, x_2, \dots, x_l has the value TRUE under that assignment. In the particular case that x_1, x_2, \dots, x_l are atoms, (3.2) is a CNF formula. We will need to calculate $|\nabla\{x_e \mid 1 \leq e \leq l\}|$ in this case. Formula (3.2) contains a clause consisting of two literals for each pair (e, f) with $1 \leq e < f \leq l$, followed by a single clause of l literals. Since there are $l(l - 1)/2$ such pairs (e, f) , and since in our notation, with / being used to separate clauses, each clause is of length 1 plus the number of its literals, we have

$$|\nabla\{x_e \mid 1 \leq e \leq l\}| = \left(\frac{l(l - 1)}{2} \cdot 3 + (l + 1) \right) \cdot O(t^2) = O(l^2 t^2).$$

Let us write

$$u = s_{u_1} s_{u_2} \cdots s_{u_z}, \quad \text{where } |u| = z.$$

We present a sequence of CNF formulas whose conjunction δ_u (which is then also a CNF formula) may be thought of as simulating the behavior of \mathcal{M} in accepting u . Each of these formulas has the value TRUE under the assignment v . We precede each formula with an English sentence in quotes, which may be thought of as expressing a corresponding property of the accepting computation by \mathcal{M} for u ; each such sentence is intended to make it clear that the corresponding formula is indeed true under the assignment v . In some cases the formula as written will not be in CNF; in these cases the formula written is intended to stand for a formula in CNF obtained from it by using the methods of Chapter 12, Section 3.

(1) “The initial configuration has tape contents corresponding to the first row of the array, with \mathcal{M} in state q_1 scanning the symbol s_0 immediately to the left of the first symbol of u .”

$$\bigwedge_{0 < j \leq t+1} \sigma_{0,j,1} \wedge \bigwedge_{0 < j \leq z} \sigma_{u_j,t+j+1,1} \wedge \bigwedge_{0 < j \leq t-z} \sigma_{0,t+z+j+1,1} \wedge \rho_{1,t+1,1}.$$

This expression is clearly of length $O(t^3)$.

(2) “At each step of the computation there is a unique state and a unique scanned square.”

$$\bigwedge_{1 \leq k \leq t} \nabla \{ \rho_{h,j,k} \mid 1 \leq h \leq m, 1 \leq j \leq 2t + 1 \}.$$

By the preceding remarks, the length of this expression is $O(t^5)$.

(3) “Each entry of the array contains exactly one symbol.”

$$\bigwedge_{1 \leq k \leq t} \bigwedge_{1 \leq j \leq 2t+1} \nabla \{ \sigma_{i,j,k} \mid 0 \leq i \leq r \}.$$

r is a constant, so that this expression is of length $O(t^4)$.

(4) “Each configuration in the computation, after the first, is identical to the preceding configuration, or is obtained from it by applying one of the quadruples of \mathcal{M} . ”

This formula will be the most complicated. Let the quadruples of \mathcal{M} be as follows:

$$\{ q_{i_a} s_{j_a} s_{k_a} q_{l_a} \mid a = 1, 2, \dots, \bar{a} \}, \quad (3.3a)$$

$$\{ q_{i_b} s_{j_b} R q_{l_b} \mid b = 1, 2, \dots, \bar{b} \}, \quad (3.3b)$$

$$\{ q_{i_c} s_{j_c} L q_{l_c} \mid c = 1, 2, \dots, \bar{c} \}. \quad (3.3c)$$

To make the formula easier to understand, we write it in the form

$$\bigwedge_{1 \leq k < t} \bigwedge_{1 \leq j \leq 2t+1} (\text{NOTHEAD}(j, k) \vee \text{IDENT}(j, k) \\ \vee A(j, k) \vee B(j, k) \vee C(j, k)),$$

where each of these five disjuncts will be explained below. It will turn out that each disjunct has length $O(t^2)$; hence we may conclude that the length of the entire formula will be $O(t^4)$.

We define

$$\text{NOTHEAD}(j, k) = \bigvee_{0 \leq i \leq r} (\sigma_{i,j,k} \wedge \sigma_{i,j,k+1}) \wedge \bigwedge_{1 \leq h \leq m} \neg \rho_{h,j,k}$$

so that $\text{NOTHEAD}(j, k)^v = 1$ for given j, k if and only if \mathcal{M} is not scanning the j th position at the k th step of the computation.

Next we set

$$\text{IDENT}(j, k) = \bigvee_{1 \leq h \leq m} \bigvee_{0 \leq i \leq r} (\rho_{h,j,k} \wedge \sigma_{i,j,k} \wedge \rho_{h,j,k+1} \wedge \sigma_{i,j,k+1}),$$

so that $\text{IDENT}(j, k)^v = 1$ for given j, k if and only if \mathcal{M} is scanning the j th position at both the k th and the $(k + 1)$ th steps of the computation, and both the state and the symbol are the same in both of these configurations.

Next,

$$A(j, k) = \bigvee_{1 \leq a \leq \bar{a}} (\rho_{i_a, j, k} \wedge \sigma_{j_a, j, k} \wedge \sigma_{k_a, j, k+1} \wedge \rho_{l_a, j, k+1}),$$

where $A(j, k)^v = 1$ if and only if the $(k + 1)$ th step results from the k th by one of the quadruples of (3.3a).

Similarly, we will define $B(j, k)$ so that $B(j, k)^v = 1$ if and only if the $(k + 1)$ th step results from the k th by one of the quadruples of (3.3b). For $j \neq 2t + 1$, we can define

$$B(j, k) = \bigvee_{1 \leq b \leq \bar{b}} (\rho_{i_b, j, k} \wedge \sigma_{j_b, j, k} \wedge \sigma_{j_b, j, k+1} \wedge \rho_{l_b, j+1, k+1}).$$

This definition will not work for $j = 2t + 1$ because there are no atoms $\rho_{h, 2t+2, k}$. But since the computation cannot proceed beyond the boundaries of our array, it suffices to take $B(2t + 1, k)$ to be any unsatisfiable formula, e.g., the empty clause.

Finally, we will define $C(j, k)$ so that $C(j, k)^v = 1$ if and only if the $(k + 1)$ th step results from the k th by one of the quadruples of (3.3c). For $j \neq 1$, we can define

$$C(j, k) = \bigvee_{1 \leq c \leq \bar{c}} (\rho_{i_c, j, k} \wedge \sigma_{j_c, j, k} \wedge \sigma_{j_c, j, k+1} \wedge \rho_{l_c, j-1, k+1}).$$

This definition will not work for $j = 1$ because there are no atoms $\rho_{h, 0, k}$. But since the computation cannot proceed beyond the boundaries of our array, it suffices to let $C(1, k)$ be any unsatisfiable formula, e.g., the empty clause.

(5) “The t th configuration is a terminal configuration.” Equivalently, “At the t th step, \mathcal{M} is in state q_m .”

$$\bigvee_{1 \leq j \leq 2t+1} \rho_{m, j, t}.$$

This expression is clearly of length $O(t^3)$.

Now, we take δ_u to be simply the conjunction of the CNF formulas (1) through (5) above. It is clear from what has already been said that if \mathcal{M} accepts u , then δ_u is satisfiable; in fact, $\delta_u^v = 1$.

Conversely, let v be an assignment such that $\delta_u^v = 1$. We will show that \mathcal{M} accepts u . By (3), we see that for each $1 \leq j \leq 2t + 1$, $1 \leq k \leq t$, there is a unique i such that $v(\sigma_{i,j,k}) = 1$. Hence we can uniquely reconstruct our t by $2t + 1$ array. By (2), for each row of the array there is a unique state q_h and position j in the row such that $v(\rho_{h,j,k}) = 1$. Thus, each row can be made into a configuration of \mathcal{M} so that (3.1) is satisfied. By (1), the configuration corresponding to the first row of the array is an initial configuration for \mathcal{M} with input u . By (4), for each row of the array after the first, the corresponding configuration is identical to it or results from it using one of the quadruples of \mathcal{M} . Finally, by (5) the entire sequence of configurations constitutes an accepting computation by \mathcal{M} for u . Thus, u is accepted by \mathcal{M} .

It remains to be shown that there is a polynomial-time computable function that maps each string u onto the corresponding CNF formula δ_u . Now, the CNF formulas of (2)–(5) do not depend on u , and a Turing machine can easily be constructed to write these on a tape in a number of steps proportional to the length of the expression, which, as we have seen, is $O(t^5)$, and hence polynomial in $|u|$. It remains to consider (1), which is a conjunction of atoms. Some of these atoms do not depend directly on u ; producing this part of (1) simply involves writing $O(t^3)$ symbols. The remaining atoms of (1) correspond in a one-one manner to the symbols making up u ; they can obviously be produced by a Turing machine in a number of steps proportional to $|u|$. This completes the proof. ■

Using Theorem 2.6, we have at once

Corollary 3.2. $P = NP$ if and only if $SAT \in P$.

Exercises

- Let \mathcal{M} be the Turing machine with the single tuple $q_1 \xrightarrow{B} q_2$, and let u be the string a . Give δ_u for $t = 1$.
- For any set \mathcal{A} of atoms, show that the set of all propositional DNF formulas over \mathcal{A} that are not tautologies is NP -complete.
- For any set \mathcal{A} of atoms, show that the set of all satisfiable propositional formulas over \mathcal{A} is NP -complete.
- The HALF-SAT problem is this: given a propositional CNF formula γ , determine if there is an assignment v on the atoms in $\gamma^v = 1$ and such that $\alpha^v = 1$ for exactly half of the atoms α in γ . [Hint: Show that $SAT \leq_p HALF-SAT$. Given a CNF formula γ , create a new atom α' for each atom α in γ and add clauses of the form $\{\alpha, \alpha'\}, \{\bar{\alpha}, \bar{\alpha}'\}$.]

4. Other NP-Complete Problems

The principal technique for proving a problem to be **NP**-complete is given by the following result:

Theorem 4.1. Let Q be an **NP**-complete problem, and let $Q \leq_p L$. Then L is **NP-hard**.

Proof. Let R be any language such that $R \in \mathbf{NP}$. Since Q is **NP**-complete, we have $R \leq_p Q$. By Theorem 2.5, $R \leq_p L$. Thus, L is **NP-hard**. ■

Corollary 4.2. Let Q be an **NP**-complete problem, let $L \in \mathbf{NP}$, and let $Q \leq_p L$. Then L is **NP-complete**.

Thus, once it has been shown that a problem is **NP**-complete, it can be used to show that other problems are **NP**-complete. In this way many problems have been shown to be **NP**-complete. It is this fact that constitutes the main evidence for regarding **NP**-complete problems as being intractable. Since the existence of a polynomial-time algorithm for even a single one of these problems would imply that there is a polynomial-time algorithm for every one of them, and, since it is argued that it is most unlikely that this could be the case without even one of these algorithms having been discovered, it is concluded that in all likelihood none of these problems have polynomial-time algorithms, and so they should all be regarded as intractable.

We will present a very small sample of this work, showing that a few problems are **NP**-complete. We begin with a restricted form of the satisfiability problem.

The 3-SAT problem is to determine whether a formula in CNF in which no clause contains more than three literals is satisfiable. We show that 3-SAT is **NP**-complete by showing that any CNF formula ζ can be transformed in polynomial time to a CNF formula ζ' containing at most three literals per clause such that ζ is satisfiable if and only if ζ' is satisfiable.

Theorem 4.3. 3-SAT is **NP**-complete.

Proof. Since 3-SAT is a special case of SAT, and SAT is in **NP**, it follows that 3-SAT is in **NP**. Let

$$/\alpha_1 \alpha_2 \cdots \alpha_k, \quad k \geq 4, \tag{4.1}$$

be any one of the clauses of ζ containing more than three literals. Let $\beta_1, \beta_2, \dots, \beta_{k-3}$ be atoms which do not appear in ζ . We construct ζ' by

replacing (4.1) by the conjunction

$$/\alpha_1\alpha_2\beta_1/\alpha_3\bar{\beta}_1\beta_2/\alpha_4\bar{\beta}_2\beta_3/\cdots/\alpha_{k-2}\bar{\beta}_{k-4}\beta_{k-3}/\alpha_{k-1}\alpha_k\bar{\beta}_{k-3}.$$

It is easy to see that ζ is satisfiable if and only if ζ' is satisfiable. Moreover, since the length of ζ' is bounded by a constant times the length of ζ , the transformation can be performed in linear time. ■

It is interesting that there are problems which superficially appear to be unrelated, but between which we can readily find a polynomial-time transformation. Our next example is known as the COMPLETE-SUBGRAPH problem. A *graph* G consists of a finite nonempty set of *vertices* $V = \{v_1, \dots, v_n\}$ and a finite set of *edges* E . Each edge is a pair of vertices. The *size* of the graph is simply the number of vertices it contains. A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subseteq V$, and $E' \subseteq E$. A graph $G = (V, E)$ is *complete* if there is an edge in E between every pair of distinct vertices in V .

The COMPLETE-SUBGRAPH problem is this: given a graph and a number k , does the graph have a complete subgraph of size k ?

Theorem 4.4. COMPLETE-SUBGRAPH is NP-complete.

Proof. We show informally that COMPLETE-SUBGRAPH is in NP. Let the number k and a list of the vertices and of the edges of the given graph be written on the tape of a Turing machine in any reasonable notation. The procedure begins by nondeterministically selecting a vertex and then decrementing k . By continuing this process until k has been decremented to 0, a list of k vertices is obtained. The procedure then tests [in time $O(k^2)$] whether the graph has a complete subgraph in those vertices. Since

$$k \leq n \leq \text{length of the string representing } G \text{ on the tape},$$

where G is the given graph, this shows that COMPLETE-SUBGRAPH \in NP.

To show that COMPLETE-SUBGRAPH is NP-hard, we show that SAT \leq_p COMPLETE-SUBGRAPH. Thus, we must show how to map each CNF formula γ into a pair consisting of a number k and a graph G so that γ is satisfiable if and only if G has a complete subgraph of size k . If $\gamma = / \gamma_1 / \gamma_2 \cdots / \gamma_k$ is a CNF formula, where $\gamma_1, \gamma_2, \dots, \gamma_k$ are clauses, then we take the number k to be simply the number of clauses in γ and construct the graph $G = (V, E)$, where

$$V = \{(\alpha, i) \mid \alpha \text{ is a literal in } \gamma_i\},$$

$$E = \{((\alpha, i), (\beta, j)) \mid \alpha \neq \neg \beta \text{ and } i \neq j\}.$$

Thus we have a vertex for each occurrence of each literal in γ . Edges join pairs of vertices that represent literals in different clauses provided one is not the negation of the other. This means that these literals can both be assigned the value “TRUE” at the same time. If γ is satisfiable, there is some way to assign truth values to the atoms so that γ evaluates to “TRUE.” Thus at least one literal of each clause of γ must be assigned the value “TRUE,” and in G there will be an edge connecting each pair of “true literals.” This means that the nodes of G corresponding to the “true literals” of γ form a complete subgraph of size k . Conversely, if γ contains a complete subgraph of size k , then since edges join pairs of literals in different clauses that can be true at the same time, there is a way to make each clause of γ true at the same time. Thus γ is satisfiable. Furthermore, G can clearly be obtained from γ by a polynomial-time computable function. ■

A *clique* in a given graph is a maximal complete subgraph of that graph; that is, a clique is a complete subgraph of a given graph that is *not* a subgraph of any other complete subgraph of that graph. The MAX-CLIQUE problem is to find the size of the largest clique in a given graph. Of course, in this form, MAX-CLIQUE is not a language but rather a function, and so it does not make sense in terms of our definitions to ask whether it is in NP. However, since removing a vertex and all edges containing it from a complete subgraph yields another complete subgraph, we see that any algorithm for the MAX-CLIQUE problem that could actually be implemented using reasonable resources could easily be transformed into an equally usable algorithm for the COMPLETE-SUBGRAPH problem. Hence, to the extent that NP-completeness can be regarded as implying intractability, we are entitled to conclude that MAX-CLIQUE is likewise intractable.

We next consider a closely related graph-theoretic problem, known as VERTEX-COVER. A set S is a *vertex cover* for a graph $G = (V, E)$ if $S \subseteq V$ and for every $(x, y) \in E$, either $x \in S$ or $y \in S$. The VERTEX-COVER problem is to determine for a given graph G and integer k whether G has a vertex cover of size k .

Theorem 4.5. Let $G = (V, E)$ be a graph and let

$$E' = \{(x, y) \mid x, y \in V, x \neq y, \text{ and } (x, y) \notin E\}.$$

Let us consider the graph $G' = (V, E')$ (sometimes called the *complement graph* of G). Then $S \subseteq V$ is the set of vertices of a complete subgraph of G if and only if $V - S$ is a vertex cover in G' .

Proof. Let S be the set of vertices of a complete subgraph of G . Then, by definition, for any $(x, y) \in E'$, either $x \in V - S$ or $y \in V - S$. Thus, $V - S$ is a vertex cover of G' . Conversely, if $V - S$ is a vertex cover of G' , then for any $(x, y) \in E'$, either $x \in V - S$ or $y \in V - S$. Thus no edge of G' connects two vertices in S . Thus for every $u, v \in S$, $u \neq v$, we have $(u, v) \in E$, and so S is the set of vertices of a complete subgraph of G . ■

Corollary 4.6. VERTEX-COVER is NP-complete.

The SET-COVER problem is to determine for a family of sets $\Delta = \{S_1, S_2, \dots, S_n\}$, and number k , whether there exists a subfamily Γ of Δ of size k , $\Gamma = \{S_{m_1}, S_{m_2}, \dots, S_{m_k}\}$, such that

$$\bigcup_{i \leq n} S_i = \bigcup_{j \leq k} S_{m_j}.$$

Corollary 4.7. SET-COVER is NP-complete.

Proof. Let $G = (V, E)$ be a graph with $V = \{v_1, v_2, \dots, v_n\}$. For $i = 1, 2, \dots, n$, let

$$S_i = \{(v_i, v_j) | (v_i, v_j) \in E\} \cup \{(v_j, v_i) | (v_i, v_j) \in E\}.$$

Clearly $\Gamma = \{S_{i_1}, S_{i_2}, \dots, S_{i_k}\}$ is a set cover for $\Delta = \{S_1, S_2, \dots, S_n\}$ if and only if $\{v_{i_1}, v_{i_2}, \dots, v_{i_k}\}$ is a vertex cover for G . ■

Many hundreds of NP-complete problems have been identified in quite diverse areas. We conclude this section with a few more examples. For each we indicate in brackets the nature of some known proof of NP-hardness.

1. HAMILTONIAN-CIRCUIT (HC): given a graph $G = (V, E)$ with k vertices, determine if there is an ordering v_1, \dots, v_k of the vertices in V such that $(v_i, v_{i+1}) \in E$, $1 \leq i < k$, and $(v_k, v_1) \in E$. [VERTEX-COVER \leq_p HC.]
2. 3-DIMENSIONAL-MATCHING (3DM): given a set $S \subseteq A \times B \times C$, where A, B, C are disjoint finite sets each with q elements, determine if there is a subset $M \subseteq S$ with q elements such that for any $(a, b, c), (a', b', c') \in M$, $a \neq a'$, $b \neq b'$, and $c \neq c'$. [3SAT \leq_p 3DM.]
3. PARTITION: given a set $A = \{a_1, \dots, a_n\}$ of positive integers, determine if there is a subset $S \subseteq A$ such that $\sum_{a \in S} a = \sum_{a \in A - S} a$. [3DM \leq_p PARTITION.]
4. INTEGER-PROGRAMMING (IP): given a finite set

$$X = \{((x_1^i, \dots, x_n^i), z_i) | 1 \leq i \leq m\},$$

where all x_j^i, z_i are integers, and given a tuple (c_1, \dots, c_n) of integers and an integer b , determine if there is a tuple (y_1, \dots, y_n) of integers such that $(x_1^i, \dots, x_n^i) \cdot (y_1, \dots, y_n) \leq z_i$, $1 \leq i \leq n$, and $(c_1, \dots, c_n) \cdot (y_1, \dots, y_n) \geq b$. (The dot product of any two n -tuples is defined $(x_1, \dots, x_n) \cdot (y_1, \dots, y_n) = \sum_{i=1}^n x_i \cdot y_i$.) [$3\text{SAT} \leq_p \text{IP}$.]

5. QUADRATIC-DIOPHANTINE-EQUATIONS (QDE): given positive integers a, b, c , determine if there are positive integers x, y such that $ax^2 + by = c$. [$3\text{SAT} \leq_p \text{QDE}$.]
6. STRAIGHTLINE-PROGRAM-INEQUIVALENCE (SPI): given a set of variables $\{X_1, \dots, X_n\}$, two programs \mathcal{P}, \mathcal{Q} each being a sequence of assignments of the form

$$V \leftarrow \text{IF } W = X \text{ THEN } Y \text{ ELSE } Z,$$

where $V, W, X, Y, Z \in \{X_1, \dots, X_n\}$, and given a set of values $\{v_1, \dots, v_m\}$, determine if there is an initial state

$$\{X_1 = v_{i_1}, \dots, X_n = v_{i_n}\},$$

where each $v_{i_j} \in \{v_1, \dots, v_m\}$, such that \mathcal{P}, \mathcal{Q} end with a different value for some variable. [$3\text{SAT} \leq_p \text{SPI}$.]

Exercises

1. The CHROMATIC-NUMBER problem is to determine for a given graph $G = (V, E)$ and integer k whether there is a function f from V to $\{1, 2, \dots, k\}$ such that if $(x, y) \in E$, then $f(x) \neq f(y)$. (Intuitively, this problem amounts to determining whether or not it is possible to “color” the vertices of G using k colors in such a way that no two adjacent vertices are colored the same.) Show that CHROMATIC-NUMBER is NP-complete. [Hint: Show $3\text{-SAT} \leq_p \text{CHROMATIC-NUMBER}$.] [Further hint: Assume $\gamma = / \gamma_1 / \gamma_2 \cdots / \gamma_m$ is a CNF formula such that no γ_i contains more than three literals. Assume there are n atoms $\alpha_1, \alpha_2, \dots, \alpha_n$ that appear either negated or unnegated in γ . Construct a graph G with $3n + m$ vertices such that G is $n + 1$ colorable if and only if γ is satisfiable.]
2. The 2-COLORABILITY problem is to determine whether a given graph can be colored using only two colors. Show that 2-COLORABILITY is in P.
3. The 2-SAT problem is to determine whether a CNF formula in which no clause contains more than two literals is satisfiable. It is known that $2\text{-SAT} \in \text{P}$. Show why a technique like the one used to show

3-SAT is **NP**-complete does not work for 2-SAT. Show that 2-SAT is in **P**.

4. The EXACT-COVER problem is to determine for a finite family of sets $\Delta = \{S_1, S_2, \dots, S_n\}$ whether there exists a set cover Γ of Δ such that the elements of Γ are pairwise disjoint. Show that EXACT-COVER is **NP**-complete. [Hint: Show that

$$\text{CHROMATIC-NUMBER} \leq_p \text{EXACT-COVER}.$$

5. The SUBGRAPH-ISOMORPHISM (SI) problem is, given graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, to determine if there is a one-one function f from V_1 to V_2 such that $(v_i, v_j) \in E_1$ if and only if $(f(v_i), f(v_j)) \in E_2$. Show that SUBGRAPH-ISOMORPHISM is **NP**-complete. [Hint: Show COMPLETE-SUBGRAPH \leq_p SI.]
6. The LONGEST-COMMON-SUBSEQUENCE (LCS) problem is, given an alphabet A , a set $\{w_1, \dots, w_n\}$ of strings on A , and a positive integer k , to determine if there is a string $y \in A^*$ with $|y| \geq k$ such that, for $1 \leq i \leq n$, $w_i = x_0 y_1 x_1 y_2 x_2 \cdots y_l x_l$ and $y = y_1, \dots, y_l$ for some $x_0, \dots, x_l, y_1, \dots, y_l \in A^*$. Show that LCS is **NP**-complete. [Hint: Show VERTEX-COVER \leq_p LCS. Let $G = (V, E)$ be a graph, where $V = \{v_1, \dots, v_n\}$ and $E = \{(v_{i_1}, v_{j_1}), \dots, (v_{i_m}, v_{j_m})\}$, where $i_l \leq j_l$, $1 \leq l \leq m$. For each edge (v_{i_l}, v_{j_l}) , create the string

$$w_l = v_1 \cdots v_{i_{l-1}} v_{i_l+1} \cdots v_n v_1 \cdots v_{j_{l-1}} v_{j_l+1} \cdots v_n,$$

and also create the string $u = v_1 \cdots v_n$. Show that G has a vertex cover of size k if and only if $\{w_1, \dots, w_m, u\}$ has a common subsequence of size $n - k$.]

7. The TRAVELING-VENDOR (TV) problem is, given a set $C = \{c_1, \dots, c_n\}$ of cities, a positive integer distance $d(c_i, c_j)$ for each pair of cities, and a positive integer b , to determine if there is a Hamiltonian circuit $\langle c_{i_1}, \dots, c_{i_m} \rangle$ such that

$$\sum_{j=1}^{m-1} d(c_{i_j}, c_{i_{j+1}}) + d(c_{i_m}, c_{i_1}) \leq b.$$

Show that TV is **NP**-complete. [Hint: show HC \leq_p TV.]

8. The SUBSET-SUM problem is, given a set $\{a_1, \dots, a_n\}$ of positive integers and positive integer b , to determine if there is a subset $\{b_1, \dots, b_m\} \subseteq \{a_1, \dots, a_n\}$ such that $\sum_{i=1}^m b_i = b$. Show that SUBSET-SUM is **NP**-complete. [Hint: Show PARTITION \leq_p SUBSET-SUM.]

9. The KNAPSACK problem is, given a set $S = \{(s_1, v_1), \dots, (s_n, v_n)\}$ of pairs of positive integers, where s_i is a size and v_i is a value, $1 \leq i \leq n$, and given positive integers b, k , to determine if there is a subset $A \subseteq S$ such that

$$\sum_{(s, v) \in A} s \leq b \quad \text{and} \quad \sum_{(s, v) \in A} v \geq k.$$

Show that KNAPSACK is **NP**-complete. [Hint: Show PARTITION \leq_p KNAPSACK.]

10. The MULTIPROCESSOR-SCHEDULING (MS) problem is, given a set $T = \{t_1, \dots, t_n\}$ of positive integers (task times) and positive integers m (number of processors) and d (deadline), to determine if there is a partition of T into disjoint sets T_1, \dots, T_m such that for $1 \leq i \leq m$, $\sum_{t \in T_i} t \leq d$. Show that MS is **NP**-complete. [Hint: Show PARTITION \leq_p MS.]
11. The RECORD-ALLOCATION (RA) problem is, given a set $L = \{l_1, \dots, l_n\}$ of positive integers (record lengths) and positive integers t (track length) and k (number of tracks), to determine if there is a partition of L into disjoint sets L_1, \dots, L_k such that for $1 \leq i \leq k$, $\sum_{l \in L_i} l \leq t$. Show that RA is **NP**-complete. [Hint: Show PARTITION \leq_p RA.]
12. The TASK-SEQUENCING (TS) problem is, given a set

$$\{(t_1, d_1, p_1), \dots, (t_n, d_n, p_n)\}$$

of triples of positive integers (where for $1 \leq i \leq n$, t_i is the amount of time necessary to complete task i , d_i is the deadline for task i , and p_i is the penalty for failing to complete task i by its deadline) and given a positive integer b , to determine if there is a sequence $\langle i_1, \dots, i_n \rangle$ of tasks such that $\sum_{i \in L} p_i \leq b$, where $L \subseteq \{1, \dots, n\}$ is the set of late tasks, i.e., those i_j with $\sum_{r=1}^j t_{i_r} > d_{i_j}$. Show that TS is **NP**-complete. [Hint: Show PARTITION \leq_p TS.]

Part 5

Semantics

16

Approximation Orderings

1. Programming Language Semantics

In Part 1 of this book we studied various classes of functions, principally the class of partially computable functions. In Part 5 we also investigate classes of functions but from a different perspective. One of the key results from Part 1 is that the partially computable functions can be defined by way of any number of substantially different formalisms. Once the equivalence of \mathcal{S} programs, Turing machines, etc., has been demonstrated, it becomes clear that the definition of partially computable functions in terms of \mathcal{S} programs is an artifact of our particular exposition, and in results like Theorem 2.1 of Chapter 4 concerning the HALT predicate, the role of \mathcal{S} programs recedes to the formal background. In direct proportion to the accumulation of equivalent formal systems, the class of partially computable numeric functions takes on an independent, absolute status, and the status of each particular formal system declines. It is fair to say that computability is about a certain class of functions, however they are defined.

On the more practical side of computer science, however, the formal description of functions has blossomed into the elaborate field of programming languages, where we find thousands of formal systems far richer and more complex than any we describe in this book. The differences between

the two fields are entirely appropriate. In the *theory* of computation, where we are interested in the abstract mathematical properties of functions and classes of functions, it is appropriate to eliminate all but the most essential components of our formal systems. In the areas of computer science that support the *practice* of solving problems, it is appropriate to elaborate a wide range of programming languages to support the needs of various problem domains, programming styles, and philosophies of language design.

When the various programming languages are important in their own right, the business of associating a function to each program is more than a means to the end of defining a class of functions. It becomes the subject of *programming language semantics*.

The issue came up already in Chapter 2 when we carefully defined a semantics for \mathcal{S} programs, associating a partial function $\psi_{\mathcal{P}}$ with each program \mathcal{P} . In the course of defining the semantics of \mathcal{S} programs we defined the notion of a computation, which characterizes a mechanical process of deriving a numerical output value from an input value. In essence we have defined an abstract machine that stores the values of the X , Y , and Z variables and updates those values by performing various operations as specified by a program. A computation describes the sequence of states assumed by the machine in the process of deriving an output (if there is one), and the meaning of a program is characterized by all of the computations it performs, one for each possible input. This style of defining the meaning of programs is called *operational semantics* because it depends on the operation of some kind of machine.

It is clear that programs and computations are very different sorts of objects. Computations are dynamic in nature; that is, they describe a process that evolves over time. Without the benefit of a semantics to give them meaning, programs are simply static sequences of syntactic symbols. Now, the goal of creating a semantics is to associate a function with a program, and at least from the perspective of set theory, a function is a static entity: it is simply a set of ordered pairs. We could argue, then, that it is a diversion to interpose the conceptual complication of computations between a program and its function. It would be more straightforward to define a function, by purely “mathematical” means, directly from the syntactic structure of the program. Of course, the concept of “purely ‘mathematical’ means” is not precisely defined. In the present context it implies, at the very least, an absence of operational detail. Its connotation will become clearer as we proceed. This alternative to operational semantics is called *denotational semantics*.

The denotational approach might be preferable for its conceptual simplicity, but we do not mean to imply that it is “better” than, or a

replacement for, an operational semantics. In a practical setting the two are complementary. A denotational semantics can provide a succinct description of the meaning of a programming language, abstracted from the level of pragmatic details, and an operational semantics approaches more closely an actual implementation. In the theoretical area of computability, the operational style is crucial, preceding the denotational style both historically and conceptually. It is the mechanical nature of an operational semantics that gives sense to the term *computable function*. If computability theory is more about the class of partially computable functions than the particular formal systems for defining them, it is just as much about the concept of mechanical computation which is embodied in the operational semantics of \mathcal{S} programs, Turing machines, Pascal, LISP, etc.

The exposition given here of semantics, both operational and denotational, has two goals.

1. It should broaden and deepen the understanding of computable functions and computation.
2. It is an introduction to some of the ideas found in the theoretical study of programming languages.

There are two ways in which we will extend the theory of computable functions covered in Part 1. One is to expand the class of data objects that are directly covered by the theory. We have accounted for computable functions on the natural numbers and computable functions on strings over arbitrary finite alphabets, but the typical programming language offers a much greater variety of data types like lists, arrays, and in many cases, user-defined data types. Now, natural numbers and strings are both perfectly appropriate data types for a theory of computable functions because of their capacity for encoding more complex structures. We showed in Chapter 3, for example, how finite lists of numbers can be encoded as a single Gödel number. So a theorem like the universality theorem in Chapter 4 implicitly tells us that there is a partially computable universal function for partially computable functions on lists of numbers. However, by explicitly admitting a richer assortment of data types, we can bring the theory closer to the actual practice of computation.

In every model of computation we covered in Part 1, a function computed by some program or machine was considered to be defined for a given input just when there was a finite computation for that input. Indeed, in Chapter 2 we defined a computation as a *finite* sequence of instantaneous descriptions. When a given input leads to an infinite sequence of instantaneous descriptions, we did not consider that sequence a computation, and so we did not consider the program to be doing any useful work

in this case. However, there are programs whose sole reason for being is the work they accomplish while they are running, rather than the output they produce at the end of a computation. For example, an operating system, the program that organizes the functioning of a computer, produces little useful output when it is terminated, and, in fact, it is designed to be able to run without ever terminating. The termination of an operating system might just as well indicate a failure rather than the successful completion of a computation.

Our second extension, then, is to try to account for the work done in the course of a computation. Our perspective here is that instantaneous descriptions represent partial results that *approximate* the overall result of a computation. We will admit the possibility of infinite computations as well. In a sense, the result of an infinite computation is the *computation itself*, and each instantaneous description is a piece of the result, approximating the whole. Our data structures, therefore will come equipped with an ordering, an *approximation ordering*, which formalizes the notion of different partial results being more or less complete realizations of the total result.

We will focus here on the equational style of function definition. In particular, we will work with equations like

$$F(X) = H(G(X), X). \quad (1.1)$$

There are two kinds of variables in Eq. (1.1). X is intended to denote individuals, say, natural numbers, and F , G , and H denote functions. There is another important difference in our interpretation of these variables. In the equation $2x = x + x$, with x ranging over the natural numbers, equality holds for all values of x , but the equation $x^2 - 4 = 0$ calls for one or more particular values of x which make equality hold. In (1.1) X has the first interpretation, and F , G , and H have the second. That is, we are looking for functions f , g , and h that make $f(X) = h(g(X), X)$ true for all values of X , in which case the assignment of f , g , and h to variables F , G , and H constitutes a solution to (1.1). In equational programming we define functions by writing sets of equations to be satisfied.

Normally equations are understood to be symmetric in their left and right sides, but the two sides of (1.1) have an important distinction. If F is assigned a function there might be a number of assignments for G and H that solve (1.1), but assigning functions to G and H induces a unique value for F . In this sense we can interpret the right side as a function (sometimes called a *higher order function*) which takes any pair of functions g , h assigned to G and H and yields a unique function for F . We will make

essential use of such higher order functions in the denotational semantics of recursion equations.

We need to consider the issue of partial functions. In the simple equation

$$F(X) = G(X), \quad (1.2)$$

if F and G are interpreted as total functions f and g , then there is no ambiguity in the requirement that $f(x) = g(x)$ for all values of x . Suppose, though, that f and g are defined

$$f(x) = g(x) = \begin{cases} 0 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0. \end{cases}$$

What should be the meaning of $f(0) = g(0)$? Previously we have interpreted equality to mean

1. either both sides are defined or both sides are undefined, and
2. when both sides are defined they have the same value.

An alternative, which we will now adopt, is to extend the universe of objects with a new element that represents the property of being undefined. For instance, we extend N to $N_{\perp} = N \cup \{\perp_N\}$, where \perp_N (pronounced “bottom”) is different from all natural numbers. Now we can define

$$f_{\perp}(x) = g_{\perp}(x) = \begin{cases} 0 & \text{if } x > 0 \\ \perp_N & \text{if } x = 0. \end{cases}$$

f and g are not total functions, but f_{\perp} and g_{\perp} are, and $f_{\perp}(x) = g_{\perp}(x)$ for all $x \in N$.

There are two distinct kinds of elements in N_{\perp} : numbers and \perp_N . We can compare them by saying that numbers are completely defined elements and \perp_N is the unique completely undefined element. This is a simple example of our notion of approximation. In a sense \perp_N is an approximation, a very weak one, to any natural number n . The idea is clearer, perhaps, in a richer set like N_{\perp}^3 , where we can say that $(\perp_N, \perp_N, \perp_N)$ and $(3, \perp_N, 5)$ both approximate $(3, 7, 5)$, and that $(3, \perp_N, 5)$ is a better approximation than $(\perp_N, \perp_N, \perp_N)$. Thus, we can think of a sequence like $(\perp_N, \perp_N, \perp_N), (3, \perp_N, 5), (3, 7, 5)$ as a computation, where $(\perp_N, \perp_N, \perp_N)$ and $(3, \perp_N, 5)$ are partial results leading to the final value $(3, 7, 5)$.

In this chapter we investigate the mathematical aspects of approximation orderings and functions defined on them. In the next chapter we apply these ideas to the semantics of recursion equations.

2. Partial Orders

For a set D , a *binary relation on D* is any subset of $D \times D$. If R is a binary relation on some set, we generally write $a R b$ to mean $(a, b) \in R$. If R is a binary relation on D and $E \subseteq D$, then the binary relation $\{(a, b) \in E \times E \mid a R b\}$ on E is the *restriction of R to E* .

Definition. Let D be a set and \sqsubseteq a binary relation on D . \sqsubseteq is a *partial ordering of D* if it has the properties of

1. reflexivity: $a \sqsubseteq a$ for all $a \in D$;
2. antisymmetry: $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$ for all $a, b \in D$;
3. transitivity: $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$ for all $a, b, c \in D$.

If \sqsubseteq is a partial ordering of D , then the pair (D, \sqsubseteq) is a *partially ordered set*, or simply a *partial order*. We will sometimes write $a \sqsubset b$ to mean $a \sqsubseteq b$ and $a \neq b$.

It is easy to find examples of partial orders. (N, \leq) , where \leq is the usual ordering of N , is a partial order. $(N_\perp, \sqsubseteq_{N_\perp})$, where \sqsubseteq_{N_\perp} is defined

$$m \sqsubseteq_{N_\perp} n \quad \text{if and only if} \quad m = \perp_N \text{ or } m = n,$$

is also a partial order. Note that while $5 \leq 7$, for example, it is not true that $5 \sqsubseteq_{N_\perp} 7$. If D is a set, the *power set* of D , denoted $\mathcal{P}(D)$, is the set of all subsets of D . For any set D , it is easy to see that $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$ is a partial order, where $\subseteq_{\mathcal{P}(D)}$ is the subset relation on the sets in $\mathcal{P}(D)$. Also, $(D, =_D)$ is a partial order for any set D , where $=_D$ is the equality relation on the elements of D . Although \leq , \sqsubseteq_{N_\perp} , $\subseteq_{\mathcal{P}(N)}$, and $=_N$ are all partial orderings, they are quite different in structure.

Definition. A partial ordering \sqsubseteq of a set D is a *linear ordering of D* if for every $a, b \in D$, either $a \sqsubseteq b$ or $b \sqsubseteq a$. (D, \sqsubseteq) is a *linearly ordered set*, or simply a *linear order*.

So, for example, (N, \leq) is a linear order, but $(N_\perp, \sqsubseteq_{N_\perp})$, $(N, =_N)$ and $(\mathcal{P}(N), \subseteq_{\mathcal{P}(N)})$ are not.

We will often find it useful to create new partial orders from given partial orders.

Definition. Let $(D_1, \sqsubseteq_{D_1}), \dots, (D_n, \sqsubseteq_{D_n})$ be partial orders. Then $\sqsubseteq_{D_1 \times \dots \times D_n}$, the *Cartesian product ordering* on $D_1 \times \dots \times D_n$ determined

by $\sqsubseteq_{D_1}, \dots, \sqsubseteq_{D_n}$, is defined

$(d_1, \dots, d_n) \sqsubseteq_{D_1 \times \dots \times D_n} (e_1, \dots, e_n)$ if and only if $d_i \sqsubseteq_{D_i} e_i$ for all $1 \leq i \leq n$.

Theorem 2.1. If $(D_1, \sqsubseteq_{D_1}), \dots, (D_n, \sqsubseteq_{D_n})$ are partial orders, then $(D_1 \times \dots \times D_n, \sqsubseteq_{D_1 \times \dots \times D_n})$ is a partial order.

Proof. We will write \sqsubseteq for $\sqsubseteq_{D_1 \times \dots \times D_n}$ and \sqsubseteq_i for \sqsubseteq_{D_i} , $1 \leq i \leq n$. We need to show that \sqsubseteq is reflexive, symmetric, and transitive. For any $(d_1, \dots, d_n) \in D_1 \times \dots \times D_n$, we have $d_i \sqsubseteq_i d_i$ by the reflexivity of \sqsubseteq_i , $1 \leq i \leq n$, so $(d_1, \dots, d_n) \sqsubseteq (d_1, \dots, d_n)$. If $(d_1, \dots, d_n) \sqsubseteq (e_1, \dots, e_n)$ and $(e_1, \dots, e_n) \sqsubseteq (d_1, \dots, d_n)$, then, for $1 \leq i \leq n$, $d_i \sqsubseteq_i e_i$ and $e_i \sqsubseteq_i d_i$, which implies $d_i = e_i$ by the antisymmetry of \sqsubseteq_i , so we have $(d_1, \dots, d_n) = (e_1, \dots, e_n)$. Finally, if $(d_1, \dots, d_n) \sqsubseteq (e_1, \dots, e_n)$ and $(e_1, \dots, e_n) \sqsubseteq (f_1, \dots, f_n)$, then, for $1 \leq i \leq n$, $d_i \sqsubseteq_i e_i$ and $e_i \sqsubseteq_i f_i$, which implies $d_i \sqsubseteq_i f_i$ by the transitivity of \sqsubseteq_i , so we have $(d_1, \dots, d_n) \sqsubseteq (f_1, \dots, f_n)$. ■

For example, (N_\perp^n, \sqsubseteq) is a partial order, where \sqsubseteq is the Cartesian product ordering on N_\perp^n determined by $\sqsubseteq_{N_\perp}, \dots, \sqsubseteq_{N_\perp}$.

Definition. Let D and E be sets. A function whose domain is D and whose range is a subset of E is a function *from D into E*. The set of all functions from D into E is denoted $D \rightarrow E$. If \sqsubseteq_E is a partial ordering of E , then $\sqsubseteq_{D \rightarrow E}$, the *function space ordering* on $D \rightarrow E$ determined by \sqsubseteq_E , is defined

$$f \sqsubseteq_{D \rightarrow E} g \quad \text{if and only if} \quad f(d) \sqsubseteq_E g(d) \text{ for all } d \in D.$$

We sometimes write $f: D \rightarrow E$ to indicate that $f \in D \rightarrow E$.

Theorem 2.2. If D is a set and (E, \sqsubseteq_E) is a partial order, then $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$ is a partial order.

Proof. We will write \sqsubseteq for $\sqsubseteq_{D \rightarrow E}$. Let f, g, h be arbitrary functions in $D \rightarrow E$. For any $d \in D$, $f(d) \sqsubseteq_E f(d)$ by the reflexivity of \sqsubseteq_E , so $f \sqsubseteq f$. If $f \sqsubseteq g$ and $g \sqsubseteq f$, then for any $d \in D$, $f(d) \sqsubseteq_E g(d)$ and $g(d) \sqsubseteq_E f(d)$, which implies by the antisymmetry of \sqsubseteq_E that $f(d) = g(d)$ for all $d \in D$, i.e., $f = g$. If $f \sqsubseteq g$ and $g \sqsubseteq h$, then for any $d \in D$, $f(d) \sqsubseteq_E g(d)$ and $g(d) \sqsubseteq_E h(d)$, which implies by the transitivity of \sqsubseteq_E that $f(d) \sqsubseteq_E h(d)$ for all $d \in D$, i.e., $f \sqsubseteq h$. ■

For example, $(N_\perp^n \rightarrow N_\perp, \sqsubseteq)$ is a partial order, where \sqsubseteq is the function space ordering on $N_\perp^n \rightarrow N_\perp$ determined by \sqsubseteq_{N_\perp} .

Definition. Let A be a set and let \mathcal{D} be a function with domain A such that $\mathcal{D}(a)$ is a partial order for all $a \in A$. We will write (D_a, \sqsubseteq_a) for $\mathcal{D}(a)$. A \mathcal{D} -choice function¹ is a function f with domain A such that $f(a) \in D_a$ for all $a \in A$. $\text{ch}(\mathcal{D})$ is the set of all \mathcal{D} -choice functions. The \mathcal{D} -choice function ordering $\sqsubseteq_{\text{ch}(\mathcal{D})}$ is defined

$$f \sqsubseteq_{\text{ch}(\mathcal{D})} g \quad \text{if and only if} \quad f(a) \sqsubseteq_a g(a) \quad \text{for all } a \in A.$$

For example, let $A = \{1, 2\}$, let $\mathcal{D}(i) = (N^i \rightarrow N_1, \sqsubseteq_{N^i \rightarrow N_1})$ for $i = 1, 2$, and let $f(i) = u_i^i$ for $i = 1, 2$. That is, $f(1)$ is the unary projection function, and $f(2)$ is a binary projection function. (Recall that u_1^1 and u_1^2 are defined in Chapter 3.) Then $f \in \text{ch}(\mathcal{D})$.

Theorem 2.3. Let A be a set and $\mathcal{D}(a)$ a partial order for all $a \in A$. Then $(\text{ch}(\mathcal{D}), \sqsubseteq_{\text{ch}(\mathcal{D})})$ is a partial order.

Proof. The proof is identical to the proof of Theorem 2.2, except that instead of a single ordering \sqsubseteq_E , we have a different ordering \sqsubseteq_a for each $a \in A$. ■

Exercises

1. Show that $(N, |)$ is a partial order, where $m | n$ is the predicate “ m is a divisor of n ” defined in Chapter 3. [Note that $0 | 0$ is true.] Is it linear?
2. Let (D, \sqsubseteq_D) be a partial order, and let

$$\exists_D = \{(x, y) \in D \times D \mid y \sqsubseteq_D x\}.$$

Show that (D, \exists_D) is a partial order.

3. Let (D, \sqsubseteq_D) be a partial order, let $E \subseteq D$, and let \sqsubseteq_E be the restriction of \sqsubseteq_D to E .
 - (a) Show that (E, \sqsubseteq_E) is a partial order.
 - (b) Show that if (D, \sqsubseteq_D) is a linear order, then (E, \sqsubseteq_E) is a linear order.
4. For which set(s) A is (A, \emptyset) a partial order?

¹ When A is infinite, proving the existence of \mathcal{D} -choice functions generally requires an axiom from set theory known as the axiom of choice. However, our treatment of denotational semantics will require only \mathcal{D} -choice functions with a finite domain, so we need not be concerned with this issue. The interested reader should consult any introductory text on set theory, e.g., those mentioned in “Suggestions for Further Reading.”

5. Let (D, \sqsubseteq_D) be a partial order, and let $d_1, \dots, d_n \in D$ be such that $d_1 \sqsubseteq_D d_2 \sqsubseteq_D \dots \sqsubseteq_D d_n \sqsubseteq_D d_1$. Show that $d_1 = d_2 = \dots = d_n$.
6. (a) Show that there are three distinct partial orderings of $\{0, 1\}$.
 (b) Show that there are nineteen distinct partial orderings of $\{0, 1, 2\}$.
7. Let $D = \{a, b, c\}$, $\sqsubseteq_D = \{(a, a), (b, b), (c, c), (a, b), (a, c)\}$, $E = \{d, e\}$, and $\sqsubseteq_E = \{(d, d), (e, e), (d, e)\}$.
 (a) What is $(D \times E, \sqsubseteq_{D \times E})$?
 (b) What is $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$?
8. Let (D, \sqsubseteq_D) be a partial order, and let

$$\sqsubset_D = \{(x, y) \in D \times D \mid x \sqsubseteq_D y \text{ and } x \neq y\}.$$

Show that \sqsubset_D is transitive and satisfies the property of *asymmetry*, namely, that $x \sqsubset_D y$ implies $y \not\sqsubset_D x$, for all $x, y \in D$. Is \sqsubset_D reflexive?

9. Let D, E be finite sets with m, n elements, respectively.
 (a) Show by induction on m that $D \times E$ has $m \cdot n$ elements.
 (b) Show by induction on m that $D \rightarrow E$ has n^m elements.
10. Give linear orders (D, \sqsubseteq_D) and (E, \sqsubseteq_E) such that $(D \times E, \sqsubseteq_{D \times E})$ and $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$ are not linear orders.
11. Give a linear order (D, \sqsubseteq_D) with $D \neq \emptyset$ such that $(N \times D, \sqsubseteq_{N \times D})$ and $(N \rightarrow D, \sqsubseteq_{N \rightarrow D})$ are linear orders (where $\sqsubseteq_{N \times D}$ is determined by \leq , the usual ordering on N , and \sqsubseteq_D). Is $(D \times N, \sqsubseteq_{D \times N})$ a linear order? $(D \rightarrow N, \sqsubseteq_{D \rightarrow N})$?
12. Give distinct functions $f, g \in N_\perp^n \rightarrow N_\perp$ such that $f \sqsubseteq_{N_\perp^n \rightarrow N_\perp} g$.
13. Let $\mathcal{D}(i) = N_\perp^i \rightarrow N_\perp$ for all $i \in N$, $i \neq 0$. Give distinct functions $f, g \in \text{ch}(\mathcal{D})$ such that $f \sqsubseteq_{\text{ch}(\mathcal{D})} g$.

3. Complete Partial Orders

We will be particularly interested in partial orders that are rich in certain kinds of elements.

Definition. Let (D, \sqsubseteq_D) be a partial order, and let $E \subseteq D$. An element $e_0 \in E$ is the *least element* of E with respect to \sqsubseteq_D if $e_0 \sqsubseteq_D e$ for all $e \in E$, and it is the *greatest element* of E with respect to \sqsubseteq_D if $e \sqsubseteq_D e_0$ for all $e \in E$.

If (D, \sqsubseteq_D) is a partial order and $E \subseteq D$, then E can have at most one least element: if e, e' are least elements of E , then $e \sqsubseteq_D e'$ and $e' \sqsubseteq_D e$, so by antisymmetry $e = e'$. Similarly, E can have at most one greatest element. Therefore, we are justified in speaking about *the* least element of E and *the* greatest element of E .

Definition. Let (D, \sqsubseteq_D) be a partial order, and let $E \subseteq D$. An element $d \in D$ is a *lower bound* of E in (D, \sqsubseteq_D) if $d \sqsubseteq_D e$ for all $e \in E$, and it is an *upper bound* of E in (D, \sqsubseteq_D) if $e \sqsubseteq_D d$ for all $e \in E$. Moreover, d is the *least upper bound* of E in (D, \sqsubseteq_D) if it is the least element with respect to \sqsubseteq_D of the set of all upper bounds of E in (D, \sqsubseteq_D) , and it is the *greatest lower bound* of E in (D, \sqsubseteq_D) if it is the greatest element with respect to \sqsubseteq_D of the set of all lower bounds of E in (D, \sqsubseteq_D) . If the least upper bound of E in (D, \sqsubseteq_D) exists, it is denoted $\sqcup_{(D, \sqsubseteq_D)} E$. If the greatest lower bound of E in (D, \sqsubseteq_D) exists, it is denoted $\sqcap_{(D, \sqsubseteq_D)} E$.

Suppose (D, \sqsubseteq_D) is a partial order and $E \subseteq D$. Since the set of upper bounds of E in (D, \sqsubseteq_D) can have at most one least element, it follows that E can have at most one least upper bound in (D, \sqsubseteq_D) . Similarly, E can have at most one greatest lower bound in (D, \sqsubseteq_D) . Note that $\sqcup_{(D, \sqsubseteq_D)} E$, if it exists, is not necessarily an element of E , though if it is then it is the greatest element of E . A similar observation holds for $\sqcap_{(D, \sqsubseteq_D)} E$.

In our work on semantics we are interested primarily in least upper bounds. We will generally drop the subscript and write $\sqcup E$ when it is apparent to which partial order we are referring. Occasionally we will write $\sqcup_D E$.

Partial orders can differ greatly in the existence of upper and lower bounds of their various subsets. Let $A = \{0, 1, 2\}$ and let \sqsubseteq_A be the usual ordering on $\{0, 1, 2\}$. Then every subset of A has one or more upper bounds and one least upper bound. For example, 1 and 2 are both upper bounds of $\{0, 1\}$, and 1 is the least upper bound. Note that $2 = \sqcap \emptyset$ and $0 = \sqcup \emptyset$. However, consider (N, \leq) . Every finite subset of N has a greatest element, and every nonempty subset of N has a finite set of lower bounds, so every nonempty subset of N has a greatest lower bound. Also, every nonempty subset of N has a least element, and every finite subset of N has a nonempty set of upper bounds, so every finite subset of N has a least upper bound. However, an infinite subset of N has no upper bounds. Note that $\sqcap_N \emptyset$ does not exist. (Why?)

A subset of a partial order can fail to have a least upper bound for one of two reasons. Either the set of upper bounds is empty, as in the case of an infinite subset of N , or it is nonempty but has no least element. For example, let (\mathbf{Q}, \leq_Q) be the ordered set of the rational numbers. Then

$A = \{q \in \mathbf{Q} \mid q^2 < 2\}$ has plenty of upper bounds, but it has no *rational* least upper bound. On the other hand, A has a least upper bound in $(\mathbf{R}, \leq_{\mathbf{R}})$, the ordered set of the real numbers. In fact, $\sqcup_{\mathbf{R}} A = \sqrt{2}$. For a simpler example, consider (D, \sqsubseteq_D) , where $D = \{a, b, c, d\}$ and

$$\sqsubseteq_D = \{(a, a), (b, b), (c, c), (d, d), (a, c), (a, d), (b, c), (b, d)\}.$$

Here, c and d are both upper bounds of $\{a, b\}$, but $\{c, d\}$ has no least element.

If we have a sequence d_0, d_1, d_2, \dots that represents a finite or infinite computation, where d_0, d_1, d_2, \dots are elements in some partial order (D, \sqsubseteq_D) , then we want (D, \sqsubseteq_D) to contain some element d which represents the result of that computation. The following definition formalizes this idea.

Definition. Let (D, \sqsubseteq_D) be a partial order. A *chain* in (D, \sqsubseteq_D) is a nonempty set $C \subseteq D$ such that $c \sqsubseteq_D c'$ or $c' \sqsubseteq_D c$ for every $c, c' \in C$. (D, \sqsubseteq_D) is a *complete partial order*, or *cpo*, if

1. D has a least element with respect to \sqsubseteq_D , and
2. $\sqcup_{(D, \sqsubseteq_D)} C$ exists for every chain C in (D, \sqsubseteq_D) .

The least element in a partial order (D, \sqsubseteq_D) is generally written \perp_D or \perp , and called the *bottom element* of (D, \sqsubseteq_D) , or simply *bottom* of (D, \sqsubseteq_D) . Note that if (D, \sqsubseteq_D) is a partial order, C is a chain in (D, \sqsubseteq_D) , and \sqsubseteq_C is the restriction of \sqsubseteq_D to C , then (C, \sqsubseteq_C) is a linear order.

Every nonempty subset of N is a chain in (N, \leq) , since (N, \leq) is a linear order, and, as we showed previously, no infinite subset of N has a least upper bound in (N, \leq) , so (N, \leq) is not a cpo. However, any set can be turned into a cpo, in the same way that we turned N into $(N_{\perp}, \sqsubseteq_{N_{\perp}})$. Let D be a set, let \perp_D be some new object not in D , and let $D_{\perp} = D \cup \{\perp_D\}$. Then $\sqsubseteq_{D_{\perp}}$, defined

$$d \sqsubseteq_{D_{\perp}} e \quad \text{if and only if} \quad d = \perp_D \text{ or } d = e,$$

is the *flat partial ordering* of D_{\perp} . Every chain in $(D_{\perp}, \sqsubseteq_{D_{\perp}})$ is either $\{\perp_D\}$, or $\{d\}$ for some $d \in D$, or $\{\perp_D, d\}$ for some $d \in D$, so every chain in $(D_{\perp}, \sqsubseteq_{D_{\perp}})$ has a least upper bound, and therefore $(D_{\perp}, \sqsubseteq_{D_{\perp}})$ is a cpo. We call $(D_{\perp}, \sqsubseteq_{D_{\perp}})$ the *flat cpo* on D . For example, $(N_{\perp}, \sqsubseteq_{N_{\perp}})$ is the flat cpo on N .

We can generalize this discussion about flat cpos.

Theorem 3.1. Let (D, \sqsubseteq_D) be a partial order, and let C be a finite chain in (D, \sqsubseteq_D) . Then $\sqcup C$ exists and $\sqcup C \in C$.

Proof. We argue by induction on the size of finite chains in (D, \sqsubseteq_D) . If $C = \{c\}$, then obviously $\sqcup C = c$. If $C = \{c_1, \dots, c_{n+1}\}$, then $C' = \{c_1, \dots, c_n\}$ is also a chain, so $\sqcup C' \in C'$ by the induction hypothesis. Now, if $c_{n+1} \sqsubseteq_D \sqcup C'$, then $\sqcup C = \sqcup C' \in C$. Otherwise, $\sqcup C' \sqsubseteq_D c_{n+1}$, since C is a chain, so $\sqcup C = c_{n+1} \in C$. ■

We immediately get

Corollary 3.2. Let (D, \sqsubseteq_D) be a partial order with a bottom element. If every chain in (D, \sqsubseteq_D) is finite, then (D, \sqsubseteq_D) is a cpo.

Corollary 3.3. Every finite partial order with a bottom element is a cpo.

Power sets are another source of cpos. Let D be a set, and let $\mathcal{E} \subseteq \mathcal{P}(D)$. Then the *union* of \mathcal{E} , denoted $\sqcup \mathcal{E}$, is defined

$$\sqcup \mathcal{E} = \{d \in D \mid d \in E \text{ for some } E \in \mathcal{E}\}.$$

It is a basic mathematical fact that $\sqcup \mathcal{E}$ exists.

Theorem 3.4. Let D be a set. Then $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$ is a cpo.

Proof. We have already noted that $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$ is a partial order. For any set $E \in \mathcal{P}(D)$, $\emptyset \subseteq E$, so \emptyset is the bottom element of $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$. Let $\mathcal{E} \subseteq \mathcal{P}(D)$. It is clear that $\sqcup \mathcal{E} \in \mathcal{P}(D)$, and we claim that $\sqcup \mathcal{E} = \sqcup \mathcal{E}$. For any $E \in \mathcal{E}$, we have $E \subseteq \sqcup \mathcal{E}$, so $\sqcup \mathcal{E}$ is an upper bound of \mathcal{E} in $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$. Let $A \in \mathcal{P}(D)$ be any upper bound of \mathcal{E} . Then for any $d \in \sqcup \mathcal{E}$, $d \in E$ for some $E \in \mathcal{E}$, and $E \subseteq A$, so $d \in A$. Therefore, $\sqcup \mathcal{E} \subseteq A$, which implies $\sqcup \mathcal{E} = \sqcup \mathcal{E}$. This argument holds, in particular, when \mathcal{E} is a chain, so $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$ is a cpo. ■

Note that in the proof of Theorem 3.4, we actually showed that *every* subset of $\mathcal{P}(D)$ has a least upper bound. (See Exercise 20 for more on this point.)

The constructions of Section 2 can also be used to construct cpos with a richer structure than flat cpos. Let D_1, \dots, D_n be sets, and let $D = D_1 \times \dots \times D_n$. We define the *projection functions* $\downarrow i: D \rightarrow D_1, \dots, \downarrow n: D \rightarrow D_n$ as follows. For $1 \leq i \leq n$,

$$\downarrow i(d_1, \dots, d_n) = d_i \quad \text{for all } (d_1, \dots, d_n) \in D.$$

Note that if $D_i = N$, for all $1 \leq i \leq n$, then $\downarrow j$ is the function u_j^n from Chapter 3, where $1 \leq j \leq n$. We will write $(d_1, \dots, d_n) \downarrow i$ instead of $\downarrow i(d_1, \dots, d_n)$. If $E \subseteq D$, we write $E \downarrow i$ to denote $\{e \downarrow i \mid e \in E\}$.

Theorem 3.5. Let $(D_1, \sqsubseteq_1), \dots, (D_n, \sqsubseteq_n)$ be partial orders, and let $E \subseteq D_1 \times \dots \times D_n$. Then $\sqcup E$ exists if and only if $\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n)$ exist, and if $\sqcup E$ exists, then $\sqcup E = (\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n))$.

Proof. We will write \sqsubseteq for $\sqsubseteq_{D_1 \times \dots \times D_n}$. Suppose $\sqcup E$ exists, and let $\sqcup E = (e_1, \dots, e_n)$. Then $e_i \in D_i$, $1 \leq i \leq n$, and we claim that $e_i = \sqcup(E \downarrow i)$. For $1 \leq i \leq n$, if $c \in E \downarrow i$, then there is some element

$$(c_1, \dots, c_{i-1}, c, c_{i+1}, \dots, c_n) \in E,$$

and $(c_1, \dots, c_{i-1}, c, c_{i+1}, \dots, c_n) \sqsubseteq (e_1, \dots, e_n)$ implies $c \sqsubseteq_i e_i$, so e_i is an upper bound of $E \downarrow i$. Let d be any upper bound of $E \downarrow i$. Then for any $(c_1, \dots, c_n) \in E$, $c_j \sqsubseteq_j e_j$ for $1 \leq j \leq n$, $j \neq i$, and $c_i \sqsubseteq_i d$, so $(e_1, \dots, d, \dots, e_n)$ is an upper bound of E . But (e_1, \dots, e_n) is the least upper bound of E , so $(e_1, \dots, e_n) \sqsubseteq (e_1, \dots, d, \dots, e_n)$, and, in particular, $e_i \sqsubseteq_i d$, so $e_i = \sqcup(E \downarrow i)$. Therefore, $\sqcup(E \downarrow i)$ exists, $1 \leq i \leq n$, and $\sqcup E = (e_1, \dots, e_n) = (\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n))$.

Now, suppose $\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n)$ exist. Then $(\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n))$ is an element of $D_1 \times \dots \times D_n$, and we claim that it is $\sqcup E$. If $(c_1, \dots, c_n) \in E$, then for $1 \leq i \leq n$, $c_i \in E \downarrow i$, which implies $c_i \sqsubseteq_i \sqcup(E \downarrow i)$, so $(c_1, \dots, c_n) \sqsubseteq (\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n))$. Therefore, $(\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n))$ is an upper bound of E . Let (d_1, \dots, d_n) be any upper bound of E . For $1 \leq i \leq n$, if $c \in E \downarrow i$ then there is some $(c_1, \dots, c, \dots, c_n) \in E$, and $(c_1, \dots, c, \dots, c_n) \sqsubseteq (d_1, \dots, d_n)$ implies $c \sqsubseteq_i d_i$, so d_i is an upper bound of $E \downarrow i$. But then $\sqcup(E \downarrow i) \sqsubseteq_i d_i$, $1 \leq i \leq n$, which implies $(\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n)) \sqsubseteq (d_1, \dots, d_n)$, so $(\sqcup(E \downarrow 1), \dots, \sqcup(E \downarrow n)) = \sqcup E$. ■

Theorem 3.6. If $(D_1, \sqsubseteq_1), \dots, (D_n, \sqsubseteq_n)$ are cpos, then $(D_1 \times \dots \times D_n, \sqsubseteq_{D_1 \times \dots \times D_n})$ is a cpo.

Proof. We will write D for $D_1 \times \dots \times D_n$ and \sqsubseteq for $\sqsubseteq_{D_1 \times \dots \times D_n}$. (D, \sqsubseteq) is a partial order by Theorem 2.1. Let \perp_i be the bottom element of (D_i, \sqsubseteq_i) , $1 \leq i \leq n$. Then $(\perp_1, \dots, \perp_n) \sqsubseteq (d_1, \dots, d_n)$ for all $(d_1, \dots, d_n) \in D$, so (D, \sqsubseteq) has a bottom element.

Now, let C be a chain in (D, \sqsubseteq) . We must show that $\sqcup C$ exists. For $1 \leq i \leq n$, if $c_i, c'_i \in C \downarrow i$, then there are $c, c' \in C$ such that $c_i = c \downarrow i$ and $c'_i = c' \downarrow i$. Since C is a chain, either $c \sqsubseteq c'$ or $c' \sqsubseteq c$, which implies that either $c_i \sqsubseteq_i c'_i$ or $c'_i \sqsubseteq_i c_i$. Therefore $C \downarrow i$ is a chain in cpo (D_i, \sqsubseteq_i) , so $\sqcup(C \downarrow i)$ exists, $1 \leq i \leq n$, and by Theorem 3.5, $\sqcup C$ exists. ■

We can prove a similar result for function space orderings. If $\mathcal{F} \subseteq D \rightarrow E$ for some sets D, E , then for any $d \in D$ we write $\mathcal{F}(d)$ to denote the set $\{f(d) \mid f \in \mathcal{F}\}$.

Theorem 3.7. Let D be a set and (E, \sqsubseteq_E) a partial order, and let $\mathcal{F} \subseteq D \rightarrow E$. Then $\sqcup \mathcal{F}$ exists if and only if $\sqcup(\mathcal{F}(d))$ exists for all $d \in D$, and if $\sqcup \mathcal{F}$ exists then $(\sqcup \mathcal{F})(d) = \sqcup(\mathcal{F}(d))$ for all $d \in D$.

Proof. We will write \sqsubseteq for $\sqsubseteq_{D \rightarrow E}$. Suppose $\sqcup \mathcal{F}$ exists, and let $d \in D$. Then $(\sqcup \mathcal{F})(d)$ is an element of E , and we claim $(\sqcup \mathcal{F})(d) = \sqcup(\mathcal{F}(d))$. For any $f \in \mathcal{F}$, $f \sqsubseteq \sqcup \mathcal{F}$ implies $f(d) \sqsubseteq_E (\sqcup \mathcal{F})(d)$, so $(\sqcup \mathcal{F})(d)$ is an upper bound of $\mathcal{F}(d)$. Let e be any upper bound of $\mathcal{F}(d)$, and let $f_e: D \rightarrow E$ be defined

$$f_e(x) = \begin{cases} e & \text{if } x = d \\ (\sqcup \mathcal{F})(x) & \text{otherwise.} \end{cases}$$

Then for any $f \in \mathcal{F}$, $f(x) \sqsubseteq_E (\sqcup \mathcal{F})(x) = f_e(x)$ for $x \in D$ such that $x \neq d$, and $f(d) \sqsubseteq_E e = f_e(d)$, so f_e is an upper bound of \mathcal{F} . But then $\sqcup \mathcal{F} \sqsubseteq f_e$, and, in particular, $(\sqcup \mathcal{F})(d) \sqsubseteq_E f_e(d) = e$, so $(\sqcup \mathcal{F})(d) = \sqcup(\mathcal{F}(d))$.

Now, suppose $\sqcup(\mathcal{F}(d))$ exists for all $d \in D$. Then the function

$$g(d) = \sqcup(\mathcal{F}(d)) \quad \text{for all } d \in D$$

belongs to $D \rightarrow E$, and we claim that $g = \sqcup \mathcal{F}$. If $f \in \mathcal{F}$, then for any $d \in D$, $f(d) \in \mathcal{F}(d)$, which implies $f(d) \sqsubseteq_E \sqcup(\mathcal{F}(d)) = g(d)$, so $f \sqsubseteq g$. Therefore, g is an upper bound of \mathcal{F} . Let h be any upper bound of \mathcal{F} . For any $d \in D$, if $e \in \mathcal{F}(d)$, then $e = f(d)$ for some $f \in \mathcal{F}$, and $f \sqsubseteq h$ implies $e = f(d) \sqsubseteq_E h(d)$, so $h(d)$ is an upper bound of $\mathcal{F}(d)$. But then $g(d) = \sqcup(\mathcal{F}(d)) \sqsubseteq_E h(d)$ for all $d \in D$, which implies $g \sqsubseteq h$, so $g = \sqcup \mathcal{F}$. ■

Theorem 3.8. If D is a set and (E, \sqsubseteq_E) a cpo, then $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$ is a cpo.

Proof. We will write \sqsubseteq for $\sqsubseteq_{D \rightarrow E}$. $(D \rightarrow E, \sqsubseteq)$ is a partial order by Theorem 2.2. Define the constant function $\perp_{D \rightarrow E}(d) = \perp_E$ for all $d \in D$, where \perp_E is the bottom element of (E, \sqsubseteq_E) . Then $\perp_{D \rightarrow E} \sqsubseteq f$ for all $f \in D \rightarrow E$, so $(D \rightarrow E, \sqsubseteq)$ has a bottom element.

Now, let \mathcal{F} be a chain in $(D \rightarrow E, \sqsubseteq)$. Then $\mathcal{F}(d)$ is a chain in (E, \sqsubseteq_E) for any $d \in D$, since, for any $f(d), g(d) \in \mathcal{F}(d)$, $f \sqsubseteq g$ implies $f(d) \sqsubseteq_E g(d)$ and $g \sqsubseteq f$ implies $g(d) \sqsubseteq_E f(d)$. Therefore $\sqcup(\mathcal{F}(d))$ exists for all $d \in D$, since (E, \sqsubseteq_E) is a cpo, and by Theorem 3.7, $\sqcup \mathcal{F}$ exists. ■

The proofs of the following two theorems are almost identical to the proofs of Theorem 3.7 and Theorem 3.8.

Theorem 3.9. Let A be a set, let $\mathcal{D}(a)$ be a partial order for each $a \in A$, and let $\mathcal{F} \subseteq \text{ch}(D)$. Then $\sqcup \mathcal{F}$ exists if and only if $\sqcup(\mathcal{F}(a))$ exists for all $a \in A$, and if $\sqcup \mathcal{F}$ exists, then $(\sqcup \mathcal{F})(a) = \sqcup(\mathcal{F}(a))$ for all $a \in A$.

Theorem 3.10. Let A be a set and let $\mathcal{D}(a)$ be a cpo for each $a \in A$. Then $(\text{ch}(\mathcal{D}), \sqsubseteq_{\text{ch}(\mathcal{D})})$ is a cpo.

The iteration of our operations for constructing partial orders quickly gives us partial orders of considerable complexity. Theorems 3.6, 3.8, and 3.10 tell us that if we start with cpos, we end up with cpos. For example,

$$\left((N_\perp^2 \rightarrow N_\perp) \rightarrow (N_\perp \rightarrow N_\perp), \sqsubseteq_{(N_\perp^2 \rightarrow N_\perp) \rightarrow (N_\perp \rightarrow N_\perp)} \right)$$

is the cpo of functions that transform binary functions in $N_\perp^2 \rightarrow N_\perp$ into unary functions in $N_\perp \rightarrow N_\perp$. Functions that operate on other functions are sometimes called *higher order functions*. For example $\text{Id}_{D \rightarrow D}: (D \rightarrow D) \rightarrow (D \rightarrow D)$, defined $\text{Id}_{D \rightarrow D}(f) = f$, is an easily described higher order function. One way of defining a higher order function F is to give a definition of the function $F(f)$ for every function f in the domain of F . For example,

$$\text{Id}_{D \rightarrow D}(f)(d) = f(d) \quad \text{for all } d \in D.$$

Note that $\text{Id}_{D \rightarrow D}(f)(d)$ is to be interpreted as $(\text{Id}_{D \rightarrow D}(f))(d)$. Similarly, when we write an expression such as $f(g)(h)(d)$, we mean $((f(g))(h))(d)$. Another example is the composition operator $\circ: (E \rightarrow F) \times (D \rightarrow E) \rightarrow (D \rightarrow F)$, for some sets D, E, F , where, for any $f: E \rightarrow F$ and $g: D \rightarrow E$, $\circ(f, g)$ is defined

$$\circ(f, g)(d) = f(g(d)) \quad \text{for all } d \in D.$$

($\circ(f, g)$ is usually written $f \circ g$.) We will make frequent use of this sort of definition in the next chapter.

One way to show that a partial order (E, \sqsubseteq_E) is a cpo is to build it up explicitly by the constructions we have described. Another is to show that it is contained in another partial order (D, \sqsubseteq_D) known to be a cpo and that the least upper bounds of all chains in (E, \sqsubseteq_E) belong to E .

Theorem 3.11. Let (D, \sqsubseteq_D) be a cpo, let $E \subseteq D$, and let \sqsubseteq_E be the restriction of \sqsubseteq_D to E . If

1. E has a least element with respect to \sqsubseteq_E and
2. $\sqcup_D C \in E$ for all chains C in (E, \sqsubseteq_E) ,

then (E, \sqsubseteq_E) is a cpo and $\sqcup_E C = \sqcup_D C$ for all chains C in (E, \sqsubseteq_E) .

Proof. E has a least element by assumption, so we only need to show that every chain C in (E, \sqsubseteq_E) has a least upper bound in (E, \sqsubseteq_E) , i.e., that

$\sqcup_E C$ exists. If C is a chain in (E, \sqsubseteq_E) , then it is also a chain in (D, \sqsubseteq_D) , and we know that $\sqcup_D C$ exists. We claim that $\sqcup_D C$ is $\sqcup_E C$. For any $c \in C$, we have $c \sqsubseteq_D \sqcup_D C$, which implies $c \sqsubseteq_E \sqcup_D C$, since c and $\sqcup_D C$ are both in E . Therefore, $\sqcup_D C$ is an upper bound of C in (E, \sqsubseteq_E) . Let e be any upper bound of C in (E, \sqsubseteq_E) . Then for all $c \in C$, $c \sqsubseteq_E e$ implies $c \sqsubseteq_D e$, so e is an upper bound of C in (D, \sqsubseteq_D) . Therefore, $\sqcup_D C \sqsubseteq_D e$, which implies $\sqcup_D C \sqsubseteq_E e$ since $\sqcup_D C$ and e are both in E , so $\sqcup_D C = \sqcup_E C$. ■

We give one application of Theorem 3.11 here and another in the next section. The following construction gives us a way of turning an arbitrary partial order into a cpo.

Definition. Let (D, \sqsubseteq_D) be a partial order. A set $E \subseteq D$ is *downward closed* if for all $e \in E$ and all $d \in D$, if $d \sqsubseteq_D e$, then $d \in E$. E is *directed* if for all $c, d \in E$, there is some $e \in E$ such that $c \sqsubseteq_D e$ and $d \sqsubseteq_D e$. An *ideal* of (D, \sqsubseteq_D) is a nonempty, downward closed, directed subset of D . The set of all ideals of (D, \sqsubseteq_D) is denoted $\text{id}(D, \sqsubseteq_D)$. The ordering $\subseteq_{\text{id}(D, \sqsubseteq_D)}$ is $\subseteq_{\mathcal{P}(D)}$ restricted to $\text{id}(D, \sqsubseteq_D)$.

We will write $\text{id}(D)$ when the ordering \sqsubseteq_D is understood.

Theorem 3.12. Let (D, \sqsubseteq_D) be a partial order with a bottom element. Then $(\text{id}(D), \subseteq_{\text{id}(D)})$ is a cpo.

Proof. It is easy to check that $(\text{id}(D), \subseteq_{\text{id}(D)})$ is a partial order. An ideal is nonempty by definition, so $\perp_D \in I$ for any ideal I , since I is downward closed. Moreover, $\{\perp_D\}$ is an ideal, and $\{\perp_D\} \subseteq I$ for any ideal I , so $\{\perp_D\}$ is the bottom element of $(\text{id}(D), \subseteq_{\text{id}(D)})$. Now, let \mathcal{I} be a chain in $(\text{id}(D), \subseteq_{\text{id}(D)})$. It is obvious that $\text{id}(D) \subseteq \mathcal{P}(D)$, so by Theorem 3.11 we need to show only that $\sqcup_{\mathcal{P}(D)} \mathcal{I} \in \text{id}(D)$, i.e., that $\sqcup \mathcal{I}$ is an ideal. \mathcal{I} is a nonempty set of nonempty sets, so $\sqcup \mathcal{I}$ is nonempty. Let $e \in \sqcup \mathcal{I}$, $d \in D$, and $d \sqsubseteq_D e$. Then $e \in I$ for some $I \in \mathcal{I}$, which implies $d \in I$ since I is an ideal. Therefore, $d \in \sqcup \mathcal{I}$, so $\sqcup \mathcal{I}$ is downward closed. Now, if $c, d \in \sqcup \mathcal{I}$, then $c \in I_1$ and $d \in I_2$ for some $I_1, I_2 \in \mathcal{I}$, which implies $c, d \in I_1 \cup I_2$. But $I_1 \cup I_2 \in \mathcal{I}$, since $I_1 \subseteq I_2$ implies $I_1 \cup I_2 = I_2$ and $I_2 \subseteq I_1$ implies $I_1 \cup I_2 = I_1$. Therefore, there is an $e \in I_1 \cup I_2$ such that $c \sqsubseteq_D e$ and $d \sqsubseteq_D e$, since $I_1 \cup I_2$ is directed, and $I_1 \cup I_2 \subseteq \sqcup \mathcal{I}$, so $e \in \sqcup \mathcal{I}$. So $\sqcup \mathcal{I}$ is directed, and it is an ideal. ■

Let (D, \sqsubseteq_D) be a partial order. For each $e \in D$, the *principal ideal generated by e* , denoted $\text{pid}(e)$, is the set $\{d \in D \mid d \sqsubseteq_D e\}$. It is easy to see that $\text{pid}(e)$ is an ideal of (D, \sqsubseteq_D) . The set of principal ideals of (D, \sqsubseteq_D) ,

denoted $\text{pid}(D)$, is $\{\text{pid}(d) \mid d \in D\}$, and $\subseteq_{\text{pid}(D)}$ is the restriction of $\subseteq_{\mathcal{P}(D)}$ to $\text{pid}(D)$. We can think of the partial order $(\text{pid}(D), \subseteq_{\text{pid}(D)})$ as a “copy” of (D, \sqsubseteq_D) in $(\text{id}(D), \subseteq_{\text{id}(D)})$, and any chain C in (D, \sqsubseteq_D) has a “copy” $\{\text{pid}(c) \mid c \in C\}$ in $(\text{id}(D), \subseteq_{\text{id}(D)})$. Going from (D, \sqsubseteq_D) to $(\text{id}(D), \subseteq_{\text{id}(D)})$, then, has the effect of guaranteeing the existence of a least upper bound in $(\text{id}(D), \subseteq_{\text{id}(D)})$ for each (“copy” of a) chain in (D, \sqsubseteq_D) . For this reason, $(\text{id}(D), \subseteq_{\text{id}(D)})$ is called the *ideal completion* of (D, \sqsubseteq_D) . For more on this subject, see Exercise 19.

Exercises

1. Give an example of a partial order that is not a cpo.
2. Give an example of a cpo in which not every chain has a greatest lower bound.
3. Let ω be some object not in N . Give a binary relation \sqsubseteq such that $(N \cup \{\omega\}, \leq \cup \sqsubseteq)$ is a cpo (where \leq is the usual ordering of N).
4. Let (D, \sqsubseteq_D) be a cpo, let C be a chain in (D, \sqsubseteq_D) , and let $d \in D$. Show that if $c \sqsubseteq_D d$ for all $c \in C$, then $\sqcup C \sqsubseteq_D d$.
5. Let (D, \sqsubseteq_D) be a cpo, and let $C_1 \cup C_2$ be a chain in (D, \sqsubseteq_D) , where $C_1, C_2 \subseteq D$. Show that $\sqcup(C_1 \cup C_2) = \sqcup\{\sqcup C_1, \sqcup C_2\}$.
6. Let (D, \sqsubseteq_D) be a cpo, and let C_1, C_2 be chains in (D, \sqsubseteq_D) .
 - (a) Show that if for all $c_1 \in C_1$ there is a $c_2 \in C_2$ such that $c_1 \sqsubseteq_D c_2$, then $\sqcup C_1 \sqsubseteq_D \sqcup C_2$.
 - (b) Show that if for all $c_1 \in C_1$ there is a $c_2 \in C_2$ such that $c_1 \sqsubseteq_D c_2$, and if for all $c_2 \in C_2$ there is a $c_1 \in C_1$ such that $c_2 \sqsubseteq_D c_1$, then $\sqcup C_1 = \sqcup C_2$.
7. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be cpos, and let C be a chain in $(D \times E, \sqsubseteq_{D \times E})$. Show that $\sqcup C = \sqcup(C \downarrow 1 \times C \downarrow 2)$.
8. (a) Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be partial orders such that the largest chain in (D, \sqsubseteq_D) has $m \in N$ elements and the largest chain in (E, \sqsubseteq_E) has $n \in N$ elements. What is the size of the largest chain in $(D \times E, \sqsubseteq_{D \times E})$?

 (b) Let $(D_1, \sqsubseteq_{D_1}), \dots, (D_n, \sqsubseteq_{D_n})$ be partial orders such that the largest chain in (D_i, \sqsubseteq_{D_i}) has $m_i \in N$ elements, $1 \leq i \leq n$. Prove by induction on n that all chains in

$$(D_1 \times \cdots \times D_n, \sqsubseteq_{D_1 \times \cdots \times D_n})$$

are finite.

9. Let D be a set with $m \in N$ elements, and let (E, \sqsubseteq_E) be a partial order in which the largest chain has $n \in N$ elements. What is the size of the largest chain in $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$?
10. Let D be a set and (E, \sqsubseteq_E) a partial order. Show that (E, \sqsubseteq_E) is a cpo if and only if $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$ is a cpo.
11. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be cpos. Show that

$$\left((D \times E) \rightarrow (D \times E), \sqsubseteq_{(D \times E) \rightarrow (D \times E)} \right)$$
 is a cpo.
12. Let D be a set and (E, \sqsubseteq_E) a cpo. Show that

$$\left((D \rightarrow E) \rightarrow (D \rightarrow E), \sqsubseteq_{(D \rightarrow E) \rightarrow (D \rightarrow E)} \right)$$
 is a cpo.
13. Let D be a set, let $E \subseteq D$, let $\mathcal{P}_E(D) = \{A \in \mathcal{P}(D) \mid E \subseteq A\}$, and let $\sqsubseteq_{\mathcal{P}_E(D)}$ be the restriction of $\sqsubseteq_{\mathcal{P}(D)}$ to $\mathcal{P}_E(D)$. Show that $(\mathcal{P}_E(D), \sqsubseteq_{\mathcal{P}_E(D)})$ is a cpo.
14. For sets D, E , let $D \xrightarrow{p} E$ be the set of all partial functions f on D such that the range of f is a subset of E , and let $\sqsubseteq_{D \xrightarrow{p} E}$ be defined as follows: for all $f, g \in D \xrightarrow{p} E$, $f \sqsubseteq_{D \xrightarrow{p} E} g$ if and only if $f \sqsubseteq g$. Show that $(D \xrightarrow{p} E, \sqsubseteq_{D \xrightarrow{p} E})$ is a cpo. [Hint: Note that $D \xrightarrow{p} E$ is a subset of $\mathcal{P}(D \times E)$ and $\sqsubseteq_{D \xrightarrow{p} E}$ is the restriction of $\sqsubseteq_{\mathcal{P}(D \times E)}$ to $D \xrightarrow{p} E$.]
15. (a) Give a partial order (D, \sqsubseteq_D) and a chain C in (D, \sqsubseteq_D) such that C is not an ideal.
 (b) Give a partial order (D, \sqsubseteq_D) and an ideal I of (D, \sqsubseteq_D) such that I is not a chain.
16. Let (D, \sqsubseteq_D) be a partial order.
 - (a) Show that if (D, \sqsubseteq_D) has a bottom element, and if $\sqcup E$ exists for every directed set $E \subseteq D$, then (D, \sqsubseteq_D) is a cpo.
 - (b)* Show that if (D, \sqsubseteq_D) is a cpo, then $\sqcup E$ exists for every directed set $E \subseteq D$.
- 17.* Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be partial orders. The *lexicographic ordering* on $D \times E$, denoted $\sqsubseteq_{L(D \times E)}$, is defined

$$(d_1, e_1) \sqsubseteq_{L(D \times E)} (d_2, e_2) \quad \text{if and only if}$$

$$d_1 \sqsubseteq_D d_2 \text{ or } (d_1 = d_2 \text{ and } e_1 \sqsubseteq_E e_2).$$

Show that if $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ are cpos, then $(D \times E, \sqsubseteq_{L(D \times E)})$ is a cpo.

18.* For any sets D, E , a function $f: D \rightarrow E$ is *onto* if the range of f is all of E . Let (D, \sqsubseteq_D) , (E, \sqsubseteq_E) be partial orders. An *isomorphism* from (D, \sqsubseteq_D) to (E, \sqsubseteq_E) is a one-one, onto function $f: D \rightarrow E$ such that $d \sqsubseteq_D d'$ if and only if $f(d) \sqsubseteq_E f(d')$ for all $d, d' \in D$. (D, \sqsubseteq_D) , (E, \sqsubseteq_E) are *isomorphic* if there is an isomorphism from (D, \sqsubseteq_D) to (E, \sqsubseteq_E) . If $f: D \rightarrow E$ is one-one, then the *inverse* of f , denoted f^{-1} , is defined $f^{-1} = \{(e, d) \in E \times D \mid f(d) = e\}$.

- (a) Show that if f is an isomorphism from partial order (D, \sqsubseteq_D) to partial order (E, \sqsubseteq_E) , then f^{-1} is an isomorphism from (E, \sqsubseteq_E) to (D, \sqsubseteq_D) .
- (b) Let (D, \sqsubseteq_D) , (E, \sqsubseteq_E) be isomorphic partial orders. Show that (D, \sqsubseteq_D) is a cpo if and only if (E, \sqsubseteq_E) is a cpo.
- (c) Let D be a set with $n \in N$ elements and let (E, \sqsubseteq_E) be a partial order. Show that (E^n, \sqsubseteq_{E^n}) and $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$ are isomorphic.
- (d) Let $(D_1, \sqsubseteq_{D_1}), \dots, (D_n, \sqsubseteq_{D_n})$ be partial orders, let $A = \{1, \dots, n\}$, and let $\mathcal{D}(i) = (D_i, \sqsubseteq_{D_i})$, $1 \leq i \leq n$. Show that $(D_1 \times \dots \times D_n, \sqsubseteq_{D_1 \times \dots \times D_n})$ and $(\text{ch}(\mathcal{D}), \sqsubseteq_{\text{ch}(\mathcal{D})})$ are isomorphic.

19.* Let (D, \sqsubseteq_D) be a partial order.

- (a) Let I be a principal ideal generated by some $d \in D$. Show that I is an ideal of (D, \sqsubseteq_D) .
- (b) Show that (D, \sqsubseteq_D) and $(\text{pid}(D), \sqsubseteq_{\text{pid}(D)})$ are isomorphic. [See Exercise 18 for the definition of *isomorphic partial orders*.]

20.* A partial order (D, \sqsubseteq_D) is a *lattice* if $\sqcup\{d, e\}$ and $\sqcap\{d, e\}$ exist for every $d, e \in D$. It is a *complete lattice* if $\sqcup E$ and $\sqcap E$ exist for every $E \subseteq D$.

- (a) Give an example of a cpo that is not a lattice.
- (b) Give an example of a lattice that is not a cpo.
- (c) Let (D, \sqsubseteq_D) be a lattice. Show that for every nonempty finite set $E \subseteq D$, $\sqcup E$ and $\sqcap E$ exist.
- (d) Show that for any set D , $(\mathcal{P}(D), \sqsubseteq_{\mathcal{P}(D)})$ is a complete lattice.
- (e) Show that for any set D , $(\mathcal{P}(D), \supseteq_{\mathcal{P}(D)})$ is a complete lattice.

21.* Let (D, \sqsubseteq_D) be a partial order with $D \neq \emptyset$. (D, \sqsubseteq_D) is *bounded-complete* if $\sqcup B$ exists for every $B \subseteq D$ that has an upper bound in (D, \sqsubseteq_D) .

- (a) Give an example of a cpo that is not bounded-complete.
- (b) Give an example of a bounded-complete partial order that is not a cpo.
- (c) Show that every bounded-complete partial order has a bottom element. [Hint: Consider $\sqcup \emptyset$.]

- (d) Let (D, \sqsubseteq_D) be a bounded-complete partial order. Show that for every nonempty $E \subseteq D$, $\sqcup E$ exists. [Hint: Consider the least upper bound of $\sqcup\{d \in D \mid d \sqsubseteq_D e \text{ for all } e \in E\}$.]
- (e) (D, \sqsubseteq_D) is *well-founded* if every nonempty subset of D has a least element. Prove that every nonempty well-founded partial order is bounded-complete.
- (f) Let (D, \sqsubseteq_D) , (E, \sqsubseteq_E) be bounded-complete cpos. Show that $(D \times E, \sqsubseteq_{D \times E})$ is a bounded-complete cpo.
- (g) Let D be a set and (E, \sqsubseteq_E) a bounded-complete cpo. Show that $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$ is a bounded-complete cpo.
- 22.* Let (D, \sqsubseteq_D) be a cpo. An element $d \in D$ is *compact* (sometimes called *finite*) if for every chain C in (D, \sqsubseteq_D) such that $d \sqsubseteq_D \sqcup C$, there is a $c \in C$ such that $d \sqsubseteq_D c$. The set of compact elements in (D, \sqsubseteq_D) is denoted $K(D)$. (D, \sqsubseteq_D) is *algebraic* if for every $d \in D$, there is a chain $C \subseteq K(D)$ such that $d = \sqcup C$.
- (a) Let (D, \sqsubseteq_D) be a cpo, let $d \in K(D)$, and let C be a chain in (D, \sqsubseteq_D) such that $d = \sqcup C$. Show that $d \in C$.
- (b) Let (D, \sqsubseteq_D) be a cpo in which every chain is finite. Show that (D, \sqsubseteq_D) is algebraic.
- (c) Let D be a set. Show that $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$ is an algebraic cpo.
- (d) Give an example of a cpo (D, \sqsubseteq_D) , a compact $d \in K(D)$, and an infinite chain C in (D, \sqsubseteq_D) such that $d = \sqcup C$.
- (e) Give an example of a cpo that is not algebraic.
- (f) Show that if (D, \sqsubseteq_D) , (E, \sqsubseteq_E) are algebraic cpos, then so is $(D \times E, \sqsubseteq_{D \times E})$.
- (g) Let (D, \sqsubseteq_D) be a partial order. Show that $(\text{id}(D), \subseteq_{\text{id}(D)})$ is an algebraic cpo and that $K(\text{id}(D)) = \text{pid}(D)$.
- (h) Show that if (D, \sqsubseteq_D) is an algebraic cpo, then (D, \sqsubseteq_D) and $(\text{id}(K(D)), \subseteq_{\text{id}(K(D))})$ are isomorphic. [See Exercise 18 for the definition of *isomorphic partial orders*.]

4. Continuous Functions

Consider a computable function f composed with a partially computable function g applied to a number n , where $g(n) \uparrow$. How should we understand the composition $f(g(n))$? One interpretation is that the computation of $g(n)$ never terminates, so f never gets a result from $g(n)$ and $f(g(n))$ must be undefined. In fact, this is the treatment of composition given in Chapter 3.

Definition. Let $(D_1, \sqsubseteq_1), \dots, (D_n, \sqsubseteq_n), (E, \sqsubseteq_E)$ be cpos with bottom elements $\perp_{D_1}, \dots, \perp_{D_n}, \perp_E$. A function $f: D_1 \times \dots \times D_n \rightarrow E$ is *strict* if $f(d_1, \dots, d_n) = \perp_E$ for all (d_1, \dots, d_n) such that $d_i = \perp_{D_i}$ for one or more $1 \leq i \leq n$.

Let D_1, \dots, D_n, E be sets, and let $((D_i)_\perp, \sqsubseteq_{(D_i)_\perp})$, $1 \leq i \leq n$, and $(E_\perp, \sqsubseteq_{E_\perp})$ be the flat cpos on D_1, \dots, D_n, E , with bottom elements \perp_{D_i} , $1 \leq i \leq n$, and \perp_E . For any partial function f on $D_1 \times \dots \times D_n$ with range contained in E , the *strict extension* $f_\perp: (D_1)_\perp \times \dots \times (D_n)_\perp \rightarrow E_\perp$ of f is defined.

$$f_\perp(x_1, \dots, x_n)$$

$$= \begin{cases} \perp_E & \text{if } (x_1, \dots, x_n) \notin D_1 \times \dots \times D_n \\ \perp_E & \text{if } (x_1, \dots, x_n) \in D_1 \times \dots \times D_n \text{ and } f(x_1, \dots, x_n) \uparrow \\ f(x_1, \dots, x_n) & \text{otherwise.} \end{cases}$$

Clearly any such f_\perp is strict. We have not defined what it means for a function in $N_\perp^n \rightarrow N_\perp$, for example, to be computable, but if f is partially computable, then it certainly would be reasonable to consider f_\perp to be partially computable. In the next chapter we will use strict functions in some situations, but we will not require computable functions to be strict. For example, the function $g(x) = 3$ for all $x \in N_\perp$ is not strict, but it certainly should be considered computable. However, some restrictions on computable functions appear to be reasonable.

Consider the elements (\perp_N, \perp_N) , $(\perp_N, 3)$, and $(7, 3)$ of $N_\perp \times N_\perp$. If (\perp_N, \perp_N) approximates $(7, 3)$ and $(\perp_N, 3)$ better approximates $(7, 3)$, then we should expect a function $f: N_\perp \times N_\perp \rightarrow N_\perp$ to behave such that $f(\perp_N, \perp_N)$ approximates $f(7, 3)$ and $f(\perp_N, 3)$ better approximates $f(7, 3)$. That is, if a function gets more information to compute with, it should be able to give a more informative result. It makes little sense, for our purposes, to consider a function f such that, for example, $f(\perp_N, \perp_N) = 6$, $f(\perp_N, 3) = 8$, and $f(7, 3) = \perp_N$. Since 6 and 8 are completely defined, neither approximates the other, and since \perp_N is completely undefined, neither 6 nor 8 approximates \perp_N . We formalize this notion as follows.

Definition. Let (D, \sqsubseteq_D) and (E, \sqsubseteq_E) be partial orders. A function $f: D \rightarrow E$ is *monotonic* if, for all $d, d' \in D$, $d \sqsubseteq_D d'$ implies $f(d) \sqsubseteq_E f(d')$.

It is easy to see that any strict function $f: N_\perp^n \rightarrow N_\perp$ is monotonic, though the reverse is not necessarily true. (See Exercise 1.) Though we will not require computable functions to be strict, we will certainly expect them to be monotonic. For example, let $\text{eq}: N_\perp \times N_\perp \rightarrow N_\perp$ be the equality

predicate on N_{\perp} , that is,

$$\text{eq}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y, \end{cases}$$

where as before 1 represents TRUE and 0 represents FALSE. Then eq is not monotonic, since $(\perp_N, \perp_N) \sqsubseteq_{N_{\perp} \times N_{\perp}} (\perp_N, 0)$ and $\text{eq}(\perp_N, \perp_N) = 1 \not\sqsubseteq_{N_{\perp}} 0 = \text{eq}(\perp_N, 0)$. Now let $\Phi_{\perp}(x, y)$ be the strict extension of the universal function $\Phi(x, y)$ defined in Chapter 4, and let n_{\perp} be the strict extension of the function $n(x) = 0$. Then for all $x, y \in N$,

$$\text{HALT}(x, y) = \text{eq}(0, n_{\perp}(\Phi_{\perp}(x, y))).$$

But we showed in Chapter 4 that $\text{HALT}(x, y)$ is not computable, so if n_{\perp} and Φ_{\perp} are partially computable, then eq certainly is not.

We will make frequent use of the following simple theorem. First we introduce a new piece of notation. If f is a function with domain D , and $E \subseteq D$, then $f(E)$ denotes the set $\{f(e) \mid e \in E\}$.

Theorem 4.1. Let (D, \sqsubseteq_D) , (E, \sqsubseteq_E) be partial orders and $f: D \rightarrow E$ monotonic. If C is a chain in (D, \sqsubseteq_D) , then $f(C)$ is a chain in (E, \sqsubseteq_E) .

Proof. Let $f(d_1), f(d_2) \in f(C)$. Then either $d_1 \sqsubseteq_D d_2$, which implies $f(d_1) \sqsubseteq_E f(d_2)$, or $d_2 \sqsubseteq_D d_1$, which implies $f(d_2) \sqsubseteq_E f(d_1)$. ■

Ordered sets like $(N_{\perp}^n, \sqsubseteq_{N_{\perp}^n})$ are fairly simple, since all chains in $(N_{\perp}^n, \sqsubseteq_{N_{\perp}^n})$ are finite, but when we go on to consider richer structures, we will require a property that is, in general, stronger than monotonicity. Let (D, \sqsubseteq_D) and (E, \sqsubseteq_E) be cpos, and let C be a chain in (D, \sqsubseteq_D) . C might be an infinite set, so that we reach $\sqcup C$ by way of an infinite chain of approximations. For a function $f: D \rightarrow E$ we would like to be able to reach $f(\sqcup C)$ by way of the approximations $\{f(c) \mid c \in C\}$.

Definition. Let (D, \sqsubseteq_D) and (E, \sqsubseteq_E) be partial orders. A function $f: D \rightarrow E$ is *continuous* if, for any chain C in (D, \sqsubseteq_D) such that $\sqcup C$ exists, $\sqcup f(C)$ exists and $f(\sqcup C) = \sqcup f(C)$. $[D \rightarrow E]$ denotes the set of all continuous functions in $D \rightarrow E$. The *continuous function space ordering* on $[D \rightarrow E]$ determined by (E, \sqsubseteq_E) , which is denoted $\sqsubseteq_{[D \rightarrow E]}$, is the restriction of $\sqsubseteq_{D \rightarrow E}$ to $[D \rightarrow E]$.

Note that if (D, \sqsubseteq_D) is a cpo then we can drop the reference to the existence of $\sqcup C$.

Theorem 4.2. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be partial orders, and let $f: D \rightarrow E$.

1. If f is continuous then it is monotonic.
2. If f is monotonic and C is a finite chain in (D, \sqsubseteq_D) , then $\sqcup C$ and $\sqcup f(C)$ exist and $f(\sqcup C) = \sqcup f(C)$.
3. If all chains in (D, \sqsubseteq_D) are finite, then f is continuous if and only if it is monotonic.

Proof. Let f be continuous, and let $d_1, d_2 \in D$ be such that $d_1 \sqsubseteq_D d_2$. Then $\sqcup\{f(d_1), f(d_2)\} = f(\sqcup\{d_1, d_2\}) = f(d_2)$, so $f(d_1) \sqsubseteq_E f(d_2)$, and therefore f is monotonic.

Now, let f be monotonic, and let C be a finite chain in (D, \sqsubseteq_D) . Then $f(C)$ is a finite chain by Theorem 4.1, so $\sqcup C$ and $\sqcup f(C)$ both exist by Theorem 3.1. We have $\sqcup C \in C$ by Theorem 3.1, so $f(\sqcup C) \in f(C)$, which implies $f(\sqcup C) \sqsubseteq_E \sqcup f(C)$. Also, $c \sqsubseteq_D \sqcup C$ for all $c \in C$ implies $f(c) \sqsubseteq_E f(\sqcup C)$ for all $c \in C$ by the monotonicity of f , so that $f(\sqcup C)$ is an upper bound of $f(C)$. Therefore $\sqcup f(C) \sqsubseteq_E f(\sqcup C)$, and we have $f(\sqcup C) = \sqcup f(C)$ by the antisymmetry of \sqsubseteq_E .

Finally, part 3 follows immediately from parts 1 and 2. ■

Suppose $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ are cpos, $f: D \rightarrow E$ is monotonic, and C is a chain in (D, \sqsubseteq_D) . Then $\sqcup C$ exists, and $f(C)$ is a chain by Theorem 4.1, so $\sqcup f(C)$ exists. Therefore, in these circumstances we can drop the reference to the existence of both $\sqcup C$ and $\sqcup f(C)$. The following theorem simplifies matters a bit further and suggests a technique for proving continuity that is often more convenient than going to the definition.

Theorem 4.3. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be cpos, and let $f: D \rightarrow E$. Then f is continuous if and only if: (1) f is monotonic, and (2) $f(\sqcup C) \sqsubseteq_E \sqcup f(C)$ for all chains C in (D, \sqsubseteq_D) .

Proof. Let f be monotonic and let C be a chain in (D, \sqsubseteq_D) . Then for all $c \in C$, $c \sqsubseteq_D \sqcup C$ implies $f(c) \sqsubseteq_E f(\sqcup C)$, so $f(\sqcup C)$ is an upper bound of $f(C)$ and $\sqcup f(C) \sqsubseteq_E f(\sqcup C)$. Then by assumption (2) we have $f(\sqcup C) = \sqcup f(C)$, and f is continuous. The other direction follows from Theorem 4.2. ■

It follows from Theorem 4.2 and Exercise 3.8 that in $N_\perp^n \rightarrow N_\perp$, for instance, monotonicity and continuity are equivalent properties. In general, however, they are not. For example, let $T: (N_\perp \rightarrow N_\perp) \rightarrow N_\perp$ be defined

$$T(f) = \begin{cases} 1 & \text{if } f(n) \neq \perp_N \text{ for all } n \in N \\ \perp_N & \text{otherwise.} \end{cases}$$

That is, $T(f)$ is true just in case f is a total function when its domain and range are restricted to N . We will simply say that f is a total function. T is monotonic since for any $g, h \in N_\perp \rightarrow N_\perp$, if $g \sqsubseteq_{N_\perp \rightarrow N_\perp} h$ then either g is not total and $T(g) = \perp_N \sqsubseteq_{N_\perp} T(h)$, or g is total, which implies that h must be total, so that $T(g) = 1 = T(h)$. However, T is not continuous. For all $m \in N$, let $n_m: N_\perp \rightarrow N_\perp$ be the “step function” defined by

$$n_m(x) = \begin{cases} \perp_N & \text{if } x = \perp_N \\ 0 & \text{if } x \neq \perp_N \text{ and } 0 \leq x \leq m \\ \perp_N & \text{if } x \neq \perp_N \text{ and } x > m. \end{cases}$$

Then $\{n_m \mid m \in N\}$ is a chain, and $T(n_m) = \perp_N$ for all $m \in N$, so $\sqcup\{T(n_m) \mid m \in N\} = \perp_N$. But $(\sqcup\{n_m \mid m \in N\})(x) = 0$ for all $x \in N$, so $T(\sqcup\{n_m \mid m \in N\}) = 1$.

In the next chapter continuity will play a major role in our treatment of computable functions.

With the help of the following lemmas we can prove a version of Theorems 3.7 and 3.8 for continuous functions.

Exchange Lemma. Let (D, \sqsubseteq_D) and (E, \sqsubseteq_E) be partial orders, let (F, \sqsubseteq_F) be a cpo, let $f: D \times E \rightarrow F$ be monotonic, and let C_1 and C_2 be chains in (D, \sqsubseteq_D) , (E, \sqsubseteq_E) , respectively. Then

$$\sqcup\{\sqcup\{f(x, y) \mid y \in C_2\} \mid x \in C_1\}, \sqcup\{\sqcup\{f(x, y) \mid x \in C_1\} \mid y \in C_2\}$$

exist, and they are equal.

Proof. For all $c_1 \in C_1$, $\{f(c_1, y) \mid y \in C_2\}$ is a chain by the monotonicity of the unary function $f(c_1, y)$ and Theorem 4.1, so $\sqcup\{f(c_1, y) \mid y \in C_2\}$ exists. Also, if $c_1, c'_1 \in C_1$ and $c_1 \sqsubseteq_D c'_1$, then for all $c_2 \in C_2$,

$$f(c_1, c_2) \sqsubseteq_F f(c'_1, c_2) \sqsubseteq_F \sqcup\{f(c'_1, y) \mid y \in C_2\},$$

so $\sqcup\{f(c_1, y) \mid y \in C_2\} \sqsubseteq_F \sqcup\{f(c'_1, y) \mid y \in C_2\}$. Therefore,

$$\{\sqcup\{f(x, y) \mid y \in C_2\} \mid x \in C_1\}$$

is a chain and $\{\sqcup\{f(x, y) \mid y \in C_2\} \mid x \in C_1\}$ exists. A similar argument holds for $\{\sqcup\{f(x, y) \mid x \in C_1\} \mid y \in C_2\}$.

Let $c_1 \in C_1$. Then for all $c_2 \in C_2$,

$$f(c_1, c_2) \sqsubseteq_F \sqcup\{f(x, c_2) \mid x \in C_1\} \sqsubseteq_F \sqcup\{\sqcup\{f(x, y) \mid x \in C_1\} \mid y \in C_2\},$$

so

$$\sqcup\{f(c_1, y) \mid y \in C_2\} \sqsubseteq_F \sqcup\{\sqcup\{f(x, y) \mid x \in C_1\} \mid y \in C_2\}. \quad (4.1)$$

But (4.1) is true for all $c_1 \in C_1$, so

$$\sqcup\{\sqcup\{f(x, y) \mid y \in C_2\} \mid x \in C_1\} \sqsubseteq_F \sqcup\{\sqcup\{f(x, y) \mid x \in C_1\} \mid y \in C_2\}.$$

Similarly,

$$\sqcup\{\sqcup\{f(x, y) \mid x \in C_1\} \mid y \in C_2\} \sqsubseteq_F \sqcup\{\sqcup\{f(x, y) \mid y \in C_2\} \mid x \in C_1\},$$

so the lemma follows by the antisymmetry of \sqsubseteq_F . ■

Let (D, \sqsubseteq_D) , (E, \sqsubseteq_E) be cpos and define apply: $([D \rightarrow E] \times D) \rightarrow E$ by

$$\text{apply}(f, d) = f(d).$$

Then apply is monotonic, since if $(f, d_1) \sqsubseteq_{[D \rightarrow E] \times D} (g, d_2)$, then

$$\begin{aligned} \text{apply}(f, d_1) &= f(d_1) \\ &\sqsubseteq_E g(d_1) && \text{since } f \sqsubseteq_{[D \rightarrow E]} g \\ &\sqsubseteq_E g(d_2) && \text{since } d_1 \sqsubseteq_D d_2 \text{ and } g \text{ is monotonic} \\ &= \text{apply}(g, d_2). \end{aligned}$$

Now we can prove

Theorem 4.4. If (D, \sqsubseteq_D) and (E, \sqsubseteq_E) are cpos, then $([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$ is a cpo. Moreover, if \mathcal{F} is a chain in $([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$, then $(\sqcup \mathcal{F})(d) = \sqcup(\mathcal{F}(d))$ for all $d \in D$.

Proof. The bottom element $\perp_{D \rightarrow E}$ defined in the proof of Theorem 3.8 is continuous, so $[D \rightarrow E]$ has a bottom element. By Theorem 3.11, we only need to show that for all chains \mathcal{F} in $([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$, $\sqcup_{D \rightarrow E} \mathcal{F} \in [D \rightarrow E]$, that is, the least upper bound of \mathcal{F} in $(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$, which we know to exist by Theorem 3.8, is continuous. Let \mathcal{F} be a chain in $([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$ and let C be a chain in (D, \sqsubseteq_D) . Then

$$\begin{aligned} (\sqcup_{D \rightarrow E} \mathcal{F})(\sqcup C) &= \sqcup\{f(\sqcup C) \mid f \in \mathcal{F}\} && \text{by Theorem 3.7} \\ &= \sqcup\{\sqcup\{f(c) \mid c \in C\} \mid f \in \mathcal{F}\} && \text{since each } f \in \mathcal{F} \text{ is continuous} \\ &= \sqcup\{\sqcup\{\text{apply}(f, c) \mid c \in C\} \mid f \in \mathcal{F}\} \\ &= \sqcup\{\sqcup\{\text{apply}(f, c) \mid f \in \mathcal{F}\} \mid c \in C\} && \text{by the exchange lemma} \\ &= \sqcup\{\sqcup\{f(c) \mid f \in \mathcal{F}\} \mid c \in C\} \\ &= \sqcup\{(\sqcup_{D \rightarrow E} \mathcal{F})(c) \mid c \in C\} && \text{by Theorem 3.7} \\ &= \sqcup((\sqcup_{D \rightarrow E} \mathcal{F})(C)), \end{aligned}$$

so $\sqcup_{D \rightarrow E} \mathcal{F}$ is continuous, and $([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$ is a cpo. ■

We conclude this section with a result that will be applied in the next chapter. (Also, see Exercise 5.10.) The proof is very similar to the proof of the exchange lemma, so we leave it to Exercise 13.

Diagonal Lemma. Let (D, \sqsubseteq_D) be a partial order and (E, \sqsubseteq_E) a cpo, let $f: D \times D \rightarrow E$ be monotonic, and let C be a chain in (D, \sqsubseteq_D) . Then

$$\sqcup\{\sqcup\{f(x, y) \mid y \in C\} \mid x \in C\} = \sqcup\{f(x, x) \mid x \in C\}.$$

Exercises

1. Let $(D_1, \sqsubseteq_{D_1}), \dots, (D_n, \sqsubseteq_{D_n}), (E, \sqsubseteq_E)$ be flat cpos.
 - (a) Show that every strict function in $D_1 \times \dots \times D_n \rightarrow E$ is monotonic.
 - (b) Give an example of a monotonic function in $D_1 \times \dots \times D_n \rightarrow E$ that is not strict.
 - (c) Give a partial order (D, \sqsubseteq_D) and a strict function in $D \rightarrow D$ that is not monotonic.
2. Let $f: N_\perp^n \rightarrow N_\perp$ satisfy

$$f(x_1, \dots, x_n) = h(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n)),$$

where $h: N_\perp^m \rightarrow N_\perp$ and $g_i: N_\perp^n \rightarrow N_\perp$, $1 \leq i \leq n$, are strict. Show that f is strict.

3. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be partial orders, let $e \in E$, and let $f_e: D \rightarrow E$ be the constant function $f_e(d) = e$. Show that f_e is continuous.
4. Let (D, \sqsubseteq_D) be a partial order, and let $\text{Id}_D: D \rightarrow D$ be the identity function $\text{Id}_D(d) = d$. Show that Id_D is continuous.
5. Let $(D_1, \sqsubseteq_{D_1}), \dots, (D_n, \sqsubseteq_{D_n})$ be partial orders. Show that for $1 \leq i \leq n$, the projection function $\downarrow i: D_1 \times \dots \times D_n \rightarrow D_i$ is continuous.
6. Let D, E be sets, let $f \in D \rightarrow E$, and define $\hat{f}: \mathcal{P}(D) \rightarrow \mathcal{P}(E)$ as $\hat{f}(A) = \{f(a) \mid a \in A\}$. Show that \hat{f} is continuous.
7. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be cpos, and let

$$\circ: [E \rightarrow F] \times [D \rightarrow E] \rightarrow (D \rightarrow F)$$

be the composition operator on continuous functions. Show that the composition of continuous functions is a continuous function. That is, for all $f \in [E \rightarrow F]$, $g \in [D \rightarrow E]$, show that $f \circ g \in [D \rightarrow F]$.

8. Let $(D, \sqsubseteq_D), (E_1, \sqsubseteq_{E_1}), \dots, (E_n, \sqsubseteq_{E_n})$ be cpos, and define the function construct: $[D \rightarrow E_1] \times \dots \times [D \rightarrow E_n] \rightarrow (D \rightarrow (E_1 \times \dots \times E_n))$ as

$$\text{construct}(f_1, \dots, f_n)(d) = (f_1(d), \dots, f_n(d)).$$

Let $f_i \in [D \rightarrow E_i]$, $1 \leq i \leq n$. Show that $\text{construct}(f_1, \dots, f_n) \in [D \rightarrow (E_1 \times \dots \times E_n)]$.

9. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be cpos, and let $\text{apply}_n: [D \rightarrow E] \rightarrow (D^n \rightarrow E^n)$ be defined as $\text{apply}_n(f)(d_1, \dots, d_n) = (f(d_1), \dots, f(d_n))$.
- (a) Let $f \in [D \rightarrow E]$. Show that $\text{apply}_n(f) \in [D^n \rightarrow E^n]$.
 - (b) Show that $\text{apply}_n \in [[D \rightarrow E] \rightarrow [D^n \rightarrow E^n]]$, i.e., apply_n is continuous.
10. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be cpos and define curry: $[D \times E \rightarrow F] \rightarrow (D \rightarrow (E \rightarrow F))$ as $\text{curry}(f)(d)(e) = f(d, e)$. Let $f \in [D \times E \rightarrow F]$ and $d \in D$.
- (a) Show that $\text{curry}(f)(d) \in [E \rightarrow F]$.
 - (b) Show that $\text{curry}(f) \in [D \rightarrow [E \rightarrow F]]$.
 - (c) Show that $\text{curry} \in [[D \times E \rightarrow F] \rightarrow [D \rightarrow [E \rightarrow F]]]$.
11. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be cpos, let $D \xrightarrow{s} E$ be the set of all strict functions in $D \rightarrow E$, and let $\sqsubseteq_{D \xrightarrow{s} E}$ be the restriction of $\sqsubseteq_{D \rightarrow E}$ to $D \xrightarrow{s} E$. Show that $(D \xrightarrow{s}, \sqsubseteq_{D \xrightarrow{s} E})$ is a cpo.
12. Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be cpos, let $D \xrightarrow{m} E$ be the set of all monotonic functions in $D \rightarrow E$, and let $\sqsubseteq_{D \xrightarrow{m} E}$ be the restriction of $\sqsubseteq_{D \rightarrow E}$ to $D \xrightarrow{m} E$. Show that $(D \xrightarrow{m}, \sqsubseteq_{D \xrightarrow{m} E})$ is a cpo.
13. Prove the diagonal lemma.
- 14.* Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ be isomorphic cpos, and let f be an isomorphism from (D, \sqsubseteq_D) to (E, \sqsubseteq_E) . Show that f is continuous. [See Exercise 3.18 for the definition of *isomorphic partial orders*.]
- 15.* Let $(D_1, \sqsubseteq_{D_1}), \dots, (D_n, \sqsubseteq_{D_n}), (E, \sqsubseteq_E)$ be partial orders and let $f \in D_1 \times \dots \times D_n \rightarrow E$. We say that f is *monotonic (continuous) in the i th position*, $1 \leq i \leq n$, if for all $d_j \in D_j$, $1 \leq j \leq n$ and $j \neq i$, the unary function $f(d_1, \dots, d_{i-1}, x, d_{i+1}, \dots, d_n)$ is monotonic (respectively, continuous).
- (a) Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be cpos, let $f: D \times E \rightarrow F$ be monotonic, and let C be a chain in $(D \times E, \sqsubseteq_{D \times E})$. Show that $\sqcup\{\sqcup\{f(x, y) \mid x \in C \downarrow 1\} \mid y \in C \downarrow 2\}$ and $\sqcup f(C)$ exist, and that

$$\sqcup\{\sqcup\{f(x, y) \mid x \in C \downarrow 1\} \mid y \in C \downarrow 2\} = \sqcup f(C).$$

- (b) Let $(D, \sqsubseteq_D), (E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be cpos, and let $f \in D \times E \rightarrow F$. Use part (a) to show that f is continuous if and only if f is continuous in the first and second positions.
- (c) Let (D, \sqsubseteq_D) and (E, \sqsubseteq_E) be cpos. Show that the function $\text{apply}: ([D \rightarrow E] \times D) \rightarrow E$ is continuous.
- (d) Show that \circ , defined in Exercise 7, is continuous.
- (e) Generalize part (b) so that it applies to n -ary functions, $n \geq 1$.
- (f) Show that construct, defined in Exercise 8, is continuous.
- 16.* A *Scott domain* is a bounded-complete algebraic cpo. [See Exercises 3.21 and 3.22 for the definitions of *bounded-complete* and *algebraic cpos*.]
- (a) Show that if $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ are Scott domains, then so is $(D \times E, \sqsubseteq_{D \times E})$.
- (b) Show that $(N_\perp \rightarrow N_\perp, \sqsubseteq_{N_\perp \rightarrow N_\perp})$ is a Scott domain.
- (c) Show that if $(D, \sqsubseteq_D), (E, \sqsubseteq_E)$ are Scott domains, then so is $([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$. [Hint: For each $d \in K(D)$, $e \in K(E)$, define $(d \searrow e): D \rightarrow E$ as
- $$(d \searrow e)(x) = \begin{cases} e & \text{if } d \sqsubseteq_D x \\ \perp_E & \text{otherwise.} \end{cases}$$
- Show that every function $(d \searrow e)$ is continuous and compact. Then show that for every $d_1, \dots, d_n \in K(D)$ and $e_1, \dots, e_n \in K(E)$, the function $(d_1 \searrow e_1) \sqcup \dots \sqcup (d_n \searrow e_n)$ is continuous and compact. Use these functions to show that $([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$ is algebraic.]
- (d) Give an algebraic cpo (D, \sqsubseteq_D) such that $([D \rightarrow D], \sqsubseteq_{[D \rightarrow D]})$ is not algebraic.

5. Fixed Points

We will now prove a fundamental theorem that will facilitate our work on denotational semantics.

Definition. Let (D, \sqsubseteq_D) be a partial order, and let $f \in D \rightarrow D$. An element $d \in D$ is a *fixed point* of f if $f(d) = d$, and it is the *least fixed point* of f in (D, \sqsubseteq_D) if $d \sqsubseteq_D e$ for every fixed point $e \in D$ of f . The least fixed point of f in (D, \sqsubseteq_D) , if it exists, is denoted $\mu_{(D, \sqsubseteq_D)} f$.

We will generally omit the subscript and write μf when the partial order (D, \sqsubseteq_D) is understood.

Fixed points play a fundamental role in denotational semantics, and continuity is important because it allows us to guarantee the existence of least fixed points. Before we can prove the fixed point theorem, we need some notation and a lemma. Let D be a set and let $f \in D \rightarrow D$. For each $n \in N$ we define a function $f^n: D \rightarrow D$, called the n th iteration of f , as follows:

$$\begin{aligned} f^0(x) &= x \\ f^{n+1}(x) &= f(f^n(x)). \end{aligned}$$

Note that these equations can also be understood as defining a single binary function $f^y(x): D \times N \rightarrow D$.

Lemma 1. Let (D, \sqsubseteq_D) be a partial order with bottom element \perp_D , and let $f: D \rightarrow D$ be monotonic. Then $f^n(\perp_D) \sqsubseteq_D f^{n+1}(\perp_D)$ for all $n \in N$, and $\{f^n(\perp_D) \mid n \in N\}$ is a chain in (D, \sqsubseteq_D) .

Proof. First we argue by induction on n that $f^n(\perp_D) \sqsubseteq_D f^{n+m}(\perp_D)$ for all $n, m \in N$. If $n = 0$, then $f^0(\perp_D) = \perp_D \sqsubseteq_D f^m(\perp_D)$, so assume $f^n(\perp_D) \sqsubseteq_D f^{n+m}(\perp_D)$ for all $m \in N$. Then for any $m \in N$,

$$\begin{aligned} f^{n+1}(\perp_D) &= f(f^n(\perp_D)) \\ &\sqsubseteq_D f(f^{n+m}(\perp_D)) \quad \text{by the induction hypothesis and} \\ &\qquad \text{the monotonicity of } f \\ &= f^{n+1+m}(\perp_D). \end{aligned}$$

The lemma now follows immediately. ■

Theorem 5.1 (Fixed Point Theorem for cpos). Let (D, \sqsubseteq_D) be a cpo, and let $f: D \rightarrow D$ be continuous. Then the least fixed point μf exists, and $\mu f = \sqcup\{f^n(\perp_D) \mid n \in N\}$.

Proof. By Lemma 1, $\{f^n(\perp_D) \mid n \in N\}$ is a chain, so $\sqcup\{f^n(\perp_D) \mid n \in N\}$ exists. Moreover, $\sqcup\{f^n(\perp_D) \mid n \in N\}$ is a fixed point of f , since

$$\begin{aligned} f(\sqcup\{f^n(\perp_D) \mid n \in N\}) &= \sqcup f(\{f^n(\perp_D) \mid n \in N\}) \quad \text{by the continuity of } f \\ &= \sqcup\{f^{n+1}(\perp_D) \mid n \in N\} \\ &= \sqcup(\{f^{n+1}(\perp_D) \mid n \in N\} \cup \{f^0(\perp_D)\}) \quad \text{since } f^0(\perp_D) = \perp_D \\ &= \sqcup\{f^n(\perp_D) \mid n \in N\}. \end{aligned}$$

Finally, if e is any fixed point of f , then we argue by induction on n that $f^n(\perp_D) \sqsubseteq_D e$ for all $n \in N$. If $n = 0$ then $f^0(\perp_D) = \perp_D \sqsubseteq_D e$, so assume $f^n(\perp_D) \sqsubseteq_D e$. Then

$$\begin{aligned} f^{n+1}(\perp_D) &= f(f^n(\perp_D)) \\ &\sqsubseteq_D f(e) && \text{by the induction hypothesis and} \\ &= e && \text{the monotonicity of } f \\ &&& \text{since } e \text{ is a fixed point of } f, \end{aligned}$$

so e is an upper bound of $\{f^n(\perp_D) \mid n \in N\}$. Therefore,

$$\sqcup\{f^n(\perp_D) \mid n \in N\} \sqsubseteq_D e,$$

and $\sqcup\{f^n(\perp_D) \mid n \in N\}$ is the least fixed point of f . ■

The fixed point theorem has a variety of applications. One, as we will show in the next chapter, is to justify recursive definitions of functions. Another is to justify inductive definitions of sets. Consider, for example, the definition of propositional formulas in Chapter 12. For simplicity we let \mathcal{A} , the set of atoms, be $\{p, q\}$, and we consider only formulas with the connectives \neg and \supset , so that the alphabet $B = \{p, q, \neg, \supset, (\,), \}\$. An alternative statement of the definition of propositional formulas over \mathcal{A} is the following: the set of propositional formulas over \mathcal{A} is the smallest (with respect to \subseteq) subset of B^* that

1. contains \mathcal{A} ,
2. is closed under the operation that transforms α to $\neg \alpha$, and
3. is closed under the operation that transforms α and β to $(\alpha \supset \beta)$.

In other words, the set of propositional formulas over \mathcal{A} is the smallest set $X \subseteq B^*$ that satisfies

$$\mathcal{A} \cup \{\neg \alpha \mid \alpha \in X\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in X\} \subseteq X. \quad (5.1)$$

Moreover, since (5.1) would still be satisfied if any element of X not required by 1, 2, or 3 were removed, and since we are looking for the smallest X which satisfies (5.1), we can rewrite (5.1) as the equality

$$X = \mathcal{A} \cup \{\neg \alpha \mid \alpha \in X\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in X\}. \quad (5.2)$$

One way of looking at this equation is to consider the right side as a function $\Phi: \mathcal{P}(B^*) \rightarrow \mathcal{P}(B^*)$ that takes subsets $Z \subseteq B^*$ and transforms them to

$$\Phi(Z) = \mathcal{A} \cup \{\neg \alpha \mid \alpha \in Z\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in Z\}.$$

A solution to (5.2), then, is some X such that, when Φ is applied to X , the result is still X ; that is, X is a fixed point of Φ . For example, let Z be

some arbitrary fixed point of Φ . Then

$$Z = \Phi(Z) = \mathcal{A} \cup \{\neg \alpha \mid \alpha \in Z\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in Z\},$$

so Z is a solution to (5.2).

Now, the definition calls for the smallest such set, i.e., $\mu\Phi$, so for the definition to make sense we need to know that $\mu\Phi$ exists. This is where the fixed point theorem is useful. We have already shown that the partial order $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$ is a cpo for any set D , so $(\mathcal{P}(B^*), \subseteq_{\mathcal{P}(B^*)})$ is a cpo. If Φ is continuous then $\mu\Phi$ exists by the fixed point theorem. Let \mathcal{C} be a chain of subsets of B^* . Then $\cup\mathcal{C} = \cup\mathcal{C}$ in $(\mathcal{P}(B^*), \subseteq_{\mathcal{P}(B^*)})$, and

$$\begin{aligned}\Phi(\cup\mathcal{C}) &= \mathcal{A} \cup \{\neg \alpha \mid \alpha \in \cup\mathcal{C}\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in \cup\mathcal{C}\} \\ &= \mathcal{A} \cup \left(\cup \{ \{\neg \alpha \mid \alpha \in C\} \mid C \in \mathcal{C} \} \right) \\ &\quad \cup \left(\cup \{ \{(\alpha \supset \beta) \mid \alpha, \beta \in C\} \mid C \in \mathcal{C} \} \right) \\ &= \cup \{ \mathcal{A} \cup \{\neg \alpha \mid \alpha \in C\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in C\} \mid C \in \mathcal{C} \} \\ &= \cup \{ \Phi(C) \mid C \in \mathcal{C} \} \\ &= \cup \Phi(\mathcal{C}),\end{aligned}$$

so Φ is continuous, and $\mu\Phi$ exists.

Note that, although the preceding definition of the set of propositional formulas mentions the operations that transform α to $\neg \alpha$ and α, β to $(\alpha \supset \beta)$, no mention is made of a process of building up the set from \mathcal{A} by repeated application of these operations. The set we define is simply a certain solution to a certain equation. On the other hand, the definition given in Chapter 12 does mention repeated applications of these operations. The fixed point theorem, which not only tells us that $\mu\Phi$ exists, but also that $\mu\Phi = \cup\{\Phi^i(\emptyset) \mid i \in N\}$, makes the connection between these two versions of the definition. $\Phi(\emptyset), \Phi^2(\emptyset), \dots$, are subsets of $\mu\Phi$ built up by ever more applications of the formula building operations.

Another way of formalizing the notion of “repeated applications” is to give a context-free grammar² Γ such that $L(\Gamma)$ is the set of propositional formulas over \mathcal{A} . In particular, let Γ consist of the productions

$$\begin{array}{ll} S \rightarrow \neg S & S \rightarrow p \\ S \rightarrow (S \supset S) & S \rightarrow q,\end{array}$$

² The reader who is unfamiliar with Chapter 10 may skip to the definition of admissible predicates.

where S is the start symbol. By definition, $L(\Gamma) = \{u \in B^* \mid S \xrightarrow{*} u\}$. Suppose $\alpha, \beta \in L(\Gamma)$. Then $S \xrightarrow{*} \alpha$ and $S \xrightarrow{*} \beta$, which implies

$$S \Rightarrow (S \supset S) \xrightarrow{*} (\alpha \supset S) \xrightarrow{*} (\alpha \supset \beta),$$

so $(\alpha \supset \beta) \in L(\Gamma)$. Similarly, if $\alpha \in L(\Gamma)$, then $S \Rightarrow \neg S \xrightarrow{*} \neg \alpha$ and $\neg \alpha \in L(\Gamma)$.

It seems, then, that $L(\Gamma)$ is the set of propositional formulas over \mathcal{A} , but how can we prove it? By Theorem 1.4 of Chapter 10, $\alpha \in L(\Gamma)$ if and only if there is a derivation tree for α in Γ . We define the *height* of a derivation tree \mathcal{T} , denoted $h(\mathcal{T})$, as follows. If \mathcal{T} consists of exactly one vertex, the root, then $h(\mathcal{T}) = 1$. If \mathcal{T} consists of a root with successors v_1, \dots, v_n , then

$$h(\mathcal{T}) = \max\{h(\mathcal{T}^{v_1}), \dots, h(\mathcal{T}^{v_n})\} + 1,$$

where \mathcal{T}^{v_i} is the subtree of \mathcal{T} with root v_i , $1 \leq i \leq n$. For each $n \in N$, we define

$$L_n = \{u \in B^* \mid \text{there is a derivation tree } \mathcal{T} \text{ for } u \text{ in } \Gamma \text{ with } h(\mathcal{T}) \leq n + 1\}.$$

Clearly, $L(\Gamma) = \cup_{n \in N} L_n$. If we can show that $\Phi^n(\emptyset) = L_n$ for all $n \in N$, then we will have

$$\mu\Phi = \cup_{n \in N} \Phi^n(\emptyset) = \cup_{n \in N} L_n = L(\Gamma).$$

If $n = 0$ then $\Phi^0(\emptyset) = \emptyset = L_0$, since the only derivation tree of height 1 with root S does not yield a word in B^* . For $n > 0$ we argue by induction on n . If $n = 1$ then $\Phi^1(\emptyset) = \{p, q\} = L_1$. For $n + 2$ we have

$$\begin{aligned} \Phi^{n+2}(\emptyset) &= \Phi(\Phi^{n+1}(\emptyset)) \\ &= \Phi(L_{n+1}) \quad \text{by the induction hypothesis} \\ &= \{p, q\} \cup \{\neg \alpha \mid \alpha \in L_{n+1}\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in L_{n+1}\}. \end{aligned}$$

Also, it is clear from the definition of L_{n+2} and the nature of Γ that

$$L_{n+2} = L_{n+1} \cup \{\neg \alpha \mid \alpha \in L_{n+1}\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in L_{n+1}\}.$$

Since $\{p, q\} \subseteq L_{n+1}$, we have $\Phi^{n+2}(\emptyset) \subseteq L_{n+2}$. On the other hand, by Lemma 1 we have $L_{n+1} = \Phi^{n+1}(\emptyset) \subseteq \Phi^{n+2}(\emptyset)$, which implies $L_{n+2} \subseteq \Phi^{n+2}(\emptyset)$, and so we have $\Phi^{n+2}(\emptyset) = L_{n+2}$, completing the induction and the proof that $\mu\Phi = L(\Gamma)$.

These various treatments of the definition of propositional formulas help to illustrate some of the ideas in the next two chapters. On the one hand, we have an abstract mathematical characterization of the set of propositional formulas as the smallest solution to equation (5.2). On the other hand, we have $L(\Gamma) = \{u \in B^* \mid S \xrightarrow{*} u\}$, the set of words generated

from S by derivations in Γ . We can give a “deterministic” characterization of $L(\Gamma)$ as the set of words in B^* for which there exists a leftmost derivation from the start symbol. A somewhat more abstract characterization of the same set is $\cup_{n \in N} L_n$, which is given in terms of derivation trees, without any reference to the details of the choices made in the construction of derivation sequences. In the terminology of semantics, $\mu\Phi$ is a denotational definition, and $L(\Gamma)$ is an operational definition. The link between $\mu\Phi$ and $L(\Gamma)$ is given by the fixed point theorem and its characterization of $\mu\Phi$ as $\sqcup\{\Phi^i(\emptyset) \mid i \in N\}$.

A useful tool for reasoning about fixed points is embodied in the fixed point induction principle.

Definition. Let (D, \sqsubseteq_D) be a cpo. A predicate $P(x)$ on D is *admissible* if the following holds for all chains C in (D, \sqsubseteq_D) :

$$\text{if } P(c) \text{ for all } c \in C, \text{ then } P(\sqcup C).$$

Theorem 5.2 (Fixed Point Induction Principle). Let (D, \sqsubseteq_D) be a cpo, $f: D \rightarrow D$ a continuous function, and $P(x)$ an admissible predicate on D . If

1. $P(\perp_D)$, and
2. $P(f^i(\perp_D))$ implies $P(f^{i+1}(\perp_D))$ for all $i \in N$,

then $P(\mu f)$.

Proof. Ordinary induction shows that $P(f^i(\perp_D))$ holds for all $i \in N$. The set $\{f^i(\perp_D) \mid i \in N\}$ is a chain by Lemma 1, so $P(\sqcup\{f^i(\perp_D) \mid i \in N\})$ holds by the admissibility of $P(x)$, and of course $\sqcup\{f^i(\perp_D) \mid i \in N\} = \mu f$. ■

For example, suppose we define Y as the smallest subset of B^* that

1. contains $\{p, q, \neg p, \neg q\}$,
2. is closed under the operation that transforms α to $\neg \alpha$,
3. is closed under the operation that transforms α and β to $(\alpha \supset \beta)$.

Is Y equal to the set of propositional formulas over \mathcal{A} ? Let

$$\Psi(Z) = \{p, q, \neg p, \neg q\} \cup \{\neg \alpha \mid \alpha \in Z\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in Z\}.$$

Ψ is continuous, by an argument almost identical to the argument that Φ is continuous, so $\mu\Psi$ exists. The question, then, is whether $\mu\Psi = \mu\Phi$. We argue by fixed point induction that $\mu\Psi \subseteq \mu\Phi$. Let $P(X)$ be the predicate $X \subseteq \mu\Phi$. $P(X)$ is admissible, since if \mathcal{C} is a chain in $(\mathcal{P}(B^*), \subseteq_{\mathcal{P}(B^*)})$ and $C \subseteq \mu\Phi$ for all $C \in \mathcal{C}$, then clearly $\cup\mathcal{C} \subseteq \mu\Phi$. The bottom element

of $(\mathcal{P}(B^*), \subseteq_{\mathcal{P}(B^*)})$ is \emptyset , and $\emptyset \subseteq \mu\Phi$. Also, if $\Psi^i(\emptyset) \subseteq \mu\Phi$, then

$$\Psi^{i+1}(\emptyset) = \Psi(\Psi^i(\emptyset))$$

$$\begin{aligned} &\subseteq \Psi(\mu\Phi) \text{ by the induction hypothesis and the monotonicity of } \Psi \\ &= \{p, q, \neg p, \neg q\} \cup \{\neg \alpha \mid \alpha \in \mu\Phi\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in \mu\Phi\} \\ &= \{p, q\} \cup \{\neg \alpha \mid \alpha \in \mu\Phi\} \cup \{(\alpha \supset \beta) \mid \alpha, \beta \in \mu\Phi\} \\ &\quad \text{since } \neg p, \neg q \in \{\neg \alpha \mid \alpha \in \mu\Phi\} \\ &= \Phi(\mu\Phi) \\ &= \mu\Phi. \end{aligned}$$

Therefore, by the fixed point induction principle, we have $\mu\Psi \subseteq \mu\Phi$. A similar induction argument on the admissible predicate $X \subseteq \mu\Psi$ shows that $\mu\Phi \subseteq \mu\Psi$, so we have $\mu\Psi = \mu\Phi$. The point is that both definitions characterize the same set, and the second definition, with its unnecessary reference to $\neg p$ and $\neg q$, can be simplified to the first definition. When fixed points are used to define the meaning of programs, the same technique can be used to show that two programs are equivalent, or to simplify programs.

For sets defined like the set of propositional formulas over \mathcal{A} , fixed point induction is closely related to a form of induction known as *structural induction*. Let $P(x)$ be a property of propositional formulas over \mathcal{A} (rather than sets of formulas). If

1. $P(\alpha)$ for every $\alpha \in \mathcal{A}$,
2. $P(\alpha)$ implies $P(\neg \alpha)$ for all propositional formulas α over \mathcal{A} ,
3. $P(\alpha)$ and $P(\beta)$ implies $P((\alpha \supset \beta))$ for all propositional formulas α, β over \mathcal{A} ,

then the structural induction principle allows us to conclude $P(\alpha)$ for all propositional formulas α over \mathcal{A} . The assumptions $P(\alpha)$ and $P(\beta)$ in 2 and 3 are the *structural induction hypotheses*. To see why the conclusion is valid, let $\hat{P}(X)$ be the property on sets of propositional formulas over \mathcal{A} defined

$$\hat{P}(A) \text{ if and only if } P(\alpha) \text{ for all } \alpha \in A.$$

$\hat{P}(X)$ is admissible: if \mathcal{C} is a chain in $(\mathcal{P}(B^*), \subseteq_{\mathcal{P}(B^*)})$ and $\hat{P}(C)$ for all $C \in \mathcal{C}$, then for each $C \in \mathcal{C}$ we have $P(\alpha)$ for all $\alpha \in C$, which implies $P(\alpha)$ for all $\alpha \in \cup \mathcal{C}$, that is, $\hat{P}(\cup \mathcal{C})$. Now, assumptions 1, 2, and 3 enable us to prove

- $\hat{P}(\emptyset)$,
- $\hat{P}(\Phi^i(\emptyset))$ implies $\hat{P}(\Phi^{i+1}(\emptyset))$ for all $i \in N$,

from which we conclude, by fixed point induction, $\hat{P}(\mu\Phi)$, i.e., $P(\alpha)$ for all $\alpha \in \mu\Phi$, the set of propositional formulas over \mathcal{A} . We will give several structural induction arguments in the next chapter.

The reader will recall that we proved a theorem in Chapter 4 that was also called a fixed point theorem. The two are closely related. The recursion theorem from Chapter 4 is sometimes called, for historical reasons, the *second recursion theorem*. In fact, the earlier fixed point theorem, which follows from (and just as easily implies) the second recursion theorem, is itself sometimes called the *second recursion theorem*. The fixed point theorem in this chapter is a version of a classical theorem from computability theory that is sometimes called the *first recursion theorem*.

The names of these two recursion theorems come from the fact that they can both be used in proving functions to be partially computable, particularly functions defined by recursion. However, there is a significant distinction between the two theorems. The fixed point theorem in this chapter gives a fixed point for each continuous function on a cpo. In particular, if $F: [N_\perp \rightarrow N_\perp] \rightarrow [N_\perp \rightarrow N_\perp]$ is continuous, then we get a function $f \in [N_\perp \rightarrow N_\perp]$ such that $F(f) = f$. On the other hand, the fixed point theorem in Chapter 4 is more directly concerned with programs than functions. A computable function g gets the effect of transforming a function $\Phi_{\#(\mathcal{P})}$ to the function $\Phi_{g(\#(\mathcal{P}))}$ by acting on the (code of the) program that computes $\Phi_{\#(\mathcal{P})}$, and the fixed point theorem in Chapter 4 gives a program \mathcal{Q} such that $\Phi_{\#(\mathcal{Q})} = \Phi_{g(\#(\mathcal{Q}))}$. It would be reasonable, then, to call that earlier theorem a *syntactic* fixed point theorem and to call the current theorem a *semantic* fixed point theorem.

Just as the second recursion theorem gives a partially computable function $\Phi_{\#(\mathcal{Q})}$ that satisfies $\Phi_{\#(\mathcal{Q})} = \Phi_{g(\#(\mathcal{Q}))}$, so too does the first recursion theorem give, for the appropriate kind of F , a partially computable μF . We will say no more about the first recursion theorem in its classical form (other than to direct the reader to Exercise 11), but as we shall see, the main point of the next chapter is to use the fixed point theorem for cpos to define partially computable functions.

Exercises

1. Give functions $f, g, h: N_\perp \rightarrow N_\perp$ such that
 - (a) f has no fixed points;
 - (b) g has exactly one fixed point;
 - (c) h has infinitely many fixed points.

2. Give a function $f: N_\perp \rightarrow N_\perp$ such that f is not continuous and μf exists.
3. Give a partial order (D, \sqsubseteq_D) and a function $f: D \rightarrow D$ such that f is continuous, f has at least one fixed point, and μf does not exist.
4. Give a fixed point characterization of the set of **W**-terms defined at the beginning of Chapter 13, where **W** is some vocabulary.
5. Let (D, \sqsubseteq_D) be a cpo, let $f \in [D \rightarrow D]$, let $E = \{e \in D \mid f(e) = e\}$, and let \sqsubseteq_E be the restriction of \sqsubseteq_D to E . Show that (E, \sqsubseteq_E) is a cpo.
6. Let (D, \sqsubseteq_D) be a cpo.
 - (a) Let $P(X)$ be the predicate on $\mathcal{P}(D)$ defined “ X is a finite set.” In the context of $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$, is P admissible?
 - (b) Let $Q(X)$ be the predicate on $\mathcal{P}(D)$ defined “ X is an infinite set.” In the context of $(\mathcal{P}(D), \subseteq_{\mathcal{P}(D)})$, is Q admissible?
 - (c) Let $R(f)$ be the predicate on $D \rightarrow D$ defined “ f is strict.” In the context of $(D \rightarrow D, \sqsubseteq_{D \rightarrow D})$, is R admissible?
7. Let (D, \sqsubseteq_D) , (E, \sqsubseteq_E) be cpos, let $P(x)$, $Q(x)$ be admissible predicates on D , and let $R(x, y)$ be an admissible predicate on $D \times E$.
 - (a) Show that $P(x) \& Q(x)$ is admissible.
 - (b) Show that $P(x) \vee Q(x)$ is admissible.
 - (c) Show that $(\forall d \in D)R(d, y)$ is an admissible predicate on E .
- 8.* (a) Let Γ be a context-free grammar with variables \mathcal{V} and terminals T . Give a fixed point characterization of $L(\Gamma)$. [Hint: Define a function Φ such that $\mu\Phi(V) = \{w \in T^* \mid V \xrightarrow{*} w\}$ for all $V \in \mathcal{V}$.]

 (b) Let Γ be the grammar with $\mathcal{V} = \{S\}$, $T = \{a\}$, and the single production $S \rightarrow aSa$. Show by fixed point induction that $L(\Gamma) = \emptyset$.
- 9.* Let (D, \sqsubseteq_D) be a complete lattice, and let $f: D \rightarrow D$ be monotonic. Show that μf exists and that $\mu f = \sqcap\{d \in D \mid f(d) \sqsubseteq_D d\}$. [See Exercise 3.20 for the definition of *complete lattices*.]
- 10.* Let (D, \sqsubseteq_D) be a cpo, and define $\mu_D: [D \rightarrow D] \rightarrow D$ as $\mu_D(f) = \mu f$ for all $f \in [D \rightarrow D]$.
 - (a) Let $f, g \in [D \rightarrow D]$, and suppose $f \sqsubseteq_{[D \rightarrow D]} g$. Show by induction on n that $f^n \sqsubseteq_{[D \rightarrow D]} g^n$ for all $n \in \mathbb{N}$.
 - (b) Let \mathcal{F} be a chain in $([D \rightarrow D], \sqsubseteq_{[D \rightarrow D]})$. Show by induction on n that $(\sqcup \mathcal{F})^n = \sqcup\{f^n \mid f \in \mathcal{F}\}$. [Hint: Use the diagonal lemma.]

- (c) Show that μ_D is continuous. [Hint: Use parts (a) and (b) and the exchange lemma.]

11.* As in Exercise 3.14, $N \xrightarrow{p} N$ is the set of all partial functions on N . For each finite function $\theta = \{(x_1, y_1), \dots, (x_n, y_n)\}$ in $N \xrightarrow{p} N$, $n \geq 0$, we encode θ as $\tilde{\theta} \in N$, where

$$\tilde{\theta} = \prod_{i=1}^n p_{x_i}^{y_i+1}.$$

A function $F: (N \xrightarrow{p} N) \rightarrow (N \xrightarrow{p} N)$ is a *recursive operator* if there is some partially computable function $h(y, x)$ such that

$$F(g)(x) = z \text{ if and only if } h(\tilde{\theta}, x) = z \text{ for some } \theta \subseteq g.$$

- (a) Let $G: (N \xrightarrow{p} N) \rightarrow (N \xrightarrow{p} N)$ be defined $G(f)(x) = 2 \cdot f(x)$. Show that G is a recursive operator.
- (b) Show that every recursive operator is monotonic and continuous.
- (c) Show that, if F is a recursive operator, then there is a computable function f such that $F(\Phi_x) = \Phi_{f(x)}$ for all $x \in N$.
- (d) (First Recursion Theorem) Prove that, for every recursive operator F , μF exists and is partially computable.

17

Denotational Semantics of Recursion Equations

1. Syntax

Now that we have developed a theory of approximation orders, we can define recursion equations and give them a denotational semantics. The operational semantics, given in the next chapter, will show that the functions defined by recursion equations are, in a reasonable sense, computable.

As in Chapter 13,¹ where we defined the terms and formulas of quantification theory, we begin with a small alphabet

$$A = \{t, x, f, I, \times, \rightarrow, \#, (,,,), =\}$$

of symbols that are always available. The members of

$$\text{VAR}_T = \{t^{[i]} \mid i \in N\}$$

are *type variables*, and a *type* is

- a type variable, or
- $\tau_1 \times \cdots \times \tau_n$, $n \geq 1$, where τ_1, \dots, τ_n are type variables, or
- $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, $n \geq 1$, where $\tau_1, \dots, \tau_n, \tau$ are type variables.

¹ Knowledge of Chapter 13 is not assumed, but there is a substantial overlap in the treatment of the syntax and semantics of terms.

These three kinds of types are *individual types*, *product types*, and *function types*, respectively. For a given $\tau \in \text{VAR}_T$, the members of $\text{VAR}_I^\tau = \{x \# \tau \# I^{[i]} \mid i \in N\}$ are *individual variables of type τ* , and

$$\text{VAR}_I = \bigcup_{\tau \in \text{VAR}_T} \text{VAR}_I^\tau$$

is the set of all individual variables. For a function type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, the members of

$$\text{VAR}_F^{\tau_1 \times \cdots \times \tau_n \rightarrow \tau} = \{f \# \tau_1 \times \cdots \times \tau_n \rightarrow \tau \# I^{[i]} \mid i \in N\}$$

are the *function variables of type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$* , and

$$\text{VAR}_F = \bigcup_{\tau_1, \dots, \tau_n, \tau \in \text{VAR}_T} \text{VAR}_F^{\tau_1 \times \cdots \times \tau_n \rightarrow \tau}$$

is the set of all function variables. Also, $\text{VAR} = \text{VAR}_I \cup \text{VAR}_F$. We will let X, Y, Z (possibly subscripted) stand for individual variables and F, G, H (possibly subscripted) stand for function variables.² Occasionally we will write V for an arbitrary variable of either kind. We will also use more suggestive names in the examples. If O is any of the syntactic objects defined in this section (W-terms, W-programs, etc.), then $IV(O)$ is the set of all individual variables which occur in O , $FV(O)$ is the set of all function variables which occur in O , and $V(O) = IV(O) \cup FV(O)$.

A *typed vocabulary* is a pair (W, τ) , where W is a finite set of *function symbols* distinct from the symbols in A , and τ is a function on W such that for each $f \in W$, $\tau(f)$ is either an individual type or a function type. We say that $\tau(f)$ is the *type of f* . f is a *constant symbol* if $\tau(f)$ is an individual type, and it is a *proper function symbol* otherwise. Given τ , it is easy to determine the *arity* of any $f \in W$, denoted $ar(f)$. If f is a constant symbol, then $ar(f) = 0$, and if $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, then $ar(f) = n$. It will also be useful to supplement τ with the functions δ and ρ , which give the *domain type* and *range type*, respectively, of symbols in W . For constant symbols $c \in W$, $\delta(c)$ is undefined and $\rho(c) = \tau(c)$, and for proper function symbols $f \in W$ with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$,

$$\delta(f) = \tau_1 \times \cdots \times \tau_n \quad \text{and} \quad \rho(f) = \tau.$$

² Note that the letter X is not itself an individual variable. It is what we sometimes call a *metavariable*. That is, it is a variable, which we use in talking about the syntax of recursion equations, whose *values* are individual variables. Similarly, F is a metavariable whose values are function variables. We also use metavariables such as τ, c, f, t , and P , whose values are type variables, constant symbols, function symbols, terms, and programs, respectively.

We extend τ and ρ to VAR and δ to VAR_F in the obvious way. For example, $\tau(x \# \tau \# l^{(i)}) = \tau$ and $\delta(f \# \tau_1 \times \cdots \times \tau_n \rightarrow \tau \# l^{(i)}) = \tau_1 \times \cdots \times \tau_n$. For a typed vocabulary (W, τ) , $TV(W, \tau)$ is the set of all type variables that occur in the types of all of the symbols in W . We will omit τ and write $TV(W)$ for $TV(W, \tau)$.

Let (W, τ) be a typed vocabulary. For any $\tau \in TV(W)$, a $W\text{-term of type } \tau$ is

- an individual variable of type τ , or
- $c \in W$, where $\tau(c) = \tau$, or
- $g(t_1, \dots, t_n)$, where $g \in W$, $\tau(g) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, and t_i is a W -term of type τ_i , $1 \leq i \leq n$, or
- $F(t_1, \dots, t_n)$, where $F \in \text{VAR}_F$, $\tau(F) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, and t_i is a W -term of type τ_i , $1 \leq i \leq n$.

We extend τ so that $\tau(t) = \tau$ for any term t of type τ . For $V_0 \subseteq \text{VAR}$, $\text{TM}_W^\tau(V_0)$ is the set of all W -terms t of type τ such that $V(t) \subseteq V_0$, and $\text{TM}_W(V_0)$ is the set of all W -terms t such that $V(t) \subseteq V_0$. Also, we will write TM_W^τ for $\text{TM}_W^\tau(\emptyset)$ and TM_W for $\text{TM}_W(\emptyset)$. Terms in TM_W , that is, W -terms without variables, are sometimes called *ground W-terms*.

For example, let N be a type variable, and let (W_1, τ_1) be the typed vocabulary with $W_1 = \{0, s\}$, $\tau_1(0) = N$ and $\tau_1(s) = N \rightarrow N$. We have $TV(W_1) = \{N\}$ and $\text{TM}_{W_1} = \{0, s(0), s(s(0)), \dots\}$. This is a vocabulary suitable for naming the natural numbers. We call terms of the form

$$\overbrace{s(\dots s)}^n(0) \dots, \quad n \in N,$$

numerals, which we will generally write as n or $s^n(0)$.

Now, let NL be a type variable distinct from N , and let (W_2, τ_2) be the typed vocabulary with $W_2 = \{0, s, \text{nil}, \text{cons}\}$, $\tau_2(0) = N$, $\tau_2(s) = N \rightarrow N$, $\tau_2(\text{nil}) = NL$, and $\tau_2(\text{cons}) = N \times NL \rightarrow NL$. Then $TV(W_2) = \{N, NL\}$ and TM_{W_2} is

$$\text{TM}_{W_1} \cup \{\text{nil}, \text{cons}(0, \text{nil}), \text{cons}(s(0), \text{nil}), \text{cons}(0, \text{cons}(0, \text{nil})), \dots\}.$$

We might use this vocabulary for naming lists of numbers. The idea is that a list is either empty or it is constructed from a first element and a list of all succeeding elements. (The reader familiar with the programming language LISP will recognize **cons** and **nil**.)

A W -recursion equation is an equation of the form $F(X_1, \dots, X_n) = t$, where, for some $\tau \in TV(W)$

1. X_1, \dots, X_n are distinct individual variables, F is a function variable, and $\tau(F) = \tau(X_1) \times \cdots \times \tau(X_n) \rightarrow \tau$, and
2. $t \in \text{TM}_W^\tau(\{X_1, \dots, X_n\} \cup \text{VAR}_F)$.

If \mathbf{E} is the \mathbf{W} -recursion equation $\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{t}$, then \mathbf{F} is the *principal function variable* of \mathbf{E} , denoted $PF(\mathbf{E})$, and any function variable that occurs in \mathbf{t} is an *auxiliary function variable* of \mathbf{E} . Note that a function variable can be both principal and auxiliary in a given equation. $AF(\mathbf{E})$ is the set of auxiliary function variables of \mathbf{E} . A *\mathbf{W} -recursion program* (or simply *\mathbf{W} -program*) is a finite set $\{\mathbf{E}_1, \dots, \mathbf{E}_n\}$, $n \geq 0$, of \mathbf{W} -recursion equations such that

1. $PF(\mathbf{E}_i) \neq PF(\mathbf{E}_j)$ for $1 \leq i < j \leq n$, and
2. $\bigcup_{i=1}^n AF(\mathbf{E}_i) \subseteq \{PF(\mathbf{E}_1), \dots, PF(\mathbf{E}_n)\}$.

If equation \mathbf{E} in \mathbf{W} -program \mathbf{P} is $\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{t}$, then \mathbf{E} is the *defining equation* for \mathbf{F} in \mathbf{P} . The first restriction in the definition of \mathbf{W} -programs prevents inconsistencies, and the second ensures that every function variable that occurs on the right side of any equation is defined. When some program \mathbf{P} is given and $\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{t}$ is the defining equation for \mathbf{F} in \mathbf{P} , we will sometimes write $rhs(\mathbf{F})$ to denote the term \mathbf{t} on the righthand side of the equation.

Note that we require each function to be defined by exactly one equation, while in Chapter 3 we used two equations to define a function by recursion. For example,

$$\begin{aligned} + (x, 0) &= x \\ + (x, y + 1) &= s(+ (x, y)) \end{aligned}$$

is a (somewhat informal) definition of addition. Another way of describing addition is

$$+ (x, y) = \begin{cases} x & \text{if } y = 0 \\ s(+ (x, y \dot{-} 1)) & \text{otherwise,} \end{cases} \quad (1.1)$$

which can be construed as a single equation if the if-then-else test is itself a function. That is, given the function

$$\text{if}(b, x, y) = \begin{cases} x & \text{if } b = \text{TRUE} \\ y & \text{otherwise,} \end{cases}$$

we can rewrite (1.1) as

$$+ (x, y) = \text{if}(y = 0, x, s(+ (x, y \dot{-} 1))). \quad (1.2)$$

Of course, we also need the predicate $y = 0$ and the predecessor function $y \dot{-} 1$ for (1.2) to be meaningful. Therefore, we impose the following conditions on the vocabularies we will use.

Let **Bool** be some particular type variable and let **tt**, **ff** be two new symbols. (It does not matter which type variable we choose for **Bool**, but it

will remain fixed throughout.) A *standard constructor vocabulary* is any vocabulary (W_c, τ_c) such that $\text{tt}, \text{ff} \in W_c$, with $\tau_c(\text{tt}) = \tau_c(\text{ff}) = \text{Bool}$, and such that for each $\tau \in TV(W_c)$ there is at least one constant symbol $c \in W_c$ with $\tau_c(c) = \tau$. The latter requirement is not strictly necessary, but it will turn out to be convenient. We create a set of *built-in function symbols* for a standard constructor vocabulary (W_c, τ_c) as follows. Let $W_c^- = W_c - \{\text{tt}, \text{ff}\}$ and let ρ_c be the range type function derived from τ_c . For each $\tau \in TV(W_c)$ we create the new symbol if_τ , for each $f \in W_c^-$ we define the set of new symbols

$$B(f) = \{\text{is_}f\} \cup \{f_i^{-1} \mid 1 \leq i \leq \text{ar}(f)\},$$

and we define

$$B(W_c) = \{\text{if}_\tau \mid \tau \in TV(W_c)\} \cup \bigcup_{f \in W_c^-} B(f).$$

Note that $\{f_i^{-1} \mid 1 \leq i \leq \text{ar}(f)\} = \emptyset$ if f is a constant symbol. We assign types to these new symbols with $\tau_{B(W_c)}$:

$$\tau_{B(W_c)}(\text{if}_\tau) = \text{Bool} \times \tau \times \tau \rightarrow \tau \quad \text{for each } \tau \in TV(W_c)$$

$$\tau_{B(W_c)}(\text{is_}f) = \tau \rightarrow \text{Bool} \quad \text{where } \rho_c(f) = \tau$$

$$\tau_{B(W_c)}(f_i^{-1}) = \tau \rightarrow \tau_i \quad \text{where } \tau_c(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau.$$

A *standard vocabulary* is any typed vocabulary (W, τ) such that

$$(W, \tau) = (W_c \cup B(W_c), \tau_c \cup \tau_{B(W_c)})$$

for some standard constructor vocabulary (W_c, τ_c) . The symbols in W_c are *constructor symbols*: they are used to build up data objects. tt and ff , in particular, will be used to represent TRUE and FALSE. The $\text{is_}f$ symbols are *discriminator function symbols*: they are used to determine how an object is constructed. The f_i^{-1} symbols are *selector function symbols*: they are used to decompose compound objects.

For example, we can expand the typed vocabulary (W_1, τ_1) given above to the standard constructor vocabulary (W_3, τ_3) , where $W_3 = \{\text{tt}, \text{ff}, \text{0}, \text{s}\}$, $\tau_3(\text{tt}) = \tau_3(\text{ff}) = \text{Bool}$, $\tau_3(\text{0}) = \text{N}$, and $\tau_3(\text{s}) = \text{N} \rightarrow \text{N}$. Then

$$B(W_3) = \{\text{if}_{\text{Bool}}, \text{if}_{\text{N}}, \text{is_0}, \text{is_s}, \text{s}_1^{-1}\},$$

where

$$\tau_{B(W_3)}(\text{if}_{\text{Bool}}) = \text{Bool} \times \text{Bool} \times \text{Bool} \rightarrow \text{Bool}$$

$$\tau_{B(W_3)}(\text{if}_N) = \text{Bool} \times N \times N \rightarrow N$$

$$\tau_{B(W_3)}(\text{is_0}) = N \rightarrow \text{Bool}$$

$$\tau_{B(W_3)}(\text{is_s}) = N \rightarrow \text{Bool}$$

$$\tau_{B(W_3)}(s_1^{-1}) = N \rightarrow N,$$

and we can rewrite Eq. (1.2) as

$$+ (X, Y) = \text{if}_N(\text{is_0}(Y), X, s(+ (X, s_1^{-1}(Y)))), \quad (1.3)$$

where $+$ $\in \text{VAR}_F$.

Similarly, we can expand (W_2, τ_2) to (W_4, τ_4) , where $W_4 = \{\text{tt}, \text{ff}, 0, s, \text{nil}, \text{cons}\}$, $\tau_4(\text{tt}) = \tau_4(\text{ff}) = \text{Bool}$, $\tau_4(0) = N$, $\tau_4(s) = N \rightarrow N$, $\tau_4(\text{nil}) = NL$, and $\tau_4(\text{cons}) = N \times NL \rightarrow NL$. Then

$$B(W_4) = B(W_3) \cup \{\text{if}_{NL}, \text{is_nil}, \text{is_cons}, \text{cons}_1^{-1}, \text{cons}_2^{-1}\}.$$

Henceforth, we let (W_N, τ_N) be the standard vocabulary based on (W_3, τ_3) and we let (W_{NL}, τ_{NL}) be the standard vocabulary based on (W_4, τ_4) . That is,

$$W_N = \{\text{tt}, \text{ff}, 0, s, \text{if}_{\text{Bool}}, \text{if}_N, \text{is_0}, \text{is_s}, s_1^{-1}\}$$

and

$$W_{NL} = W_N \cup \{\text{nil}, \text{cons}, \text{if}_{NL}, \text{is_nil}, \text{is_cons}, \text{cons}_1^{-1}, \text{cons}_2^{-1}\}.$$

Generally we will just write τ for τ_N or τ_{NL} .

Note that we intend to interpret s_1^{-1} as the predecessor function in Eq. (1.3). At this point, of course, (1.3) has no meaning at all. The task of giving a meaning to equations like (1.3) begins in the next section.

Exercises

- Let $\tau(X) = N$ and $\tau(F) = N \times N \rightarrow N$. Describe $\text{TM}_{W_N}(\{X, F\})$.
- Let $W_c = \{\text{tt}, \text{ff}, 0, s, \text{leaf}, \text{tree}\}$, and let $\tau_c(\text{tt}) = \tau_c(\text{ff}) = \text{Bool}$, $\tau_c(0) = N$, $\tau_c(s) = N \rightarrow N$, $\tau_c(\text{leaf}) = T$, and $\tau_c(\text{tree}) = N \times T \times T \rightarrow T$.
 - Describe $\text{TM}_{W_c}^\tau$ for each $\tau \in TV(W_c)$.
 - Describe $(B(W_c), \tau_{B(W_c)})$.

- (c) Let $(W, \tau) = (W_c \cup B(W_c), \tau_c \cup \tau_{B(W_c)})$. Describe TM_W^τ for each $\tau \in TV(W)$.
3. Let $\tau(X) = N$, $\tau(Y) = NL$, $\tau(F) = N \times N \rightarrow N$, $\tau(G) = N \rightarrow N$, and $\tau(H) = N \rightarrow NL$. Which of the following are W_{NL} -terms?
- $F(s(X))$.
 - $\text{cons}(G(X), \text{cons}(X, \text{nil}))$.
 - $\text{cons}(\text{nil}, \text{cons}(0, Y))$.
 - $\text{cons}_1^{-1}(H(0))$.
 - $\text{if}_N(\text{ff}, \text{cons}(0, \text{nil}), 0)$.
 - $\text{if}_{NL}(\text{ff}, \text{cons}(0, \text{nil}), H(\text{if}_N(\text{ff}, X, 0)))$.
4. Assume that each of the following are W_{NL} -terms. Give the types of the variables in each term.
- $\text{if}_N(F(0), G(0), s(H(0)))$.
 - $s_1^{-1}(F(\text{cons}(X, G(s(Y)))))$.
 - $\text{is_cons}(F(s_1^{-1}(X), \text{if}_N(G(0), H(G(Y)), 0), X))$.
 - $F(\text{is_0}(F(X)))$.
5. Describe the values of τ and the types of X , Y , F , G , H that make $F(\text{if}_\tau(\text{tt}, G(F(X), Y), H(s(Y))))$ a W_{NL} -term.
6. Let (W, τ) be a vocabulary, let $V \subseteq \text{VAR}$, and extend τ to $(A \cup W)^*$ so that $\tau(w) = \tau$ if and only if $w = \phi u$ for some $\phi \in W \cup V$ with $\rho(\phi) = \tau$.
- Give a fixed point definition of $TM_W(V)$ in the manner of the definition of the set of propositional formulas given in Section 5 of Chapter 16.
 - State and prove a structural induction principle for $TM_W(V)$.

2. Semantics of Terms

We develop the semantics of W -programs in several stages. In this section we work on the semantics of terms, beginning with the semantics of vocabularies. We will work exclusively with standard vocabularies, so throughout the rest of this chapter we take (W, τ) to be some arbitrary standard vocabulary based on some standard constructor vocabulary (W_c, τ_c) . We will generally refer simply to W rather than (W, τ) .

Definition. A *type assignment* for W is a function \mathcal{T} with domain $TV(W)$ such that

- for each $\tau \in TV(W)$, $\mathcal{T}(\tau)$ is a partial order $(D_{\mathcal{T}(\tau)}, \sqsubseteq_{\mathcal{T}(\tau)})$ with bottom element $\perp_{\mathcal{T}(\tau)}$, and, in particular,

2. $\mathcal{T}(\text{Bool})$ is the flat cpo on some set with exactly two elements.

\mathcal{T} is a *complete type assignment* for \mathbf{W} if $\mathcal{T}(\tau)$ is a cpo for each $\tau \in TV(\mathbf{W})$.

A type assignment for \mathbf{W} gives a meaning for each type variable in $TV(\mathbf{W})$. For example, if $\mathcal{T}_N(\text{Bool})$ is the flat cpo on {TRUE, FALSE} and $\mathcal{T}_N(N) = (N_\perp, \sqsubseteq_{N_\perp})$, then \mathcal{T}_N is a complete type assignment for \mathbf{W}_N .

When \mathcal{T} is understood, we will write $(\text{Bool}, \sqsubseteq_{\text{Bool}})$ for $\mathcal{T}(\text{Bool})$ and \perp_{Bool} for the bottom element of $\mathcal{T}(\text{Bool})$. For an arbitrary $\tau \in TV(\mathbf{W})$, we will often write $(D_\tau, \sqsubseteq_\tau)$ for $(D_{\mathcal{T}(\tau)}, \sqsubseteq_{\mathcal{T}(\tau)})$ and \perp_τ for $\perp_{\mathcal{T}(\tau)}$. Also, we will sometimes write

$$(D_{\tau_1} \times \cdots \times \tau_n, \sqsubseteq_{\tau_1} \times \cdots \times \tau_n)$$

for

$$(D_{\mathcal{T}(\tau_1)} \times \cdots \times D_{\mathcal{T}(\tau_n)}, \sqsubseteq_{D_{\mathcal{T}(\tau_1)} \times \cdots \times D_{\mathcal{T}(\tau_n)}}).$$

In particular, $D_{\delta(F)} = D_{\tau_1} \times \cdots \times \tau_n$ if $\tau(F) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$.

It will be useful to define the following notation. For sets D , E and $f: D \rightarrow E$, $\text{ran } f$ is the range of f . Also, if $e \in E$, then $\text{ran } e = \{e\}$. In effect, we are treating e as a function of 0 arguments.

Definition. Let \mathcal{T} be a type assignment for \mathbf{W} . A *\mathcal{T} -interpretation* for \mathbf{W} is a function \mathcal{I} with domain \mathbf{W} that satisfies the following conditions.

1. For all constant symbols $c \in \mathbf{W}_c$ with $\tau(c) = \tau$, $\mathcal{I}(c) \in D_\tau - \{\perp_\tau\}$.

We will write tt for $\mathcal{I}(\text{tt})$ and ff for $\mathcal{I}(\text{ff})$.

2. For all proper function symbols $f \in \mathbf{W}_c$ with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$,

a. $\mathcal{I}(f) \in D_{\tau_1} \times \cdots \times D_{\tau_n} \rightarrow D_\tau$;

b. if $\mathcal{I}(f)(d_1, \dots, d_n) = \mathcal{I}(f)(e_1, \dots, e_n) \neq \perp_\tau$, then $(d_1, \dots, d_n) = (e_1, \dots, e_n)$;

c. if $d_i \neq \perp_{\tau_i}$, $1 \leq i \leq n$, then $\mathcal{I}(f)(d_1, \dots, d_n) \neq \perp_\tau$.

3. For all $f, g \in \mathbf{W}_c$ such that $\rho(f) = \rho(g)$, $\text{ran } \mathcal{I}(f)$ and $\text{ran } \mathcal{I}(g)$ can have at most $\perp_{\rho(f)}$ in common; that is,

$$(\text{ran } \mathcal{I}(f) \cap \text{ran } \mathcal{I}(g)) - \{\perp_{\rho(f)}\} = \emptyset.$$

4. For all $\tau \in TV(\mathbf{W})$, $\mathcal{I}(\text{if}_\tau): \text{Bool} \times D_\tau \times D_\tau \rightarrow D_\tau$ is defined

$$\mathcal{I}(\text{if}_\tau)(b, d, e) = \begin{cases} d & \text{if } b = \text{tt} \\ e & \text{if } b = \text{ff} \\ \perp_\tau & \text{if } b = \perp_{\text{Bool}} \end{cases}.$$

5. For all constant symbols $c \in W_c^-$ with $\tau(c) = \tau$, $\mathcal{I}(\text{is_c}): D_\tau \rightarrow \text{Bool}$ is defined

$$\mathcal{I}(\text{is_c})(d) = \begin{cases} \text{tt} & \text{if } d = \mathcal{I}(c) \\ \text{ff} & \text{if } d \neq \perp_\tau \text{ and } d \neq \mathcal{I}(c) \\ \perp_{\text{Bool}} & \text{if } d = \perp_\tau; \end{cases}$$

for all proper function symbols $f \in W_c$ with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, $\mathcal{I}(\text{is_f}): D_\tau \rightarrow \text{Bool}$ is defined

$$\mathcal{I}(\text{is_f})(d) = \begin{cases} \text{tt} & \text{if } d \neq \perp_\tau \text{ and } d \in \text{ran } \mathcal{I}(f) \\ \text{ff} & \text{if } d \neq \perp_\tau \text{ and } d \notin \text{ran } \mathcal{I}(f) \\ \perp_{\text{Bool}} & \text{if } d = \perp_\tau. \end{cases}$$

6. For all proper function symbols $f \in W_c$ with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$ and for all $1 \leq i \leq n$, $\mathcal{I}(f_i^{-1}): D_\tau \rightarrow D_{\tau_i}$ is defined

$$\mathcal{I}(f_i^{-1})(d) = \begin{cases} d_i & \text{if } d \neq \perp_\tau \text{ and } d = \mathcal{I}(f)(d_1, \dots, d_n) \text{ for} \\ & \text{some } (d_1, \dots, d_n) \in D_{\tau_1} \times \cdots \times D_{\tau_n} \\ \perp_{\tau_i} & \text{otherwise.} \end{cases}$$

If $\mathcal{I}(f)$ is continuous for all proper function symbols $f \in W$, then \mathcal{I} is a *continuous \mathcal{T} -interpretation*. A **W -structure** is a pair $\Sigma = (\mathcal{T}, \mathcal{I})$, where \mathcal{T} is a type assignment for W and \mathcal{I} is a \mathcal{T} -interpretation for W . Σ is a *complete W -structure* if \mathcal{T} is a complete type assignment, and it is a *continuous W -structure* if \mathcal{I} is a continuous \mathcal{T} -interpretation.

A \mathcal{T} -interpretation for W gives a meaning to each symbol in W , using the objects made available by \mathcal{T} in the sets D_τ . Conditions 4, 5, and 6 require a specific interpretation for the built-in function symbols. Note in particular that $\mathcal{I}(f_i^{-1})$ is a well-defined function because of condition 2b. Conditions 2c and 3 are imposed to make certain information about the objects of a W -structure available at the syntactic level of W -terms. Condition 2c implies that the meaning of a ground term is never the bottom element, so that it makes sense, for example, to replace $s_1^{-1}(s(0))$ with 0 , since $\mathcal{I}(s)(\mathcal{I}(0)) \neq \perp_N$. Condition 3 implies that it makes sense to replace a term such as $\text{is_f}(g(c))$ with ff . As we will see in the next chapter, the replacement of terms by equivalent terms is the basis of the operational semantics of recursion equations, so these conditions are included to make the operational semantics work correctly.

For an example, let \mathbf{W}_N and \mathcal{T}_N be as before. If

$$\mathcal{I}_N(\text{tt}) = \text{TRUE}$$

$$\mathcal{I}_N(\text{ff}) = \text{FALSE}$$

$$\mathcal{I}_N(0) = 0$$

$$\mathcal{I}_N(s)(m) = \begin{cases} \perp_N & \text{if } m = \perp_N \\ m + 1 & \text{otherwise} \end{cases}$$

$$\mathcal{I}_N(\text{if}_{\text{Bool}})(b, d, e) = \begin{cases} d & \text{if } b = \text{TRUE} \\ e & \text{if } b = \text{FALSE} \\ \perp_{\text{Bool}} & \text{if } b = \perp_{\text{Bool}} \end{cases}$$

$$\mathcal{I}_N(\text{if}_N)(b, d, e) = \begin{cases} d & \text{if } b = \text{TRUE} \\ e & \text{if } b = \text{FALSE} \\ \perp_N & \text{if } b = \perp_{\text{Bool}} \end{cases}$$

$$\mathcal{I}_N(\text{is_0})(d) = \begin{cases} \text{TRUE} & \text{if } d = 0 \\ \text{FALSE} & \text{if } d \neq \perp_N \text{ and } d > 0 \\ \perp_{\text{Bool}} & \text{if } d = \perp_N \end{cases}$$

$$\mathcal{I}_N(\text{is_s})(d) = \begin{cases} \text{TRUE} & \text{if } d \neq \perp_N \text{ and } d > 0 \\ \text{FALSE} & \text{if } d = 0 \\ \perp_{\text{Bool}} & \text{if } d = \perp_N \end{cases}$$

$$\mathcal{I}_N(s_1^{-1})(d) = \begin{cases} d - 1 & \text{if } d \neq \perp_N \text{ and } d > 0 \\ \perp_N & \text{otherwise,} \end{cases}$$

then \mathcal{I}_N is a \mathcal{T}_N -assignment for \mathbf{W}_N . We will write Σ_N for $(\mathcal{T}_N, \mathcal{I}_N)$. It is easy to check that Σ_N is complete and continuous.

We now have a way of interpreting the symbols of \mathbf{W} , but before we can give a meaning to arbitrary terms, we need a way of interpreting variables.

Definition. Let \mathcal{T} be a type assignment for \mathbf{W} and V a set of variables. A *variable assignment* for V based on \mathcal{T} is a function α with domain V such that

1. $\alpha(X) \in D_\tau$ for each individual variable $X \in V$ with $\tau(X) = \tau$, and
2. $\alpha(F) \in D_{\tau_1} \times \cdots \times D_{\tau_n} \rightarrow D_\tau$ for each function variable $F \in V$ with $\tau(F) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$.

α is a *continuous variable assignment* for V if $\alpha(F)$ is continuous for each function variable $F \in V$. $\mathcal{A}_{\mathcal{T}}(V)$ is the set of all variable assignments for V

based on \mathcal{T} , and $\mathcal{CA}_{\mathcal{T}}(V)$ is the set of all continuous variable assignments for V based on \mathcal{T} .

Let $(\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure and V a set of variables. For any $\alpha \in \mathcal{A}_{\mathcal{T}}(V)$, we extend α to a function $\bar{\alpha}_{\mathcal{I}}$ with domain $\text{TM}_{\mathbf{W}}(V)$ as follows:

$$\bar{\alpha}_{\mathcal{I}}(\mathbf{c}) = \mathcal{I}(\mathbf{c}) \quad \text{for all constant symbols } \mathbf{c} \in \mathbf{W}$$

$$\bar{\alpha}_{\mathcal{I}}(\mathbf{X}) = \alpha(\mathbf{X}) \quad \text{for all individual variables } \mathbf{X} \in V$$

$$\bar{\alpha}_{\mathcal{I}}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) = \mathcal{I}(\mathbf{f})(\bar{\alpha}_{\mathcal{I}}(\mathbf{t}_1), \dots, \bar{\alpha}_{\mathcal{I}}(\mathbf{t}_n)) \quad \text{where } \mathbf{f} \in \mathbf{W}$$

$$\bar{\alpha}_{\mathcal{I}}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) = \alpha(\mathbf{F})(\bar{\alpha}_{\mathcal{I}}(\mathbf{t}_1), \dots, \bar{\alpha}_{\mathcal{I}}(\mathbf{t}_n)) \quad \text{where } \mathbf{F} \in V.$$

$\bar{\alpha}_{\mathcal{I}}$ is a function we can use to assign a meaning to any term in $\text{TM}_{\mathbf{W}}(V)$. Note that \emptyset is the unique assignment in $\mathcal{A}_{\mathcal{T}}(\emptyset)$, and if $\mathbf{t} \in \text{TM}_{\mathbf{W}}$, i.e., \mathbf{t} contains no variables, then $\bar{\emptyset}_{\mathcal{I}}$ is sufficient for interpreting \mathbf{t} . When \mathcal{I} is understood, we will often write $\bar{\alpha}$ for $\bar{\alpha}_{\mathcal{I}}$.

For example, let $V = \{\mathbf{X}, \mathbf{Y}, \mathbf{F}\}$, and let

$$\alpha(\mathbf{X}) = 3, \quad \alpha(\mathbf{Y}) = 5, \quad \alpha(\mathbf{F}) = +_{\perp},$$

where $+_{\perp}$ is the strict extension of $+$. Then $\alpha \in \mathcal{A}_{\mathcal{I}_N}(V)$, and

$$\begin{aligned} \bar{\alpha}_{\mathcal{I}_N}(\mathbf{s}(\mathbf{F}(\mathbf{X}, \mathbf{s}(\mathbf{Y})))) &= \mathcal{I}_N(\mathbf{s})(\bar{\alpha}_{\mathcal{I}_N}(\mathbf{F}(\mathbf{X}, \mathbf{s}(\mathbf{Y})))) \\ &= \mathcal{I}_N(\mathbf{s})(\alpha(\mathbf{F})(\bar{\alpha}_{\mathcal{I}_N}(\mathbf{X}), \bar{\alpha}_{\mathcal{I}_N}(\mathbf{s}(\mathbf{Y})))) \\ &= \mathcal{I}_N(\mathbf{s})(\alpha(\mathbf{F})(\alpha(\mathbf{X}), \mathcal{I}_N(\mathbf{s})(\bar{\alpha}_{\mathcal{I}_N}(\mathbf{Y})))) \\ &= \mathcal{I}_N(\mathbf{s})(\alpha(\mathbf{F})(\alpha(\mathbf{X}), \mathcal{I}_N(\mathbf{s})(\alpha(\mathbf{Y})))) \\ &= \mathcal{I}_N(\mathbf{s})(\alpha(\mathbf{F})(3, \mathcal{I}_N(\mathbf{s})(5))) \\ &= \mathcal{I}_N(\mathbf{s})(\alpha(\mathbf{F})(3, 6)) \\ &= \mathcal{I}_N(\mathbf{s})(9) \\ &= 10. \end{aligned}$$

The next theorem shows that $\bar{\alpha}_{\mathcal{I}}(\mathbf{t})$ assigns a value to term \mathbf{t} in the appropriate set, namely, $D_{\tau(\mathbf{t})}$. We need it to verify that the definition of $\bar{\alpha}_{\mathcal{I}}$ makes sense. For example, when we define

$$\bar{\alpha}_{\mathcal{I}}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) = \mathcal{I}(\mathbf{f})(\bar{\alpha}_{\mathcal{I}}(\mathbf{t}_1), \dots, \bar{\alpha}_{\mathcal{I}}(\mathbf{t}_n)),$$

where $\tau(\mathbf{f}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, we have $\mathcal{I}(\mathbf{f}) \in D_{\tau_1} \times \dots \times D_{\tau_n} \rightarrow D_{\tau}$, so we want to know that $\bar{\alpha}_{\mathcal{I}}(\mathbf{t}_i) \in D_{\tau_i}$, $1 \leq i \leq n$.

Theorem 2.1. Let $(\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure, V a set of variables, $\alpha \in \mathcal{A}_{\mathcal{I}}(V)$, and $\mathbf{t} \in \text{TM}_{\mathbf{W}}^{\tau}(V)$ for some $\tau \in TV(\mathbf{W})$. Then $\bar{\alpha}_{\mathcal{I}}(\mathbf{t}) \in D_{\tau}$.

Proof. We argue by structural induction on \mathbf{t} . If \mathbf{t} is a constant symbol $\mathbf{c} \in \mathbf{W}$, then $\bar{\alpha}_{\mathcal{I}}(\mathbf{c}) = \mathcal{I}(\mathbf{c}) \in D_{\tau}$, and if \mathbf{t} is an individual variable $\mathbf{X} \in V$, then $\bar{\alpha}_{\mathcal{I}}(\mathbf{X}) = \alpha(\mathbf{X}) \in D_{\tau}$. If \mathbf{t} is $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)$, where $\mathbf{f} \in \mathbf{W}$, $\tau(\mathbf{f}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, and $\mathbf{t}_i \in \text{TM}_{\mathbf{W}}^{\tau_i}(V)$, $1 \leq i \leq n$, then $\bar{\alpha}_{\mathcal{I}}(\mathbf{t}_i) \in D_{\tau_i}$, $1 \leq i \leq n$, by the induction hypothesis, and $\mathcal{I}(\mathbf{f}) \in D_{\tau_1} \times \dots \times D_{\tau_n} \rightarrow D_{\tau}$, so

$$\bar{\alpha}_{\mathcal{I}}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) = \mathcal{I}(\mathbf{f})(\bar{\alpha}_{\mathcal{I}}(\mathbf{t}_1), \dots, \bar{\alpha}_{\mathcal{I}}(\mathbf{t}_n)) \in D_{\tau}.$$

Similarly, if \mathbf{t} is $\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)$, where $\mathbf{F} \in V$, $\tau(\mathbf{F}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, and $\mathbf{t}_i \in \text{TM}_{\mathbf{W}}^{\tau_i}(V)$, $1 \leq i \leq n$, then $\alpha(\mathbf{F}) \in D_{\tau_1} \times \dots \times D_{\tau_n} \rightarrow D_{\tau}$ and

$$\bar{\alpha}_{\mathcal{I}}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) = \alpha(\mathbf{F})(\bar{\alpha}_{\mathcal{I}}(\mathbf{t}_1), \dots, \bar{\alpha}_{\mathcal{I}}(\mathbf{t}_n)) \in D_{\tau}. \quad \blacksquare$$

Let \mathcal{T} be a complete type assignment for \mathbf{W} and V a set of variables. Then for each individual variable $\mathbf{X} \in V$ with $\tau(\mathbf{X}) = \tau$, $(D_{\tau}, \sqsubseteq_{\tau})$ is a cpo by assumption, and for each function variable $\mathbf{F} \in V$ with $\tau(\mathbf{F}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$,

$$([D_{\tau_1} \times \dots \times D_{\tau_n} \rightarrow D_{\tau}], \sqsubseteq_{[D_{\tau_1} \times \dots \times D_{\tau_n} \rightarrow D_{\tau}]})$$

is a cpo by Theorems³ 16.3.6 and 16.4.4. Now, let \mathcal{D}_V be the function with domain V such that

$$\mathcal{D}_V(\mathbf{X}) = (D_{\tau}, \sqsubseteq_{\tau}) \quad \text{for each } \mathbf{X} \in V \text{ with } \tau(\mathbf{X}) = \tau$$

$$\mathcal{D}_V(\mathbf{F}) = ([D_{\tau_1} \times \dots \times D_{\tau_n} \rightarrow D_{\tau}], \sqsubseteq_{[D_{\tau_1} \times \dots \times D_{\tau_n} \rightarrow D_{\tau}]})$$

for each $\mathbf{F} \in V$ with $\tau(\mathbf{F}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$.

Then a continuous variable assignment for V based on \mathcal{T} is a \mathcal{D}_V -choice function and vice versa, so $\mathcal{CA}_{\mathcal{T}}(V) = \text{ch}(\mathcal{D}_V)$, and $(\text{ch}(\mathcal{D}_V), \sqsubseteq_{\text{ch}(\mathcal{D}_V)})$ is a cpo by Theorem 16.3.10. Writing $\sqsubseteq_{\mathcal{CA}_{\mathcal{T}}(V)}$ for $\sqsubseteq_{\text{ch}(\mathcal{D}_V)}$, we have proved

Theorem 2.2. Let \mathcal{T} be a complete type assignment for \mathbf{W} and V a set of variables. Then $(\mathcal{CA}_{\mathcal{T}}(V), \sqsubseteq_{\mathcal{CA}_{\mathcal{T}}(V)})$ is a cpo.

Note that the bottom element of $(\mathcal{CA}_{\mathcal{T}}(V), \sqsubseteq_{\mathcal{CA}_{\mathcal{T}}(V)})$, which we will

³ We will refer to theorems in Chapter 16 frequently here, so we adopt the convention of writing Theorem 16.3.6, for example, to refer to Theorem 3.6 in Chapter 16.

write $\Omega_{\mathcal{CA}_{\mathcal{T}}(V)}$, or simply Ω when \mathcal{T} and V are understood, satisfies

$$\Omega(\mathbf{X}) = \perp_{\tau(\mathbf{X})} \quad \text{for each individual variable } \mathbf{X} \in V$$

$$\begin{aligned} \Omega(\mathbf{F})(d_1, \dots, d_n) &= \perp_{\rho(\mathbf{F})} \quad \text{for each function variable } \mathbf{F} \in V \text{ and all} \\ (d_1, \dots, d_n) &\in D_{\delta(\mathbf{F})}. \end{aligned}$$

Note that if $V = \emptyset$, then $\Omega_{\mathcal{CA}_{\mathcal{T}}(V)} = \emptyset$. Given a ground term $\mathbf{t} \in \text{TM}_W$, we will generally write $\bar{\Omega}(\mathbf{t})$ to interpret \mathbf{t} .

The next theorem says, in effect, that the function that extends assignments α to $\bar{\alpha}_{\mathcal{T}}$ is monotonic and continuous.

Theorem 2.3. Let $(\mathcal{T}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, V a set of variables, and $\mathbf{t} \in \text{TM}_W^{\tau}(V)$ for some $\tau \in TV(\mathbf{W})$.

1. For $\alpha, \beta \in \mathcal{CA}_{\mathcal{T}}(V)$, $\alpha \sqsubseteq_{\mathcal{CA}_{\mathcal{T}}(V)} \beta$ implies $\bar{\alpha}_{\mathcal{T}}(\mathbf{t}) \sqsubseteq_{\tau} \bar{\beta}_{\mathcal{T}}(\mathbf{t})$.
2. For a chain \mathcal{A} in $(\mathcal{CA}_{\mathcal{T}}(V), \sqsubseteq_{\mathcal{CA}_{\mathcal{T}}(V)})$, $\overline{\sqcup \mathcal{A}}(\mathbf{t}) = \sqcup \{\bar{\alpha}_{\mathcal{T}}(\mathbf{t}) \mid \alpha \in \mathcal{A}\}$.

Proof. Both parts can be proven by structural induction on \mathbf{t} , and part 1 is straightforward, so we leave it as an exercise and concentrate on part 2. Let \mathcal{A} be a chain in $(\mathcal{CA}_{\mathcal{T}}(V), \sqsubseteq_{\mathcal{CA}_{\mathcal{T}}(V)})$. Then $\sqcup \mathcal{A}$ exists by Theorem 2.2. If \mathbf{t} is a constant symbol $c \in \mathbf{W}$, then

$$\overline{\sqcup \mathcal{A}}(\mathbf{c}) = \mathcal{I}(\mathbf{c}) = \sqcup \{\mathcal{I}(\mathbf{c}) \mid \alpha \in \mathcal{A}\} = \sqcup \{\bar{\alpha}(\mathbf{c}) \mid \alpha \in \mathcal{A}\},$$

and if \mathbf{t} is an individual variable $\mathbf{X} \in V$, then

$$\begin{aligned} \overline{\sqcup \mathcal{A}}(\mathbf{X}) &= (\sqcup \mathcal{A})(\mathbf{X}) \\ &= \sqcup \{\alpha(\mathbf{X}) \mid \alpha \in \mathcal{A}\} \quad \text{by Theorem 16.3.9} \\ &= \sqcup \{\bar{\alpha}(\mathbf{X}) \mid \alpha \in \mathcal{A}\}. \end{aligned}$$

If \mathbf{t} is $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)$, where $\mathbf{f} \in \mathbf{W}$, then

$$\begin{aligned} \overline{\sqcup \mathcal{A}}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) &= \mathcal{I}(\mathbf{f})(\overline{\sqcup \mathcal{A}}(\mathbf{t}_1), \dots, \overline{\sqcup \mathcal{A}}(\mathbf{t}_n)) \\ &= \mathcal{I}(\mathbf{f})(\sqcup \{\bar{\alpha}(\mathbf{t}_1) \mid \alpha \in \mathcal{A}\}, \dots, \sqcup \{\bar{\alpha}(\mathbf{t}_n) \mid \alpha \in \mathcal{A}\}) \quad \text{by the induction hypothesis} \\ &= \mathcal{I}(\mathbf{f})(\sqcup \{(\bar{\alpha}(\mathbf{t}_1), \dots, \bar{\alpha}(\mathbf{t}_n)) \mid \alpha \in \mathcal{A}\}) \quad \text{by Theorem 16.3.5} \\ &= \sqcup \{\mathcal{I}(\mathbf{f})(\bar{\alpha}(\mathbf{t}_1), \dots, \bar{\alpha}(\mathbf{t}_n)) \mid \alpha \in \mathcal{A}\} \quad \text{since } \mathcal{I}(\mathbf{f}) \text{ is continuous and} \\ &\quad \{(\bar{\alpha}(\mathbf{t}_1), \dots, \bar{\alpha}(\mathbf{t}_n)) \mid \alpha \in \mathcal{A}\} \text{ is a chain by part 1} \\ &= \sqcup \{\bar{\alpha}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \mid \alpha \in \mathcal{A}\}. \end{aligned}$$

Finally, let \mathbf{t} be $\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)$, where \mathbf{F} is a function variable in V , and let $\Gamma: \mathcal{CA}_{\mathcal{T}}(V) \times \mathcal{CA}_{\mathcal{T}}(V) \rightarrow D_{\tau}$ be defined $\Gamma(\alpha, \beta) = \alpha(\mathbf{F})(\bar{\beta}(\mathbf{t}_1), \dots, \bar{\beta}(\mathbf{t}_n))$. Then Γ is monotonic by part 1 and by the monotonicity of $\alpha(\mathbf{F})$ for all

$\alpha \in \mathcal{A}$, so

$$\begin{aligned}
& \overline{\sqcup \mathcal{A}}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \\
&= (\sqcup \mathcal{A})(\mathbf{F})(\overline{\sqcup \mathcal{A}}(\mathbf{t}_1), \dots, \overline{\sqcup \mathcal{A}}(\mathbf{t}_n)) \\
&= (\sqcup \{\alpha(\mathbf{F}) \mid \alpha \in \mathcal{A}\})(\overline{\sqcup \mathcal{A}}(\mathbf{t}_1), \dots, \overline{\sqcup \mathcal{A}}(\mathbf{t}_n)) \quad \text{by Theorem 16.3.9} \\
&= \sqcup \{\alpha(\mathbf{F})(\overline{\sqcup \mathcal{A}}(\mathbf{t}_1), \dots, \overline{\sqcup \mathcal{A}}(\mathbf{t}_n)) \mid \alpha \in \mathcal{A}\} \quad \text{by Theorem 16.3.7} \\
&= \sqcup \{\alpha(\mathbf{F})(\sqcup \{\bar{\beta}(\mathbf{t}_1) \mid \beta \in \mathcal{A}\}, \dots, \sqcup \{\bar{\beta}(\mathbf{t}_n) \mid \beta \in \mathcal{A}\}) \mid \alpha \in \mathcal{A}\} \quad \text{by} \\
&\qquad \text{the induction hypothesis} \\
&= \sqcup \{\alpha(\mathbf{F})(\sqcup \{(\bar{\beta}(\mathbf{t}_1), \dots, \bar{\beta}(\mathbf{t}_n)) \mid \beta \in \mathcal{A}\}) \mid \alpha \in \mathcal{A}\} \quad \text{by Theorem} \\
&\qquad 16.3.5 \\
&= \sqcup \{\sqcup \{\alpha(\mathbf{F})(\bar{\beta}(\mathbf{t}_1), \dots, \bar{\beta}(\mathbf{t}_n)) \mid \beta \in \mathcal{A}\} \mid \alpha \in \mathcal{A}\} \quad \text{since } \alpha(\mathbf{F}) \text{ is con-} \\
&\qquad \text{tinuous for each } \alpha \in \mathcal{A} \text{ and } \{(\bar{\beta}(\mathbf{t}_1), \dots, \bar{\beta}(\mathbf{t}_n)) \mid \beta \in \mathcal{A}\} \\
&\qquad \text{is a chain by part 1} \\
&= \sqcup \{\sqcup \{\Gamma(\alpha, \beta) \mid \beta \in \mathcal{A}\} \mid \alpha \in \mathcal{A}\} \\
&= \sqcup \{\Gamma(\alpha, \alpha) \mid \alpha \in \mathcal{A}\} \quad \text{by the diagonal lemma} \\
&= \sqcup \{\alpha(\mathbf{F})(\bar{\alpha}(\mathbf{t}_1), \dots, \bar{\alpha}(\mathbf{t}_n)) \mid \alpha \in \mathcal{A}\} \\
&= \sqcup \{\bar{\alpha}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \mid \alpha \in \mathcal{A}\}. \quad \blacksquare
\end{aligned}$$

We also prove one more result about variable assignments that we will use in the next section.

Coincidence Lemma. Let $(\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure, let V_1, V_2 be sets of variables, let $\alpha \in \mathcal{A}_{\mathcal{T}}(V_1)$ and $\beta \in \mathcal{A}_{\mathcal{T}}(V_2)$, and let

$$V = \{\mathbf{V} \in V_1 \cap V_2 \mid \alpha(\mathbf{V}) = \beta(\mathbf{V})\}.$$

Then for all $\mathbf{t} \in \text{TM}_{\mathbf{W}}(V)$, $\bar{\alpha}(\mathbf{t}) = \bar{\beta}(\mathbf{t})$.

Proof. We argue by structural induction on \mathbf{t} . If \mathbf{t} is an individual variable $\mathbf{X} \in V$, then $\bar{\alpha}(\mathbf{X}) = \alpha(\mathbf{X}) = \beta(\mathbf{X}) = \bar{\beta}(\mathbf{X})$. If \mathbf{t} is a constant symbol $\mathbf{c} \in \mathbf{W}$, then $\bar{\alpha}(\mathbf{c}) = \mathcal{I}(\mathbf{c}) = \bar{\beta}(\mathbf{c})$. If \mathbf{t} is $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)$, where $\mathbf{f} \in \mathbf{W}$, then

$$\begin{aligned}
\bar{\alpha}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) &= \mathcal{I}(\mathbf{f})(\bar{\alpha}(\mathbf{t}_1), \dots, \bar{\alpha}(\mathbf{t}_n)) \\
&= \mathcal{I}(\mathbf{f})(\bar{\beta}(\mathbf{t}_1), \dots, \bar{\beta}(\mathbf{t}_n)) \quad \text{by the induction hypothesis} \\
&= \bar{\beta}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)),
\end{aligned}$$

and if \mathbf{t} is $\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)$, where $\mathbf{F} \in V$, then

$$\begin{aligned}
\bar{\alpha}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) &= \alpha(\mathbf{F})(\bar{\alpha}(\mathbf{t}_1), \dots, \bar{\alpha}(\mathbf{t}_n)) \\
&= \alpha(\mathbf{F})(\bar{\beta}(\mathbf{t}_1), \dots, \bar{\beta}(\mathbf{t}_n)) \quad \text{by the induction hypothesis} \\
&= \beta(\mathbf{F})(\bar{\beta}(\mathbf{t}_1), \dots, \bar{\beta}(\mathbf{t}_n)) \quad \text{since } \mathbf{F} \in V \\
&= \bar{\beta}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)). \quad \blacksquare
\end{aligned}$$

Exercises

1. Show that Σ_N is a complete, continuous \mathbf{W}_N -structure.
2. Let $\mathcal{T}(\mathbf{Bool}) = (\{\perp, 0, 1\}, \{(\perp, \perp), (\perp, 0), (\perp, 1), (0, 0), (0, 1)\})$, $\mathcal{T}(N) = (N_\perp, \sqsubseteq_{N_\perp})$, $\mathcal{T}(\mathbf{tt}) = 0$, and $\mathcal{T}(\mathbf{ff}) = 1$. Extend \mathcal{T} to a \mathcal{F} -interpretation for \mathbf{W}_N .
3. Let \mathcal{T} be a type assignment for \mathbf{W}_{NL} with $\mathcal{T}(N) = \mathcal{T}(NL) = (N_\perp, \sqsubseteq_{N_\perp})$.
 - (a) Let $\mathcal{T}(\mathbf{nil}) = 0$ and $\mathcal{T}(\mathbf{cons}) = +_\perp$ (the strict extension of $+$). Show that \mathcal{T} cannot be extended to a \mathcal{F} -interpretation for \mathbf{W}_{NL} .
 - (b) Give a continuous \mathcal{F} -interpretation \mathcal{T}' for \mathbf{W}_{NL} . [Hint: Consider the pairing function $\langle x, y \rangle$ from Chapter 3.] What is $\bar{\mathcal{Q}}_{\mathcal{T}'}(\mathbf{cons}(\mathbf{s}(0), \mathbf{nil}))$?
4. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure such that $\mathcal{T}(\tau)$ is a flat cpo for all $\tau \in TV(\mathbf{W})$. Show that for every built-in function symbol $\mathbf{f} \in \mathbf{W}$, $\mathcal{I}(\mathbf{f})$ is continuous.
5. Let $\alpha(X) = 3$, $\alpha(Y) = 2$, $\alpha(F) = +_\perp$, and $\alpha(G) = \cdot_\perp$ (the strict extension of the multiplication function). Calculate $\bar{\alpha}_{\mathcal{T}_N}(t)$, where t is as follows.
 - (a) $F(s(X), G(s(X), F(X, Y)))$.
 - (b) $s_1^{-1}(F(s_1^{-1}(s(X)), Y))$.
 - (c) $\mathbf{if}_N(\mathbf{is_0}(G(X, s_1^{-1}(s_1^{-1}(0)))), X, Y)$.
 - (d) $\mathbf{if}_{\mathbf{Bool}}(\mathbf{is_s}(X), \mathbf{is_0}(X), \mathbf{is_0}(X))$.
6. Let $\mathcal{I}_e(0) = 0$ and $\mathcal{I}_e(s) = e_\perp$, where $e(n) = 2^n$ for all $n \in N$.
 - (a) Extend \mathcal{I}_e to a \mathcal{T}_N -interpretation for \mathbf{W}_N .
 - (b) Calculate $\bar{\alpha}_{\mathcal{T}_e}(t)$ for each term t given in Exercise 5.
7. Let $\mathcal{T}(\mathbf{Bool}) = \mathcal{T}_N(\mathbf{Bool})$, $\mathcal{T}(N) = \mathcal{T}_N(N)$, and $\mathcal{T}(NL) = (TUP_\perp, \sqsubseteq_{TUP_\perp})$, where TUP is the set of all tuples of natural numbers and $(TUP_\perp, \sqsubseteq_{TUP_\perp})$ is the flat cpo on TUP . Give a \mathcal{F} -interpretation for \mathbf{W}_{NL} .
8. Let $\mathcal{T}(\mathbf{Bool}) = \mathcal{T}_N(\mathbf{Bool})$, $\mathcal{T}(N) = \mathcal{T}_N(N)$, and $\mathcal{T}(NL) = (\mathcal{P}_f(N), \subseteq_{\mathcal{P}_f(N)})$, where $\mathcal{P}_f(N)$ consists of all the finite subsets of N , and let $\mathcal{T}(\mathbf{cons})(e, \{d_1, \dots, d_n\}) = \{e, d_1, \dots, d_n\}$. Explain why \mathcal{T} cannot be extended to a \mathcal{F} -interpretation for \mathbf{W}_{NL} .
9. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a continuous \mathbf{W} -structure, and let $\mathbf{f} \in \mathbf{W}_c$. Show that for all $d, e \in D_{\rho(\mathbf{f})}$ such that $\perp_{\rho(\mathbf{f})} \neq d \sqsubseteq_{\rho(\mathbf{f})} e$, if $d \in \text{ran } \mathcal{I}(\mathbf{f})$ then $e \in \text{ran } \mathcal{I}(\mathbf{f})$. [Hint: Use $\mathbf{is_f}$.]
10. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a continuous \mathbf{W} -structure, and let $\mathbf{f} \in \mathbf{W}_c$. Show that for all $d, e \in D_{\delta(\mathbf{f})}$, if $\perp_{\rho(\mathbf{f})} \neq \mathcal{I}(\mathbf{f})(d) \sqsubseteq_{\rho(\mathbf{f})} \mathcal{I}(\mathbf{f})(e)$, then $d \sqsubseteq_{\delta(\mathbf{f})} e$. [Hint: Use \mathbf{f}_i^{-1} , $1 \leq i \leq \text{ar}(\mathbf{f})$.]

11. Let $(\mathcal{T}, \mathcal{I}) = \Sigma_N$, and let $X \in \text{VAR}_1^N$. Give a term $t \in \text{TM}_{W_N}(\{X\})$ and $\alpha, \beta \in \mathcal{C}\mathcal{A}_{\mathcal{T}}(\{X\})$ such that $\bar{\alpha}_{\mathcal{T}}(t) \sqsubseteq_{N_1} \bar{\beta}_{\mathcal{T}}(t)$ and $\beta \sqsubset_{\mathcal{C}\mathcal{A}_{\mathcal{T}}(\{X\})} \alpha$.
12. Let $(\mathcal{T}, \mathcal{I})$ be a **W**-structure, let V be a set of variables, and let $\alpha, \beta \in \mathcal{C}\mathcal{A}_{\mathcal{T}}(V)$. Show that if $\bar{\alpha}_{\mathcal{T}}(t) \sqsubseteq_{\tau(t)} \bar{\alpha}_{\mathcal{T}}(t)$ for all $t \in \text{TM}_W(V)$, then $\alpha \sqsubseteq_{\mathcal{C}\mathcal{A}_{\mathcal{T}}(V)} \beta$.
13. Prove part 1 of Theorem 2.3.
14. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous **W**-structure, and let V be a set of variables.
 - (a) Define a function \mathcal{D}'_V such that $\mathcal{A}_{\mathcal{T}}(V) = \text{ch}(\mathcal{D}'_V)$, and show that $(\mathcal{A}_{\mathcal{T}}(V), \sqsubseteq_{\mathcal{A}_{\mathcal{T}}(V)})$ is a cpo.
 - (b) Show that part 1 of Theorem 2.3 holds for $(\mathcal{A}_{\mathcal{T}}(V), \sqsubseteq_{\mathcal{A}_{\mathcal{T}}(V)})$.
 - (c) Show that part 2 of Theorem 2.3 fails for $(\mathcal{A}_{\mathcal{T}}(V), \sqsubseteq_{\mathcal{A}_{\mathcal{T}}(V)})$.

3. Solutions to W-Programs

Now that we have the tools for giving a meaning to terms, we can take the first step toward defining the denotational semantics of programs. Let

$$\mathbf{P} = \{F_1(X_1, \dots, X_{n_1}) = t_1, \dots, F_m(X_1, \dots, X_{n_m}) = t_m\}$$

be a **W**-program, and let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a **W**-structure. We want to define the meaning of F_i in terms of t_i , $1 \leq i \leq m$. The idea is that we start with a variable assignment $\alpha \in \mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$ which gives a meaning to each function variable in t_i . Then for any possible input $(d_1, \dots, d_{n_i}) \in D_{\delta(F_i)}$ we extend α with the assignment $\beta = \{(X_1, d_1), \dots, (X_{n_i}, d_{n_i})\}$ and use $\alpha \cup \beta$ to interpret t_i , giving us an output value for input (d_1, \dots, d_{n_i}) .

It will be convenient to introduce a special notation for the assignment β in the previous paragraph. Given an equation $F(X_1, \dots, X_n) = t$ and $d = (d_1, \dots, d_n) \in D_{\delta(F)}$, the variable assignment $\alpha_{(d_1, \dots, d_n)}$, also written α_d , is $\{(X_1, d_1), \dots, (X_n, d_n)\}$; that is,

$$\alpha_{(d_1, \dots, d_n)}(X_i) = d_i, \quad 1 \leq i \leq n.$$

The particular equation that determines the variables in the domain of α_d will always be clear from the context in which α_d is used.

Definition. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a **W**-structure and \mathbf{P} a **W**-program. We associate with \mathbf{P} the higher order function $\Phi_{\mathbf{P}}^{\Sigma}: \mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P})) \rightarrow \mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$, defined as follows. For each $F \in FV(\mathbf{P})$, with defining equation

$\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{t}$, and for all $(d_1, \dots, d_n) \in D_{\delta(\mathbf{F})}$,

$$\Phi_P^\Sigma(\alpha)(\mathbf{F})(d_1, \dots, d_n) = \overline{(\alpha \cup \alpha_{(d_1, \dots, d_n)})}_{\mathcal{J}}(\mathbf{t}).$$

When Σ is understood we will write Φ_P for Φ_P^Σ .

It is clear from the definition that $\Phi_P^\Sigma \in \mathcal{CA}_{\mathcal{J}}(FV(\mathbf{P})) \rightarrow \mathcal{A}_{\mathcal{J}}(FV(\mathbf{P}))$, but if Σ is complete and continuous we can prove something stronger.

Theorem 3.1. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous W-structure and let \mathbf{P} be a W-program. Then $\Phi_P^\Sigma \in \mathcal{CA}_{\mathcal{J}}(FV(\mathbf{P})) \rightarrow \mathcal{CA}_{\mathcal{J}}(FV(\mathbf{P}))$.

Proof. We will write Φ_P for Φ_P^Σ . Let $\alpha \in \mathcal{CA}_{\mathcal{J}}(FV(\mathbf{P}))$. We need to show that $\Phi_P(\alpha)(\mathbf{F}) \in [D_{\delta(\mathbf{F})} \rightarrow D_{\rho(\mathbf{F})}]$ for each $\mathbf{F} \in FV(\mathbf{P})$. It follows from Theorem 2.1 that $\Phi_P(\alpha)(\mathbf{F}) \in D_{\delta(\mathbf{F})} \rightarrow D_{\rho(\mathbf{F})}$, so we just need to show that $\Phi_P(\alpha)(\mathbf{F})$ is continuous. Let $\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{t}$ be an equation in \mathbf{P} with $\tau(\mathbf{F}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, and let C be a chain in $(D_{\tau_1} \times \dots \times \tau_n, \sqsubseteq_{\tau_1 \times \dots \times \tau_n})$. Then $\{\alpha \cup \alpha_c \mid c \in C\}$ is a chain in $(\mathcal{CA}_{\mathcal{J}}(V), \sqsubseteq_{\mathcal{CA}_{\mathcal{J}}(V)})$, where $V = \{\mathbf{X}_1, \dots, \mathbf{X}_n\} \cup FV(\mathbf{P})$, so $\sqcup\{\alpha \cup \alpha_c \mid c \in C\}$ exists by Theorem 2.2. Moreover, for any $\mathbf{G} \in FV(\mathbf{P})$,

$$(\alpha \cup \alpha_{\sqcup C})(\mathbf{G}) = \alpha(\mathbf{G}) = \sqcup\{(\alpha \cup \alpha_c)(\mathbf{G}) \mid c \in C\},$$

and for \mathbf{X}_i , $1 \leq i \leq n$,

$$(\alpha \cup \alpha_{\sqcup C})(\mathbf{X}_i) = (\sqcup C) \downarrow i = \sqcup(C \downarrow i) = \sqcup\{(\alpha \cup \alpha_c)(\mathbf{X}_i) \mid c \in C\},$$

so by Theorem 16.3.9, $\alpha \cup \alpha_{\sqcup C} = \sqcup\{\alpha \cup \alpha_c \mid c \in C\}$. Therefore,

$$\begin{aligned} \Phi_P(\alpha)(\mathbf{F})(\sqcup C) &= \overline{\alpha \cup \alpha_{\sqcup C}}(\mathbf{t}) \\ &= \overline{\sqcup\{\alpha \cup \alpha_c \mid c \in C\}}(\mathbf{t}) \\ &= \sqcup\{\overline{\alpha \cup \alpha_c}(\mathbf{t}) \mid c \in C\} \quad \text{by Theorem 2.3} \\ &= \sqcup\{\Phi_P(\alpha)(\mathbf{F})(c) \mid c \in C\} \\ &= \sqcup\Phi_P(\alpha)(\mathbf{F})(C), \end{aligned}$$

and $\Phi_P(\alpha)(\mathbf{F})$ is continuous. ■

Since a program \mathbf{P} is a set of equations, it makes sense to try to solve these equations to find the meaning of \mathbf{P} .

Definition. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure and \mathbf{P} a \mathbf{W} -program. A *solution* to \mathbf{P} in Σ is any $\alpha \in \mathcal{A}_{\mathcal{I}}(FV(\mathbf{P}))$ such that

$$\alpha(\mathbf{F})(d_1, \dots, d_n) = \overline{\alpha \cup \alpha_{(d_1, \dots, d_n)}}_{\mathcal{I}}(\mathbf{t}) \quad (3.1)$$

for every equation $\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{t}$ in \mathbf{P} and every $(d_1, \dots, d_n) \in D_{\delta(\mathbf{F})}$.

In other words, every function variable in \mathbf{P} is assigned a function such that every equation in \mathbf{P} is satisfied for every possible value taken by the individual variables. Note that an equivalent statement of (3.1) is

$$\overline{\alpha \cup \alpha_{(d_1, \dots, d_n)}}_{\mathcal{I}}(\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n)) = \overline{\alpha \cup \alpha_{(d_1, \dots, d_n)}}_{\mathcal{I}}(\mathbf{t})$$

for every equation $\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{t}$ in \mathbf{P} and every $(d_1, \dots, d_n) \in D_{\delta(\mathbf{F})}$.

It is important to understand that for an arbitrary $\alpha \in \mathcal{A}_{\mathcal{I}}(FV(\mathbf{P}))$, $\Phi_{\mathbf{P}}(\alpha)$ is not necessarily a solution to \mathbf{P} . Consider the \mathbf{W}_N -structure $\Sigma_N = (\mathcal{T}_N, \mathcal{I}_N)$ and the \mathbf{W}_N -program \mathbf{Q} with equations

$$\mathbf{F}(\mathbf{X}) = \mathbf{G}(\mathbf{X})$$

$$\mathbf{G}(\mathbf{X}) = \mathbf{F}(\mathbf{X}).$$

The problem is that $\Phi_{\mathbf{Q}}(\alpha)(\mathbf{F})$ is defined in terms of $\alpha(\mathbf{G})$, but applying $\Phi_{\mathbf{Q}}$ to α changes the function assigned to \mathbf{G} from $\alpha(\mathbf{G})$ to $\Phi_{\mathbf{Q}}(\alpha)(\mathbf{G})$. For example, if $\alpha(\mathbf{F})$ is the constant function of $\alpha(\mathbf{F})(x) = 3$ and $\alpha(\mathbf{G})$ is the constant function $\alpha(\mathbf{G})(x) = 7$, then

$$\Phi_{\mathbf{Q}}(\alpha)(\mathbf{F})(0) = \overline{\alpha \cup \alpha_0}(\mathbf{G}(\mathbf{X})) = \alpha(\mathbf{G})(0) = 7,$$

but

$$\begin{aligned} \overline{\Phi_{\mathbf{Q}}(\alpha) \cup \alpha_0}(\text{rhs}(\mathbf{F})) &= \overline{\Phi_{\mathbf{Q}}(\alpha) \cup \alpha_0}(\mathbf{G}(\mathbf{X})) \\ &= \Phi_{\mathbf{Q}}(\alpha)(\mathbf{G})(0) \\ &= \overline{\alpha \cup \alpha_0}(\mathbf{F}(\mathbf{X})) = \alpha(\mathbf{F})(0) = 3, \end{aligned}$$

so $\Phi_{\mathbf{Q}}(\alpha)$ is not a solution to \mathbf{Q} .

What we need is an α such that $\Phi_{\mathbf{Q}}$ leaves $\alpha(\mathbf{F})$ and $\alpha(\mathbf{G})$ unchanged; that is, we need a fixed point of $\Phi_{\mathbf{Q}}$. If α is some fixed point of $\Phi_{\mathbf{Q}}$, then for any $d \in N_{\perp}$ we have

$$\alpha(\mathbf{F})(d) = \Phi_{\mathbf{Q}}(\alpha)(\mathbf{F})(d) = \overline{\alpha \cup \alpha_d}(\text{rhs}(\mathbf{F}))$$

$$\alpha(\mathbf{G})(d) = \Phi_{\mathbf{Q}}(\alpha)(\mathbf{G})(d) = \overline{\alpha \cup \alpha_d}(\text{rhs}(\mathbf{G})),$$

so α is a solution to \mathbf{Q} . More generally, we can prove

Theorem 3.2. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a W-structure, and let \mathbf{P} be a W-program. Then $\alpha \in \mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$ is a solution to \mathbf{P} in Σ if and only if α is a fixed point of $\Phi_{\mathbf{P}}^{\Sigma}$.

Proof. If $\alpha \in \mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$ is a solution to \mathbf{P} in Σ , then

$$\alpha(\mathbf{F})(d) = \overline{\alpha \cup \alpha_d}(\text{rhs}(\mathbf{F})) = \Phi_{\mathbf{P}}^{\Sigma}(\alpha)(\mathbf{F})(d)$$

holds for all $\mathbf{F} \in FV(\mathbf{P})$ and all $d \in D_{\delta(\mathbf{F})}$, so $\Phi_{\mathbf{P}}^{\Sigma}(\alpha) = \alpha$. On the other hand, if $\alpha \in \mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$ is a fixed point of $\Phi_{\mathbf{P}}^{\Sigma}$, then

$$\alpha(\mathbf{F})(d) = \Phi_{\mathbf{P}}^{\Sigma}(\alpha)(\mathbf{F})(d) = \overline{\alpha \cup \alpha_d}(\text{rhs}(\mathbf{F}))$$

holds for all $\mathbf{F} \in FV(\mathbf{P})$ and all $d \in D_{\delta(\mathbf{F})}$, so α is a solution to \mathbf{P} in Σ . ■

Going back to the example, we still have the problem that Φ_Q has more than one fixed point. Any α that assigns the same function f to both \mathbf{F} and \mathbf{G} , where f could be anything from the everywhere undefined function to (some extension to N_{\perp} of) the total predicate $\text{HALT}(X, X)$, is a solution to \mathbf{Q} . Clearly, there is nothing in program \mathbf{Q} to indicate that the programmer meant to specify a solution to the halting problem. For that matter, there is no indication that the programmer meant to solve any problem at all in writing program \mathbf{Q} . The sensible approach is to focus on the *least* fixed point of Φ_Q , which would be $\alpha(\mathbf{F})(d) = \alpha(\mathbf{G})(d) = \perp_N$ for all $d \in N_{\perp}$, i.e., $\alpha = \Omega$.

Of course, we do not know that $\mu\Phi_{\mathbf{P}}^{\Sigma}$ exists in general, for an arbitrary W-structure Σ and an arbitrary W-program \mathbf{P} .

Theorem 3.3. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous W-structure and let \mathbf{P} be a W-program. Then $\mu\Phi_{\mathbf{P}}^{\Sigma}$ exists, and $\mu\Phi_{\mathbf{P}}^{\Sigma} \in \mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$.

Proof. We will write $\Phi_{\mathbf{P}}$ for $\Phi_{\mathbf{P}}^{\Sigma}$. If $\mu\Phi_{\mathbf{P}}$ exists, then Theorem 3.1 implies that $\mu\Phi_{\mathbf{P}} \in \mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$, so we just need to show that it exists. By Theorem 2.2, it is sufficient to show that

$$\Phi_{\mathbf{P}} \in [\mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P})) \rightarrow \mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))],$$

so by Theorem 3.1 we only need to show that $\Phi_{\mathbf{P}}$ is continuous. Let \mathcal{A} be a chain in $(\mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P})), \sqsubseteq_{\mathcal{C}\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))})$, $\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{t}$ an equation in \mathbf{P} , and $d \in D_{\delta(\mathbf{F})}$. It is easy to see that $\{\alpha \cup \alpha_d \mid \alpha \in \mathcal{A}\}$ is a chain in $(\mathcal{C}\mathcal{A}_{\mathcal{T}}(V), \sqsubseteq_{\mathcal{C}\mathcal{A}_{\mathcal{T}}(V)})$, where $V = \{\mathbf{X}_1, \dots, \mathbf{X}_n\} \cup FV(\mathbf{P})$, and that

$$\sqcup \mathcal{A} \cup \alpha_d = \sqcup \{\alpha \cup \alpha_d \mid \alpha \in \mathcal{A}\}, \quad (3.2)$$

so we have

$$\begin{aligned}
 & \Phi_P(\sqcup \mathcal{A})(F)(d) \\
 &= \overline{\sqcup \mathcal{A} \cup \alpha_d(t)} \\
 &= \overline{\sqcup \{\alpha \cup \alpha_d \mid \alpha \in \mathcal{A}\}}(t) && \text{by (3.2)} \\
 &= \overline{\sqcup \{\alpha \cup \alpha_d\}}(t) \mid \alpha \in \mathcal{A} && \text{by Theorem 2.2} \\
 &= \overline{\sqcup \{\Phi_P(\alpha)(F)(d) \mid \alpha \in \mathcal{A}\}} \\
 &= (\sqcup \{\Phi_P(\alpha)(F) \mid \alpha \in \mathcal{A}\})(d) && \text{by Theorem 16.3.7} \\
 &= (\sqcup \{\Phi_P(\alpha) \mid \alpha \in \mathcal{A}\})(F)(d) && \text{by Theorem 16.3.9} \\
 &= (\sqcup \Phi_P(\mathcal{A}))(F)(d).
 \end{aligned}$$

(Note that F and d were arbitrarily chosen, so $\sqcup \{\Phi_P(\alpha)(F)(d) \mid \alpha \in \mathcal{A}\}$ exists for all $d \in D_{\delta(F)}$ and $\sqcup \{\Phi_P(\alpha)(F) \mid \alpha \in \mathcal{A}\}$ exists for all $F \in FV(P)$, justifying the use of Theorems 16.3.7 and 16.3.9.) Now, $\Phi_P(\sqcup \mathcal{A})(F)(d) = (\sqcup \Phi_P(\mathcal{A}))(F)(d)$ for all $d \in D_{\delta(F)}$ and all $F \in FV(P)$, so $\Phi_P(\sqcup \mathcal{A}) = \sqcup \Phi_P(\mathcal{A})$ and Φ_P is continuous. ■

The fixed point theorem not only tells us that $\mu \Phi_P$ exists, but it also gives us a way of calculating $\mu \Phi_P$, since $\mu \Phi_P = \sqcup \{\Phi_P^i(\Omega) \mid i \in N\}$. For example, let **ADD** be the W_N -program with the equation

$$+ (X, Y) = \text{if}_N(\text{is_0}(Y), X, s(+ (X, s_1^{-1}(Y)))).$$

Then, writing \mathcal{I} for \mathcal{I}_N , in Σ_N we have, for any $d \in N_\perp$ and any $n \in N$,

$$\begin{aligned}
 \Phi_{\text{ADD}}^{n+1}(\Omega)(+)(d, \perp_N) \\
 &= \overline{\Phi_{\text{ADD}}^n(\Omega) \cup \alpha_{(d, \perp_N)}}(\text{if}_N(\text{is_0}(Y), X, s(+ (X, s_1^{-1}(Y))))) \\
 &= \mathcal{I}(\text{if}_N)(\mathcal{I}(\text{is_0})(\perp_N), d, \overline{\Phi_{\text{ADD}}^n(\Omega) \cup \alpha_{(d, \perp_N)}}(s(+ (X, s_1^{-1}(Y))))) \\
 &= \mathcal{I}(\text{if}_N)(\perp_{\text{Bool}}, d, \overline{\Phi_{\text{ADD}}^n(\Omega) \cup \alpha_{(d, \perp_N)}}(s(+ (X, s_1^{-1}(Y))))) \\
 &= \perp_N,
 \end{aligned}$$

and

$$\begin{aligned}
 \Phi_{\text{ADD}}^{n+1}(\Omega)(+)(d, 0) \\
 &= \overline{\Phi_{\text{ADD}}^n(\Omega) \cup \alpha_{(d, 0)}}(\text{if}_N(\text{is_0}(Y), X, s(+ (X, s_1^{-1}(Y))))) \\
 &= \mathcal{I}(\text{if}_N)(\mathcal{I}(\text{is_0})(0), d, \overline{\Phi_{\text{ADD}}^n(\Omega) \cup \alpha_{(d, 0)}}(s(+ (X, s_1^{-1}(Y))))) \\
 &= \mathcal{I}(\text{if}_N)(\text{tt}, d, \overline{\Phi_{\text{ADD}}^n(\Omega) \cup \alpha_{(d, 0)}}(s(+ (X, s_1^{-1}(Y))))) \\
 &= d,
 \end{aligned}$$

so $\mu\Phi_{\text{ADD}}(+)(d, \perp_N) = \perp_N$ and $\mu\Phi_{\text{ADD}}(+)(d, 0) = d$. The situation is more complicated if the second argument is > 0 . For example,

$$\begin{aligned}\Phi_{\text{ADD}}(\Omega)(+)(d, 1) &= \mathcal{I}(\text{if}_N)(\mathcal{I}(\text{is_0})(1), d, \overline{\Omega \cup \alpha_{(d, 1)}}(s(+ (X, s_1^{-1}(Y))))) \\ &= \overline{\Omega \cup \alpha_{(d, 1)}}(s(+ (X, s_1^{-1}(Y)))) \\ &= \mathcal{I}(s)(\Omega(+)(d, \mathcal{I}(s_1^{-1})(1))) \\ &= \mathcal{I}(s)(\perp_N) \\ &= \perp_N,\end{aligned}$$

but if we iterate Φ_{ADD} $n + 2$ times, $n \geq 0$, we get

$$\begin{aligned}\Phi_{\text{ADD}}^{n+2}(\Omega)(+)(d, 1) &= \overline{\Phi_{\text{ADD}}^{n+1}(\Omega) \cup \alpha_{(d, 1)}}(s(+ (X, s_1^{-1}(Y)))) \\ &= \mathcal{I}(s)(\Phi_{\text{ADD}}^{n+1}(\Omega)(+)(d, 0)) \\ &= \mathcal{I}(s)(d) \\ &= \begin{cases} d + 1 & \text{if } d \in N \\ \perp_N & \text{otherwise.} \end{cases}\end{aligned}$$

Similarly,

$$\begin{aligned}\Phi_{\text{ADD}}^2(\Omega)(+)(d, 2) &= \overline{\Phi_{\text{ADD}}(\Omega) \cup \alpha_{(d, 2)}}(s(+ (X, s_1^{-1}(Y)))) \\ &= \mathcal{I}(s)(\Phi_{\text{ADD}}(\Omega)(+)(d, 1)) \\ &= \perp_N,\end{aligned}$$

but if we iterate Φ_{ADD} $n + 3$ times, $n \geq 0$, we get

$$\begin{aligned}\Phi_{\text{ADD}}^{n+3}(\Omega)(+)(d, 2) &= \overline{\Phi_{\text{ADD}}^{n+2}(\Omega) \cup \alpha_{(d, 2)}}(s(+ (X, s_1^{-1}(Y)))) \\ &= \mathcal{I}(s)(\Phi_{\text{ADD}}^{n+2}(\Omega)(+)(d, 1)) \\ &= \begin{cases} d + 2 & \text{if } d \in N \\ \perp_N & \text{otherwise.} \end{cases}\end{aligned}$$

In general, it can be shown by induction on n that, for any $n \in N$,

$$\Phi_{\text{ADD}}^n(\Omega)(+)(d, e) = \begin{cases} d + e & \text{if } d, e \in N \text{ and } 0 \leq e < n \\ \perp_N & \text{otherwise,} \end{cases} \quad (3.3)$$

so, for all $d, e \in N_{\perp}$,

$$\begin{aligned}\mu\Phi_{\text{ADD}}(+)(d, e) &= \sqcup\{\Phi_{\text{ADD}}^n(\Omega)(+)(d, e) \mid n \in N\} \\ &= \begin{cases} d + e & \text{if } d, e \in N \\ \perp_N & \text{otherwise.} \end{cases}\end{aligned}$$

That is, $\mu\Phi_{\text{ADD}}(+)$ is the strict extension of $+$.

We can also use the fact that $\mu\Phi_{\text{ADD}}$ is a fixed point of Φ_{ADD} to verify that $\mu\Phi_{\text{ADD}}(+)(d, e) = d +_{\perp} e$ for any given $d, e \in N_{\perp}$. For example,

$$\begin{aligned}\mu\Phi_{\text{ADD}}(+)(3, 2) &= \Phi_{\text{ADD}}(\mu\Phi_{\text{ADD}})(+)(3, 2) \\ &= \overline{\mu\Phi_{\text{ADD}} \cup \alpha_{(3, 2)}}(\text{if}_N(\text{is_0}(Y), X, s(+ (X, s_1^{-1}(Y))))) \\ &= \mathcal{I}(s)(\mu\Phi_{\text{ADD}}(+)(3, 1)) \\ &= \mathcal{I}(s)(\Phi_{\text{ADD}}(\mu\Phi_{\text{ADD}})(+)(3, 1)) \\ &= \mathcal{I}(s)(\overline{\mu\Phi_{\text{ADD}} \cup \alpha_{(3, 1)}}(\text{if}_N(\text{is_0}(Y), X, s(+ (X, s_1^{-1}(Y))))) \\ &= \mathcal{I}(s)(\mathcal{I}(s)(\mu\Phi_{\text{ADD}}(+)(3, 0))) \\ &= \mathcal{I}(s)(\mathcal{I}(s)(\Phi_{\text{ADD}}(\mu\Phi_{\text{ADD}})(+)(3, 0))) \\ &= \mathcal{I}(s)(\mathcal{I}(s)(\overline{\mu\Phi_{\text{ADD}} \cup \alpha_{(3, 0)}}(\text{if}_N(\text{is_0}(Y), X, s(+ (X, s_1^{-1}(Y))))))) \\ &= \mathcal{I}(s)(\mathcal{I}(s)(3)) \\ &= 5.\end{aligned}$$

Before we go on we prove the following useful lemma.

Extension Lemma. Let $(\mathcal{T}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, and let \mathbf{P}, \mathbf{Q} be \mathbf{W} -programs such that $\mathbf{P} \subseteq \mathbf{Q}$.

1. For all $\mathbf{F} \in FV(\mathbf{P})$, $\mu\Phi_{\mathbf{P}}(\mathbf{F}) = \mu\Phi_{\mathbf{Q}}(\mathbf{F})$.
2. For all $\mathbf{t} \in TM_{\mathbf{W}}(FV(\mathbf{P}))$, $\overline{\mu\Phi_{\mathbf{P}}}(\mathbf{t}) = \overline{\mu\Phi_{\mathbf{Q}}}(\mathbf{t})$.

Proof. Let $\mathbf{F}(X_1, \dots, X_n) = \mathbf{t}$ be the defining equation for \mathbf{F} in \mathbf{P} . First we prove by induction on i that

$$\Phi_{\mathbf{P}}^i(\Omega)(\mathbf{F}) = \Phi_{\mathbf{Q}}^i(\Omega)(\mathbf{F}) \quad \text{for all } i \in N. \quad (3.4)$$

If $i = 0$ then $\Phi_P^0(\Omega)(F) = \Omega(F) = \Phi_Q^0(\Omega)(F)$, so assume $\Phi_P^i(\Omega)(F) = \Phi_Q^i(\Omega)(F)$ and let $d \in D_{\delta(F)}$. Then

$$\begin{aligned}\Phi_P^{i+1}(\Omega)(F)(d) &= \overline{\Phi_P^i(\Omega) \cup \alpha_d(t)} \\ &= \overline{\Phi_Q^i(\Omega) \cup \alpha_d(t)} \quad \text{by the induction hypothesis and} \\ &\quad \text{the coincidence lemma} \\ &= \Phi_Q^{i+1}(\Omega)(F)(d),\end{aligned}$$

and d is an arbitrary element of $D_{\delta(F)}$, so we have $\Phi_P^{i+1}(\Omega)(F) = \Phi_Q^{i+1}(\Omega)(F)$, concluding the induction. Now,

$$\begin{aligned}\mu\Phi_P(F) &= \sqcup\{\Phi_P^i(\Omega) \mid i \in N\}(F) \quad \text{by the fixed point theorem} \\ &= \sqcup\{\Phi_P^i(\Omega)(F) \mid i \in N\} \quad \text{by Theorem 16.3.9} \\ &= \sqcup\{\Phi_Q^i(\Omega)(F) \mid i \in N\} \quad \text{by (3.4)} \\ &= \sqcup\{\Phi_Q^i(\Omega) \mid i \in N\}(F) \quad \text{by Theorem 16.3.9} \\ &= \mu\Phi_Q(F),\end{aligned}$$

which completes the proof of part 1. Part 2 follows immediately from part 1 by the coincidence lemma. ■

We have one more step to take before we define the denotational semantics of W-programs. In the next section we will select from any complete, continuous W-structure Σ certain objects, the data objects, to get a data structure system Δ . We will then give the denotational semantics of W-programs in terms of Δ . However, it will turn out that Σ_N is already a data structure system, which we will also call Δ_N , so we can anticipate the next section and give the denotational semantics in Δ_N for W_N -programs.⁴ The idea is to give a single function that assigns a meaning to all W_N -programs.

Definition. The *denotational meaning function* for Δ_N , denoted \mathcal{D}_{Δ_N} , is defined

$$\mathcal{D}_{\Delta_N}(P) = \mu\Phi_P^\Sigma$$

for all W_N -programs P .

⁴ The reader who wishes to go on at this point to the chapter on operational semantics will be able to read the first two sections of that chapter as they apply to the particular structure Σ_N . We simply need to remark that Σ_N is a simple W_N -structure (as defined in Section 5 of the current chapter) and that $\Delta_N = \text{rep}(\Sigma_N) = \Sigma_N$ (as defined in Section 4 of the current chapter) is a simple data structure system for W_N .

Exercises

1. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, let \mathbf{P} be a \mathbf{W} -program, and let $\Phi_{\mathbf{P}} = \Phi_{\mathbf{P}}^{\Sigma}$. Prove the following statements.
 - (a) $\Phi_{\mathbf{P}}^i(\Omega) \in \mathcal{CA}_{\mathcal{T}}(FV(\mathbf{P}))$ for all $i \in N$.
 - (b) $\{\Phi_{\mathbf{P}}^i(\Omega) \mid i \in N\}$ is a chain in $(\mathcal{CA}_{\mathcal{T}}(FV(\mathbf{P})), \sqsubseteq_{\mathcal{CA}_{\mathcal{T}}(FV(\mathbf{P}))})$. [Hint: See Lemma 1 in Section 5 of Chapter 16.]
2. Give a \mathbf{W}_N -program \mathbf{P} such that $\mu\Phi_{\mathbf{P}}^{\Sigma_N} = \Omega$.
3. Give a \mathbf{W}_N -program \mathbf{P} such that $\Phi_{\mathbf{P}}^{\Sigma_N}$ has infinitely many fixed points.
4. Show that $\Phi_{\text{ADD}}^{\Sigma_N}$ has exactly one fixed point.
5. Give a \mathbf{W}_N -program \mathbf{P} with $FV(\mathbf{P}) = \{\mathbf{F}\}$ such that $\mu\Phi_{\mathbf{P}}^{\Sigma_N}(\mathbf{F})$ is not strict.
6. Prove (3.3).
7. Let $\Sigma = \Sigma_N$ and let \mathbf{P} be the \mathbf{W}_N -program with the equation

$$\mathbf{F}(X) = \text{if}_N(\text{is_0}(X), 2, s(F(s_1^{-1}(X)))).$$

- (a) Show by induction on n that, for any $n \in N$,

$$\Phi_{\mathbf{P}}^n(\Omega)(\mathbf{F})(x) = \begin{cases} x + 2 & \text{if } x \in N \text{ and } 0 \leq x < n \\ \perp_N & \text{if } x = \perp_N \text{ or } x \geq n \end{cases}$$

for all $x \in N_{\perp}$.

- (b) Show that $\mu\Phi_{\mathbf{P}}(\mathbf{F}) = f_{\perp}$, where $f(x) = x + 2$ for all $x \in N$.

8. Let \mathbf{P} be the \mathbf{W}_N -program with equations

$$\mathbf{F}(X, Y) = \text{if}_N(\text{is_0}(Y), X, s(F(X, s_1^{-1}(Y))))$$

$$\mathbf{G}(X) = F(X, X)$$

$$\mathbf{H}(X) = \text{if}_N(\text{is_0}(X), s(0), G(H(s_1^{-1}(X))))$$

and let $\Phi_{\mathbf{P}} = \Phi_{\mathbf{P}}^{\Sigma_N}$.

- (a) Let $\alpha(\mathbf{F}) = \cdot_{\perp}$ (the strict extension of the multiplication function), $\alpha(\mathbf{G})(x) = x +_{\perp} 2$ for all $x \in N_{\perp}$, and $\alpha(\mathbf{H})(x) = 3$ for all $x \in N_{\perp}$. What is $\Phi_{\mathbf{P}}(\alpha)(\mathbf{F})(3, 5)$? $\Phi_{\mathbf{P}}(\alpha)(\mathbf{G})(7)$? $\Phi_{\mathbf{P}}(\alpha)(\mathbf{H})(13)$?
- (b) What is $\Phi_{\mathbf{P}}^2(\alpha)(\mathbf{F})(3, 5)$? $\Phi_{\mathbf{P}}^2(\alpha)(\mathbf{G})(7)$? $\Phi_{\mathbf{P}}^2(\alpha)(\mathbf{H})(13)$?
- (c) Describe $\Phi_{\mathbf{P}}^i(\Omega)(\mathbf{F})$, $\Phi_{\mathbf{P}}^i(\Omega)(\mathbf{G})$, and $\Phi_{\mathbf{P}}^i(\Omega)(\mathbf{H})$ for all $i \in N$.
- (d) Describe $\mu\Phi_{\mathbf{P}}(\mathbf{F})$, $\mu\Phi_{\mathbf{P}}(\mathbf{G})$, and $\mu\Phi_{\mathbf{P}}(\mathbf{H})$.

9. Let \mathbf{P} be the \mathbf{W}_N -program with equations

$$\mathbf{F}(\mathbf{X}) = \text{if}_{\mathbf{N}}(\text{is_0}(\mathbf{X}), \mathbf{s}(0), \mathbf{G}(\mathbf{X}, \mathbf{X}))$$

$$\mathbf{G}(\mathbf{X}, \mathbf{Y}) = \text{if}_{\mathbf{N}}(\text{is_0}(\mathbf{Y}), \mathbf{F}(\mathbf{X}), \mathbf{s}(\mathbf{G}(\mathbf{X}, \mathbf{s}_1^{-1}(\mathbf{X}))))$$

and let $\Phi_{\mathbf{P}} = \Phi_{\mathbf{P}}^{\Sigma_N}$.

- (a) Let $\alpha(\mathbf{F})(x) = 3$ for all $x \in N_{\perp}$, and let $\alpha(\mathbf{G}) = +_{\perp}$. What is $\Phi_{\mathbf{P}}(\alpha)(\mathbf{F})(3)$? $\Phi_{\mathbf{P}}(\alpha)(\mathbf{G})(3, 2)$?
 - (b) What is $\Phi_{\mathbf{P}}^2(\alpha)(\mathbf{F})(3)$? $\Phi_{\mathbf{P}}^2(\alpha)(\mathbf{G})(3, 2)$?
 - (c) Describe $\Phi_{\mathbf{P}}^i(\Omega)$ for each $i \in N$.
 - (d) Describe $\mu\Phi_{\mathbf{P}}$.
10. Give a \mathbf{W}_N -program \mathbf{P} such that $\mu\Phi_{\mathbf{P}}^{\Sigma_N}(\mathbf{F}) = \cdot_{\perp}$ (the strict extension of the multiplication function) for some $\mathbf{F} \in FV(\mathbf{P})$.
11. Give a \mathbf{W}_N -program \mathbf{P} such that $\mu\Phi_{\mathbf{P}}^{\Sigma_N}(\mathbf{F}) = F_{\perp}$, where $F(n)$ is the n th Fibonacci number, for some $\mathbf{F} \in FV(\mathbf{P})$. [See Exercise 8.3 in Chapter 3 for the definition of *Fibonacci numbers*.]
12. Let $\&$, \vee , \sim be the usual operations on truth values. Give a \mathbf{W} -program \mathbf{P} such that, in any \mathbf{W} -structure, $\mu\Phi_{\mathbf{P}}(\&) = \&_{\perp}$, $\mu\Phi_{\mathbf{P}}(\vee) = \vee_{\perp}$, and $\mu\Phi_{\mathbf{P}}(\sim) = \sim_{\perp}$, where

$$\tau(\&) = \tau(\vee) = \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$$

and $\tau(\sim) = \mathbf{Bool} \rightarrow \mathbf{Bool}$.

13. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure. Suppose we extend the standard vocabulary \mathbf{W} to \mathbf{W}' by adding the symbols is_tt , is_ff , and suppose we give is_tt , is_ff their natural interpretations $\mathcal{I}(\text{is_tt})$, $\mathcal{I}(\text{is_ff})$ as in condition 5 on \mathcal{T} -interpretations. Give a \mathbf{W} -program \mathbf{P} with function variables Is_tt , Is_ff such that $\overline{\mu\Phi_{\mathbf{P}}}(\text{Is_tt}) = \mathcal{I}(\text{is_tt})$ and $\overline{\mu\Phi_{\mathbf{P}}}(\text{Is_ff}) = \mathcal{I}(\text{is_ff})$.
14. Let $\Sigma = (\mathcal{T}_N, \mathcal{I}_e)$, where \mathcal{I}_e is given in Exercise 2.6. What is $\mu\Phi_{\text{ADD}}^{\Sigma}$?
- 15.* Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, and let \mathbf{P} be a \mathbf{W} -program. Define $\Psi_{\mathbf{P}}^{\Sigma}: \mathcal{A}_{\mathcal{T}}(FV(\mathbf{P})) \rightarrow \mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$ exactly like $\Phi_{\mathbf{P}}^{\Sigma}$ except that its domain is $\mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$. [This exercise requires the results of Exercise 2.14.]
- (a) Show that $\alpha \in \mathcal{A}_{\mathcal{T}}(FV(\mathbf{P}))$ is a solution to \mathbf{P} in Σ if and only if α is a fixed point of $\Psi_{\mathbf{P}}^{\Sigma}$.
 - (b) Show that $\Psi_{\mathbf{P}}^{\Sigma}$ is monotonic.
 - (c) Give a \mathbf{W} -program \mathbf{Q} such that $\Psi_{\mathbf{Q}}^{\Sigma}$ is not continuous. [Hint: Let f be a function and C a chain such that $f(\sqcup C) \neq \sqcup f(C)$. For

- all $c \in C$, let $\alpha_c(\mathbf{F}) = f$ and let $\alpha_c(\mathbf{G})$ be the constant function $\alpha_c(\mathbf{G})(x) = c$. Put the equation $\mathbf{H}(\mathbf{X}) = \mathbf{F}(\mathbf{G}(\mathbf{X}))$ in \mathbf{Q} .]
- (d) Show that $(\Psi_P^\Sigma)^i(\Omega) \in \mathcal{CA}_T(FV(P))$ for all $i \in N$. [Hint: Use Theorem 2.3 in the induction step.]
- (e) Show that $\sqcup\{(\Psi_P^\Sigma)^i(\Omega) \mid i \in N\}$ exists and is $\mu\Psi_P^\Sigma$.

4. Denotational Semantics of W-Programs

Next we turn to the treatment of data structures. There are two properties that they should satisfy:

1. Since the semantics of program P is to be based on the function Φ_P , we want data structures to be rich enough to guarantee the existence of $\mu\Phi_P$ for every program P .
2. Since we need to be able to specify the inputs to a program, we want every element in a data structure to be the meaning of some term.

These two properties may seem to be contradictory. Property 1 requires data structures to have enough elements to give meanings to programs, and property 2 requires that data structures not have too many elements. We deal with these requirements in two steps. Theorem 3.3 guarantees that $\mu\Phi_P^\Sigma$ exists when Σ is a complete, continuous \mathbf{W} -structure, so we begin with such structures and pare them down so that property 2 is satisfied.

Definition. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure.

1. An element $d \in D_{\mathcal{T}(\tau)}$, for some $\tau \in TV(W)$, is *representable* in \mathbf{W} if there is some \mathbf{W} -program P and some term $t \in TM_W^\tau(FV(P))$ such that $d = (\mu\Phi_P^\Sigma)_\mathcal{I}(t)$. $\text{rep}(D_{\mathcal{T}(\tau)})$ is the set

$$\{d \in D_{\mathcal{T}(\tau)} \mid d \text{ is representable in } \mathbf{W}\},$$

$\sqsubseteq_{\text{rep}(\mathcal{T}(\tau))}$ is the restriction of $\sqsubseteq_{\mathcal{T}(\tau)}$ to $\text{rep}(D_{\mathcal{T}(\tau)})$, and $\text{rep}(\mathcal{T})$ is defined

$$\text{rep}(\mathcal{T})(\tau) = (\text{rep}(D_{\mathcal{T}(\tau)}), \sqsubseteq_{\text{rep}(\mathcal{T}(\tau))})$$

for all $\tau \in TV(W)$.

2. For any function

$$f \in D_{\mathcal{T}(\tau_1)} \times \cdots \times D_{\mathcal{T}(\tau_n)} \rightarrow D_{\mathcal{T}(\tau)},$$

where $\tau_1, \dots, \tau_n, \tau \in TV(W)$, let

$$\text{rep}(f) \in \text{rep}(D_{\mathcal{T}(\tau_1)}) \times \cdots \times \text{rep}(D_{\mathcal{T}(\tau_n)}) \rightarrow D_{\mathcal{T}(\tau)}$$

be defined

$$\text{rep}(f)(d_1, \dots, d_n) = f(d_1, \dots, d_n)$$

for all $(d_1, \dots, d_n) \in \text{rep}(D_{\mathcal{T}(\tau_1)}) \times \dots \times \text{rep}(D_{\mathcal{T}(\tau_n)})$. Then $\text{rep}(\mathcal{I})$ is defined

$$\text{rep}(\mathcal{I})(c) = \mathcal{I}(c) \quad \text{for all constant symbols } c \in \mathbf{W}$$

$$\text{rep}(\mathcal{I})(f) = \text{rep}(\mathcal{I}(f)) \quad \text{for all proper function symbols } f \in \mathbf{W}.$$

3. Finally, $\text{rep}(\Sigma) = (\text{rep}(\mathcal{T}), \text{rep}(\mathcal{I}))$ is the *data structure system for \mathbf{W} based on Σ* .

The point is that an arbitrary \mathbf{W} -structure Σ might contain objects that we can never use as data since there is no way to refer to them. Therefore, in defining functions that we wish to consider computable, we will restrict our attention to the representable objects in $\text{rep}(\Sigma)$. We might call these the *data objects* of Σ .

It is important to understand that even if d_1, \dots, d_n are representable, $\text{rep}(f)(d_1, \dots, d_n)$ may not be representable if f is some arbitrary function. However, we will show that $\text{rep}(f)(d_1, \dots, d_n)$ is representable when f is the interpretation $\mathcal{I}(f)$ of some $f \in \mathbf{W}$.

When \mathcal{T} is understood, we will generally write $(D_{r(\tau)}, \sqsubseteq_{r(\tau)})$ for $\text{rep}(\mathcal{T})(\tau)$,

$$D_{r(\tau_1)} \times \dots \times r(\tau_n) \quad \text{for } \text{rep}(D_{\mathcal{T}(\tau_1)}) \times \dots \times \text{rep}(D_{\mathcal{T}(\tau_n)}),$$

$$D_{r(\tau_1)} \times \dots \times r(\tau_n) \rightarrow r(\tau) \quad \text{for } \text{rep}(D_{\mathcal{T}(\tau_1)}) \times \dots \times \text{rep}(D_{\mathcal{T}(\tau_n)}) \rightarrow \text{rep}(D_{\mathcal{T}(\tau)}),$$

and, when $\delta(F) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, $D_{r(\delta(F))}$ for $D_{r(\tau_1)} \times \dots \times r(\tau_n)$.

Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure. Then for every constant symbol $c \in \mathbf{W}$, $\mathcal{I}(c)$ is representable since $\overline{\mu \Phi_P^{\Sigma}}(c) = \mathcal{I}(c)$ for any \mathbf{W} -program P . Moreover, for every $\tau \in TV(\mathbf{W})$, \perp_{τ} is representable: let $c \in \mathbf{W}_c$ be a constant symbol with $\tau(c) = \tau$, and let P be the \mathbf{W} -program with equation $B(X) = B(X)$, where $\tau(X) = \tau$ and $\tau(B) = \tau \rightarrow \tau$. Then $\overline{\mu \Phi_P^{\Sigma}}(B(c)) = \overline{\Omega(B(c))} = \perp_{\tau}$. (This explains, by the way, our requirement that \mathbf{W}_c contain a constant symbol of type τ for every $\tau \in TV(\mathbf{W}_c)$.)

In Σ_N , then, it is clear that every element in Bool is representable in \mathbf{W}_N . Moreover, every element in N_{\perp} is representable in \mathbf{W}_N : for all $n \in N$, $\overline{\mu \Phi_P(n)} = n$, where P is any \mathbf{W}_N -program, e.g., the empty program. (In a case like this we can simply say $\Omega(n) = n$.) Therefore, $\text{rep}(\mathcal{T}_N) = \mathcal{T}_N$, $\text{rep}(\mathcal{I}_N) = \mathcal{I}_N$, and $\text{rep}(\Sigma_N) = \Sigma_N$. We will write Δ_N for Σ_N when we want to emphasize that Σ_N is a data structure system for \mathbf{W}_N .

Now, let V_f be a set of function variables, and let $d = \overline{\mu \Phi_P^{\Sigma}}(t)$. It is useful to note that simply by changing the function variables in P and t , we can always find a \mathbf{W} -program Q and a term $u \in TM_W(FV(Q))$ such that

$FV(\mathbf{Q}) \cap V_f = \emptyset$ and $\overline{\mu\Phi_{\mathbf{Q}}^{\Sigma}}(\mathbf{u}) = \overline{\mu\Phi_{\mathbf{P}}^{\Sigma}}(\mathbf{t})$. Therefore, given representable elements d_1, \dots, d_n , we can always find \mathbf{W} -programs $\mathbf{P}_1, \dots, \mathbf{P}_n$ such that $FV(\mathbf{P}_i) \cap FV(\mathbf{P}_j) = \emptyset$, $1 \leq i < j \leq n$, and terms $\mathbf{t}_1, \dots, \mathbf{t}_n$ such that $\mathbf{t}_i \in TM_{\mathbf{W}}(FV(\mathbf{P}_i))$ and $d_i = \overline{\mu\Phi_{\mathbf{P}_i}^{\Sigma}}(\mathbf{t}_i)$, $1 \leq i \leq n$. We will say that $\mathbf{P}_1, \dots, \mathbf{P}_n$ are *consistent* if $FV(\mathbf{P}_i) \cap FV(\mathbf{P}_j) = \emptyset$, $1 \leq i < j \leq n$.

The first thing we need to do is show that data structure systems are \mathbf{W} -structures. We begin with a lemma that shows that, for all proper function symbols $\mathbf{f} \in \mathbf{W}$ with $\tau(\mathbf{f}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, $\text{rep}(\mathcal{I})(\mathbf{f}) \in D_{r(\tau_1)} \times \dots \times D_{r(\tau_n)} \rightarrow D_{r(\tau)}$. In other words, data structure systems are closed under the interpretations of the function symbols.

Lemma 1. Let $\Sigma = (\mathcal{F}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, and let $\mathbf{f} \in \mathbf{W}$ with $\tau(\mathbf{f}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$. If $(d_1, \dots, d_n) \in D_{r(\tau_1)} \times \dots \times D_{r(\tau_n)}$, then $\text{rep}(\mathcal{I})(\mathbf{f})(d_1, \dots, d_n) \in D_{r(\tau)}$.

Proof. Let $d_i = \overline{(\mu\Phi_{\mathbf{P}_i})}_{\mathcal{I}}(\mathbf{t}_i)$, $1 \leq i \leq n$, where $\mathbf{P}_1, \dots, \mathbf{P}_n$ are consistent. Then $\mathbf{P} = \bigcup_{i=1}^n \mathbf{P}_i$ is a \mathbf{W} -program, and

$$\begin{aligned} & (\overline{\mu\Phi_{\mathbf{P}}})_{\mathcal{I}}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \\ &= \mathcal{I}(\mathbf{f})\left(\overline{(\mu\Phi_{\mathbf{P}})}_{\mathcal{I}}(\mathbf{t}_1), \dots, \overline{(\mu\Phi_{\mathbf{P}})}_{\mathcal{I}}(\mathbf{t}_n)\right) \\ &= \mathcal{I}(\mathbf{f})\left(\overline{(\mu\Phi_{\mathbf{P}_1})}_{\mathcal{I}}(\mathbf{t}_1), \dots, \overline{(\mu\Phi_{\mathbf{P}_n})}_{\mathcal{I}}(\mathbf{t}_n)\right) \quad \text{by the extension lemma} \\ &= \mathcal{I}(\mathbf{f})(d_1, \dots, d_n) \\ &= \text{rep}(\mathcal{I})(\mathbf{f})(d_1, \dots, d_n) \quad \text{since } (d_1, \dots, d_n) \in D_{r(\tau_1)} \times \dots \times D_{r(\tau_n)}. \end{aligned}$$

■

We will use the next lemma when considering the interpretations of the built-in function symbols `is_f` and \mathbf{f}_i^{-1} .

Lemma 2. Let $\Sigma = (\mathcal{F}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, and let $\mathbf{f} \in \mathbf{W}_c$ with $\tau(\mathbf{f}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$. If $(d_1, \dots, d_n) \in D_{\tau_1} \times \dots \times D_{\tau_n}$ and $\mathcal{I}(\mathbf{f})(d_1, \dots, d_n) \in D_{r(\tau)} - \{\perp_{\tau}\}$, then $(d_1, \dots, d_n) \in D_{r(\tau_1)} \times \dots \times D_{r(\tau_n)}$.

Proof. Let $\mathcal{I}(\mathbf{f})(d_1, \dots, d_n) = \overline{(\mu\Phi_{\mathbf{P}})}_{\mathcal{I}}(\mathbf{t})$. Then for $1 \leq i \leq n$,

$$\begin{aligned} \overline{(\mu\Phi_{\mathbf{P}})}_{\mathcal{I}}(\mathbf{f}_i^{-1}(\mathbf{t})) &= \mathcal{I}(\mathbf{f}_i^{-1})\left(\overline{(\mu\Phi_{\mathbf{P}})}_{\mathcal{I}}(\mathbf{t})\right) \\ &= \mathcal{I}(\mathbf{f}_i^{-1})(\mathcal{I}(\mathbf{f})(d_1, \dots, d_n)) \\ &= d_i. \end{aligned}$$

■

Theorem 4.1. Let $\Sigma = (\mathcal{F}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure. Then $\text{rep}(\Sigma)$ is a \mathbf{W} -structure.

Proof. For any $\tau \in TV(W)$, $(D_{r(\tau)}, \sqsubseteq_{r(\tau)})$ is clearly a partial order with bottom element \perp_τ , and $(D_{r(\text{Bool})}, \sqsubseteq_{r(\text{Bool})}) = \mathcal{T}(\text{Bool})$, so $\text{rep}(\mathcal{T})$ is a type assignment for W .

Now we need to show that $\text{rep}(\mathcal{T})$ is a $\text{rep}(\mathcal{T})$ -interpretation for W . Note that in the context of the type assignment $\text{rep}(\mathcal{T})$, each reference to a set D_τ in the definition of \mathcal{T} -interpretations should be understood as referring to $\text{rep}(D_{\mathcal{T}(\tau)})$, i.e., $D_{r(\tau)}$. Also, each reference there to \mathcal{T} should be understood as a reference to $\text{rep}(\mathcal{T})$. It is clear that $\text{rep}(\mathcal{T})(c) = \mathcal{T}(c)$ for all constant symbols $c \in W_c$, so condition 1 in the definition of \mathcal{T} -interpretations is satisfied. If $f \in W_c$ is a proper function symbol with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, and $d_i \in D_{r(\tau_i)}$, $1 \leq i \leq n$, then $\text{rep}(\mathcal{T})(f)(d_1, \dots, d_n) \in D_{r(\tau)}$ by Lemma 1, so condition 2a is satisfied. Conditions 2b, 2c, and 3 follow immediately from the definition of $\text{rep}(\mathcal{T})$. Condition 4 follows immediately from Lemma 1 and the definition of $\text{rep}(\mathcal{T})$, as does condition 5 for all constant symbols in W_c^- , so let $f \in W_c^-$ with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, and let $d \in D_{r(\tau)}$. If $d = \perp_\tau$ then $\text{rep}(\mathcal{T})(\text{is_f})(d) = \mathcal{T}(\text{is_f})(d) = \perp_{\text{Bool}}$, so assume $d \neq \perp_\tau$. If $d \in \text{ran}(\text{rep}(\mathcal{T})(f))$, then obviously $d \in \text{ran } \mathcal{T}(f)$, so that $\text{rep}(\mathcal{T})(\text{is_f})(d) = \mathcal{T}(\text{is_f})(d) = \text{tt}$. Now, if $d \in \text{ran } \mathcal{T}(f)$, then $d = \mathcal{T}(f)(d_1, \dots, d_n)$ for some $(d_1, \dots, d_n) \in D_{r(\delta(f))}$ by Lemma 2, so $d \in \text{ran}(\text{rep}(\mathcal{T})(f))$. Therefore, if $d \notin \text{ran}(\text{rep}(\mathcal{T})(f))$, then $d \notin \text{ran } \mathcal{T}(f)$, and $\text{rep}(\mathcal{T})(\text{is_f})(d) = \mathcal{T}(\text{is_f})(d) = \text{ff}$, so condition 5 is satisfied. Finally, condition 6 is satisfied by a similar argument, so $\text{rep}(\mathcal{T})$ is a $\text{rep}(\mathcal{T})$ -interpretation, and $\text{rep}(\Sigma)$ is a W -structure. ■

We can now define the denotational semantics of recursion programs. For a complete, continuous W -structure $(\mathcal{T}, \mathcal{I})$ and a variable assignment $\alpha \in \mathcal{A}_{\mathcal{T}}(V)$, where V is a set of function variables, let $\text{rep}(\alpha)$ be the function on V defined by

$$\text{rep}(\alpha)(F) = \text{rep}(\alpha(F)) \quad \text{for all } F \in V.$$

Note that the domain of $\alpha(F)$ is $D_{r(\delta(F))}$ for all $F \in V$, but $\text{rep}(\alpha)$ is a variable assignment in $\mathcal{A}_{\text{rep}(\mathcal{T})}(V)$ if and only if

$$\text{rep}(\alpha)(F)(d) \in D_{r(\rho(F))} \quad \text{for all } F \in V \text{ and all } d \in D_{r(\delta(F))}.$$

Definition. Let Σ be a complete, continuous W -structure, and let $\Delta = \text{rep}(\Sigma)$. The *denotational meaning function* for Δ , denoted \mathcal{D}_Δ , is defined

$$\mathcal{D}_\Delta(P) = \text{rep}(\mu\Phi_P^\Sigma)$$

for all W -programs P .

For a W -program P and $F \in FV(P)$, we have

$$\mathcal{D}_\Delta(P)(F) = \text{rep}(\mu\Phi_P^\Sigma)(F) = \text{rep}(\mu\Phi_P^\Sigma(F)).$$

The point is that, rather than taking $\mu\Phi_P^\Sigma(F)$ as the function assigned to F , we assign to F a function whose domain consists only of representable objects.

We showed that $\mu\Phi_P^\Sigma$ is a solution to P in any complete, continuous W -structure Σ . We now show that $\mathcal{D}_\Delta(P)$ is a solution to P in $\Delta = \text{rep}(\Sigma)$. That is, we still have a solution when we restrict our attention to the data structure system based on Σ . We begin with three lemmas that let us ignore nonrepresentable objects when applying $\mu\Phi_P^\Sigma$ to terms. In particular, we want to show that if $\beta \in \mathcal{A}_{\text{rep}(\mathcal{I})}(V)$, where V is a set of individual variables, then for any term $t \in TM_W(FV(P) \cup V)$,

$$\overline{(\mu\Phi_P^\Sigma \cup \beta)}_{\mathcal{I}}(t) = \overline{\text{rep}(\mu\Phi_P^\Sigma) \cup \beta}_{\text{rep}(\mathcal{I})}(t).$$

It will follow easily, then, that $\mathcal{D}_\Delta(P) = \text{rep}(\mu\Phi_P^\Sigma)$ is a solution to P in Δ .

Lemma 3. Let $\Sigma = (\mathcal{F}, \mathcal{I})$ be a complete, continuous W -structure, let P be a W -program, let $V \subseteq \text{VAR}_1$, and let $\alpha \in \mathcal{A}_{\text{rep}(\mathcal{I})}(V)$. Then for any term $t \in TM_W(V \cup FV(P))$, $\overline{(\mu\Phi_P^\Sigma \cup \alpha)}_{\mathcal{I}}(t) \in D_{r(\tau(t))}$.

Proof. We have $\mu\Phi_P^\Sigma \cup \alpha \in \mathcal{A}_{\mathcal{I}}(V \cup FV(P))$, which implies $\overline{(\mu\Phi_P^\Sigma \cup \alpha)}_{\mathcal{I}}(t) \in D_{r(t)}$ by Theorem 2.1, so we need to show only that $\overline{(\mu\Phi_P^\Sigma \cup \alpha)}_{\mathcal{I}}(t)$ is representable. We argue by structural induction on t . If t is a constant symbol $c \in W$, then $\overline{(\mu\Phi_P^\Sigma \cup \alpha)}_{\mathcal{I}}(c)$ is clearly representable, and if t is $X \in V$, then $\overline{(\mu\Phi_P^\Sigma \cup \alpha)}_{\mathcal{I}}(X) = \alpha(X)$ is representable by assumption. If t is $f(t_1, \dots, t_n)$, where $f \in W$, then

$$\begin{aligned} & \overline{(\mu\Phi_P^\Sigma \cup \alpha)}_{\mathcal{I}}(f(t_1, \dots, t_n)) \\ &= \mathcal{I}(f)(\overline{(\mu\Phi_P^\Sigma \cup \alpha)}_{\mathcal{I}}(t_1), \dots, \overline{(\mu\Phi_P^\Sigma \cup \alpha)}_{\mathcal{I}}(t_n)) \\ &= \mathcal{I}(f)(\overline{(\mu\Phi_{P_1}^\Sigma)}_{\mathcal{I}}(u_1), \dots, \overline{(\mu\Phi_{P_n}^\Sigma)}_{\mathcal{I}}(u_n)) \end{aligned}$$

for some $P_i, u_i, 1 \leq i \leq n$, by the induction hypothesis, where P_1, \dots, P_n are consistent

$$\begin{aligned} &= \mathcal{I}(f)(\overline{(\mu\Phi_{P_0}^\Sigma)}_{\mathcal{I}}(u_1), \dots, \overline{(\mu\Phi_{P_0}^\Sigma)}_{\mathcal{I}}(u_n)) \\ &\quad \text{by the extension lemma, where } P_0 = \bigcup_{i=1}^n P_i \\ &= \overline{(\mu\Phi_{P_0}^\Sigma)}_{\mathcal{I}}(f(u_1, \dots, u_n)) \\ &\in D_{r(t)} \quad \text{by Theorem 2.1,} \end{aligned}$$

and $\overline{(\mu\Phi_{P_0}^\Sigma)}_{\mathcal{I}}(f(u_1, \dots, u_n))$ is representable, so it is in $D_{r(t)}$. The argument is similar if t is $F(t_1, \dots, t_n)$ with $F \in FV(P)$. ■

Lemma 4. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, let $\Delta = \text{rep}(\Sigma)$, and let \mathbf{P} be a \mathbf{W} -program. Then $\text{rep}(\mu\Phi_{\mathbf{P}}^{\Sigma}) \in \mathcal{A}_{\text{rep}(\mathcal{T})}(FV(\mathbf{P}))$.

Proof. Let $\mathbf{F}(X_1, \dots, X_n) = t$ be an equation in \mathbf{P} , and let $d \in D_{r(\delta(\mathbf{F}))}$. Then

$$\begin{aligned}\text{rep}(\mu\Phi_{\mathbf{P}}^{\Sigma})(\mathbf{F})(d) &= \text{rep}(\mu\Phi_{\mathbf{P}}^{\Sigma}(\mathbf{F}))(d) \\ &= \mu\Phi_{\mathbf{P}}^{\Sigma}(\mathbf{F})(d) && \text{since } d \in D_{r(\delta(\mathbf{F}))} \\ &= \overline{(\mu\Phi_{\mathbf{P}}^{\Sigma} \cup \alpha_d)}_{\mathcal{I}}(t) && \text{since } \mu\Phi_{\mathbf{P}}^{\Sigma} \text{ is a solution to } \mathbf{P} \\ &\in D_{r(\tau(t))} && \text{by Lemma 3.} \quad \blacksquare\end{aligned}$$

Lemma 5. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, let $V_i \subseteq \text{VAR}_1$ and $V_f \subseteq \text{VAR}_F$, let $\alpha \in \mathcal{A}_{\mathcal{I}}(V_f)$ be such that $\text{rep}(\alpha) \in \mathcal{A}_{\text{rep}(\mathcal{T})}(V_f)$, and let $\beta \in \mathcal{A}_{\text{rep}(\mathcal{T})}(V_i)$. Then for any term $t \in \text{TM}_{\mathbf{W}}(V_f \cup V_i)$,

$$\overline{(\alpha \cup \beta)}_{\mathcal{I}}(t) = \overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(t).$$

Proof. Note that $\text{rep}(\alpha) \cup \beta \in \mathcal{A}_{\text{rep}(\mathcal{T})}(V_f \cup V_i)$, so by Theorems 2.1 and 4.1,

$$\overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(t) \in D_{r(\tau(t))} \quad \text{for any } t \in \text{TM}_{\mathbf{W}}(V_f \cup V_i). \quad (4.1)$$

We argue by structural induction on t . If t is a constant symbol $c \in \mathbf{W}$, then

$$\overline{(\alpha \cup \beta)}_{\mathcal{I}}(c) = \mathcal{I}(c) = \text{rep}(\mathcal{I})(c) = \overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(c),$$

and if t is $X \in V_i$, then

$$\overline{(\alpha \cup \beta)}_{\mathcal{I}}(X) = \beta(X) = \overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(X).$$

If t is $f(t_1, \dots, t_n)$, where $f \in \mathbf{W}$, then

$$\begin{aligned}&\overline{(\alpha \cup \beta)}_{\mathcal{I}}(f(t_1, \dots, t_n)) \\ &= \mathcal{I}(f)(\overline{(\alpha \cup \beta)}_{\mathcal{I}}(t_1), \dots, \overline{(\alpha \cup \beta)}_{\mathcal{I}}(t_n)) \\ &= \mathcal{I}(f)(\overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(t_1), \dots, \overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(t_n)) \\ &\qquad \text{by the induction hypothesis} \\ &= \text{rep}(\mathcal{I})(f)(\overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(t_1), \dots, \overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(t_n)) \\ &\qquad \text{by (4.1)} \\ &= \overline{(\text{rep}(\alpha) \cup \beta)}_{\text{rep}(\mathcal{T})}(f(t_1, \dots, t_n)).\end{aligned}$$

The argument is similar if t is $F(t_1, \dots, t_n)$, where $F \in V_f$. ■

Theorem 4.2. Let $\Sigma = (\mathcal{F}, \mathcal{I})$ be a complete, continuous \mathbf{W} -structure, and let $\Delta = \text{rep}(\Sigma)$. Then for any \mathbf{W} -program \mathbf{P} , $\mathcal{D}_\Delta(\mathbf{P})$ is a solution to \mathbf{P} in Δ .

Proof. Let \mathbf{P} be a \mathbf{W} -program, let $\mathbf{F}(X_1, \dots, X_n) = t$ be an equation in \mathbf{P} , and let $d \in D_{r(\delta(\mathbf{F}))}$. Then

$$\begin{aligned}\mathcal{D}_\Delta(\mathbf{P})(\mathbf{F})(d) &= \text{rep}(\mu\Phi_\mathbf{P}^\Sigma)(\mathbf{F})(d) \\ &= \overline{(\mu\Phi_\mathbf{P}^\Sigma \cup \alpha_d)}_{\mathcal{I}}(t) && \text{as in the proof of Lemma 4} \\ &= \overline{\left(\text{rep}(\mu\Phi_\mathbf{P}^\Sigma) \cup \alpha_d\right)}_{\text{rep}(\mathcal{I})}(t) && \text{by Lemmas 4 and 5} \\ &= \overline{(\mathcal{D}_\Delta(\mathbf{P}) \cup \alpha_d)}_{\text{rep}(\mathcal{I})}(t).\end{aligned}$$

Let Δ be a data structure system for \mathbf{W} . Now that we have a meaning in Δ for every \mathbf{W} -program, it makes sense to ask if a given \mathbf{W} -program \mathbf{P} defines the functions we want it to define. That is, we can ask if \mathbf{P} is correct. Determining that a program is correct is known as *program verification*.

Definition. Let Δ be a data structure system for \mathbf{W} , let \mathbf{P} be a \mathbf{W} -program, and let $f \in D_{r(\tau_1) \times \dots \times r(\tau_n) \rightarrow r(\tau)}$ for some $\tau_1, \dots, \tau_n, \tau \in TV(\mathbf{W})$. We say that \mathbf{P} is *partially correct* with respect to f if

$$\mathcal{D}_\Delta(\mathbf{P})(\mathbf{F}) \sqsubseteq_{r(\tau_1) \times \dots \times r(\tau_n) \rightarrow r(\tau)} f$$

for some $\mathbf{F} \in \mathbf{P}$, and we say that \mathbf{P} is *totally correct* with respect to f if $\mathcal{D}_\Delta(\mathbf{P})(\mathbf{F}) = f$ for some $\mathbf{F} \in \mathbf{P}$.

For example, we indicated in the previous section that $\mu\Phi_{\text{ADD}}^{\Sigma_N}(+) = +_\perp$, and $\mathcal{D}_{\Delta_N}(\text{ADD}) = \mu\Phi_{\text{ADD}}^{\Sigma_N}$ since $\Delta_N = \Sigma_N$, so **ADD** is totally correct with respect to $+_\perp$. Recall that the correctness argument for **ADD** was based on ordinary induction. We will now give an application of fixed point induction in establishing a partial correctness result.

Let $\text{eq}: N_\perp \times N_\perp \rightarrow \text{Bool}$ (where $\text{Bool} = \{\perp_{\text{Bool}}, \text{TRUE}, \text{FALSE}\}$ here) be the strict function defined by

$$\text{eq}(x, y) = \begin{cases} \text{TRUE} & \text{if } x, y \in N \text{ and } x = y \\ \text{FALSE} & \text{if } x, y \in N \text{ and } x \neq y \\ \perp_{\text{Bool}} & \text{otherwise,} \end{cases}$$

and let **EQ** be the \mathbf{W}_N -program with the equation

$$\mathbf{E}(\mathbf{X}, \mathbf{Y}) = \text{if}_{\text{Bool}}(\text{is_0}(\mathbf{X}), \text{if}_{\text{Bool}}(\text{is_0}(\mathbf{Y}), \text{tt}, \text{ff}), \mathbf{E}(s_1^{-1}(\mathbf{X}), s_1^{-1}(\mathbf{Y}))).$$

Writing $\mathcal{C}\mathcal{A}$ for $\mathcal{C}\mathcal{A}_{\mathcal{I}_N}(\{\mathbf{E}\})$, let $\alpha_{\text{eq}} \in \mathcal{C}\mathcal{A}$ be the assignment $\alpha_{\text{eq}}(\mathbf{E}) = \text{eq}$, and let $P(x)$ be the predicate on $\mathcal{C}\mathcal{A}$ defined by

$$P(\alpha) = \begin{cases} \text{TRUE} & \text{if } \alpha \sqsubseteq_{\mathcal{C}\mathcal{A}} \alpha_{\text{eq}} \\ \text{FALSE} & \text{otherwise.} \end{cases}$$

If \mathcal{A} is a chain in $(\mathcal{C}\mathcal{A}, \sqsubseteq_{\mathcal{C}\mathcal{A}})$ such that $P(\alpha)$ holds for all $\alpha \in \mathcal{A}$, then clearly $P(\sqcup \mathcal{A})$ holds, so $P(x)$ is admissible. We want to show by fixed point induction that $P(\mu \Phi_{\text{EQ}})$ holds. Let $\Omega = \Omega_{\mathcal{C}\mathcal{A}}$. It is obvious that $P(\Omega)$ holds, so we assume $P(\Phi_{\text{EQ}}^i(\Omega))$ and show that $P(\Phi_{\text{EQ}}^{i+1}(\Omega))$ holds. We have $\Phi_{\text{EQ}}^i(\Omega) \sqsubseteq_{\mathcal{C}\mathcal{A}} \alpha_{\text{eq}}$ by the induction hypothesis, which implies $\Phi_{\text{EQ}}^{i+1}(\Omega) \sqsubseteq_{\mathcal{C}\mathcal{A}} \Phi_{\text{EQ}}(\alpha_{\text{eq}})$ by the monotonicity of Φ_{EQ} , so if we can show that $\Phi_{\text{EQ}}(\alpha_{\text{eq}}) \sqsubseteq_{\mathcal{C}\mathcal{A}} \alpha_{\text{eq}}$, then we will have $\Phi_{\text{EQ}}^{i+1}(\Omega) \sqsubseteq_{\mathcal{C}\mathcal{A}} \alpha_{\text{eq}}$, i.e., $P(\Phi_{\text{EQ}}^{i+1}(\Omega))$. If $x = \perp_N$ then it is easy to see that

$$\Phi_{\text{EQ}}(\alpha_{\text{eq}})(\mathbf{E})(x, y) = \perp_{\text{Bool}} = \alpha_{\text{eq}}(\mathbf{E})(x, y),$$

so assume that $x \in N$. If $x = 0$, then

$$\begin{aligned} \Phi_{\text{EQ}}(\alpha_{\text{eq}})(\mathbf{E})(x, y) &= \begin{cases} \text{TRUE} & \text{if } y = x \\ \text{FALSE} & \text{if } y \in N \text{ and } y \neq x \\ \perp_{\text{Bool}} & \text{if } y = \perp_N \end{cases} \\ &= \alpha_{\text{eq}}(\mathbf{E})(x, y), \end{aligned}$$

and if $x > 0$, then

$$\begin{aligned} \Phi_{\text{EQ}}(\alpha_{\text{eq}})(\mathbf{E})(x, y) &= \alpha_{\text{eq}}(\mathbf{E})(\mathcal{I}(s_1^{-1})(x), \mathcal{I}(s_1^{-1})(y)) \\ &= \begin{cases} \text{TRUE} & \text{if } x = y \\ \text{FALSE} & \text{if } y \in N - \{0\} \text{ and } x \neq y \\ \perp_{\text{Bool}} & \text{otherwise} \end{cases} \\ &\sqsubseteq_{\text{Bool}} \alpha_{\text{eq}}(\mathbf{E})(x, y). \end{aligned}$$

This concludes the proof of $P(\Phi_{\text{EQ}}^{i+1}(\Omega))$, so $P(\mu \Phi_{\text{EQ}})$ holds by fixed point induction. Therefore,

$$\mathcal{D}_{\Delta_N}(\text{EQ})(\mathbf{E}) = \mu \Phi_{\text{EQ}}(\mathbf{E}) \sqsubseteq_{N_1 \times N_1 \rightarrow \text{Bool}} \text{eq},$$

and **EQ** is partially correct with respect to **eq**.

Exercises

1. Let $P_1 = \{F(X) = F(s(X))\}$, $P_2 = \{F(X) = s(F(X))\}$ be W_N -programs. Give consistent W_N -programs Q_1 , Q_2 such that, in any W_N -structure Σ , $\mu\Phi_{Q_1 \cup Q_2}^\Sigma(G_1) = \mu\Phi_{P_1}^\Sigma(F)$ and $\mu\Phi_{Q_1 \cup Q_2}^\Sigma(G_2) = \mu\Phi_{P_2}^\Sigma(F)$.
2. Give a standard vocabulary $W = W_c \cup B(W_c)$ and a W -structure $\Sigma = (\mathcal{T}, \mathcal{I})$ such that, for some $\tau \in TV(W)$, there is an element $d \in D_\tau - \{\perp_\tau\}$ that is not in the range of $\mathcal{I}(f)$ for any $f \in W_c$.
3. Give a W_N -structure Σ such that $\text{rep}(\Sigma) \neq \Sigma$.
4. Show that Lemma 2 is not necessarily true for all $f \in W$.
5. (a) Give a standard vocabulary W , a W -structure $\Sigma = (\mathcal{T}, \mathcal{I})$, and a function $f \in D_{\tau_1} \rightarrow D_{\tau_2}$, for some $\tau_1, \tau_2 \in TV(W)$, such that $\text{rep}(f) \notin D_{r(\tau_1)} \rightarrow D_{r(\tau_2)}$.
(b) Give an assignment $\alpha \in \mathcal{A}_{\mathcal{I}}(\{F\})$, where F is a function variable, such that $\text{rep}(\alpha) \notin \mathcal{A}_{\text{rep}(\mathcal{T})}(\{F\})$.
6. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous W -structure, let V be a set of function variables, and let $\alpha, \beta \in \mathcal{C}\mathcal{A}_{\mathcal{I}}(V)$. Show that if $\alpha \sqsubseteq_{\mathcal{C}\mathcal{A}_{\mathcal{I}}(V)} \beta$, then $\text{rep}(\alpha)(F) \sqsubseteq_{r(\delta(F)) \rightarrow \rho(F)} \text{rep}(\beta)(F)$ for all $F \in V$.
7. Show that **EQ** is not totally correct with respect to **eq**.
8. Use fixed point induction to show that **ADD** is partially correct with respect to $+_\perp$.
9. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous W -structure such that $\text{rep}(\Sigma) = \Sigma$, let P be a W -program, and let $\alpha \in \mathcal{C}\mathcal{A}_{\mathcal{I}}(FV(P))$ be a solution to P in Σ .
 - (a) Show that $\mu\Phi_P^\Sigma \sqsubseteq_{\mathcal{C}\mathcal{A}_{\mathcal{I}}(FV(P))} \alpha$.
 - (b) Show that for all $F \in FV(P)$, P is partially correct with respect to $\alpha(F)$.
- 10.* Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous W -structure, and let $\Omega_1 = \Omega_{\mathcal{C}\mathcal{A}_{\mathcal{I}}(\text{VAR}_1)}$. For any $\tau \in TV(W)$, we will say that an element $d \in D_\tau$ is *constructed* if $d = \overline{\Omega_i}(t)$ for some $t \in TM_{W_c}(\text{VAR}_1)$.
 - (a) Let P be a W -program. Show by induction on i that for all $V \subseteq \text{VAR}_1$, all $\beta \in \mathcal{A}_{\mathcal{I}}(V)$ such that $\beta(X)$ is constructed for all $X \in V$, and all $t \in TM_W(FV(P) \cup V)$, $\Phi_P^i(\Omega) \cup \beta(t)$ is constructed. [Hint: For cases $i = 0$ and $i = k + 1$, argue by structural induction on t .]
 - (b) Let $\tau \in TV(W)$, and let $d \in D_{r(\tau)}$. Show that $d = \sqcup C$ for some chain C of constructed elements. [Hint: Use part (a) with $V = \emptyset$.]
 - (c) Let $\tau \in TV(W)$. Show that for every $d \in D_{r(\tau)} - \{\perp_\tau\}$, $d \in \text{ran } \mathcal{I}(f)$ for some $f \in W_c$. [Hint: See Exercise 2.9.] Compare with Exercise 2.

5. Simple Data Structure Systems

So far Δ_N is the only data structure system we have seen, so in this section we give some more examples. In particular, we look at a rather simple form of data structure system.

Definition. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure. Σ is a *simple* \mathbf{W} -structure if

1. $\mathcal{T}(\tau)$ is a flat cpo for all $\tau \in TV(\mathbf{W})$, and
2. $\mathcal{I}(f)$ is strict for every proper constructor function symbol $f \in \mathbf{W}_c$.

$\Delta = \text{rep}(\Sigma)$ is a *simple data structure system* for \mathbf{W} if Σ is a simple \mathbf{W} -structure.

It is easy to see that any simple \mathbf{W} -structure is complete and continuous. Clearly, Δ_N is simple. For another example, we extend Δ_N with tuples of natural numbers to create a simple \mathbf{W}_{NL} -structure, where \mathbf{W}_{NL} is the vocabulary for lists described in Section 1. Let $TUP(N)$ be the set of all tuples of natural numbers, including the “empty” tuple $()$, and let $(TUP(N)_\perp, \sqsubseteq_{TUP(N)_\perp})$ be the flat cpo on $TUP(N)$. We define $\Sigma_{NL} = (\mathcal{T}_{NL}, \mathcal{I}_{NL})$ as follows:

$$\mathcal{T}_{NL}(\mathbf{Bool}) = \mathcal{T}_N(\mathbf{Bool})$$

$$\mathcal{T}_{NL}(\mathbf{N}) = \mathcal{T}_N(\mathbf{N})$$

$$\mathcal{T}_{NL}(\mathbf{NL}) = (TUP(N)_\perp, \sqsubseteq_{TUP(N)_\perp})$$

$$\mathcal{I}_{NL}(\mathbf{0}) = 0$$

$$\mathcal{I}_{NL}(s) = s_\perp$$

$$\mathcal{I}_{NL}(\mathbf{nil}) = ()$$

$$\mathcal{I}_{NL}(\mathbf{cons}) = \mathbf{cons}_\perp \quad (\text{the strict extension of cons})$$

where $\mathbf{cons}: N \times TUP(N) \rightarrow TUP(N)$ is defined

$$\mathbf{cons}(m, (m_1, \dots, m_n)) = (m, m_1, \dots, m_n).$$

For built-in function symbols f , $\mathcal{I}_{NL}(f)$ is defined according to conditions 4–6 on \mathcal{I} -interpretations. It is easy to check that Σ_{NL} is a simple \mathbf{W} -structure. Moreover, we have $\bar{\Omega}(\mathbf{nil}) = ()$, and for any tuple (m_1, \dots, m_n) we have

$$\bar{\Omega}(\mathbf{cons}(m_1, \mathbf{cons}(m_2, \dots \mathbf{cons}(m_n, \mathbf{nil}) \dots))) = (m_1, \dots, m_n).$$

Therefore, every element in $TUP(N)_\perp$ is representable, so we have $\Sigma_{NL} = \text{rep}(\Sigma_{NL})$, and Σ_{NL} is a simple data structure system for \mathbf{W}_{NL} . We

will write Δ_{NL} for Σ_{NL} when we are interested in Σ_{NL} as a data structure system.

Now, let **LIST** be the W_{NL} -program with equations

$$\begin{aligned} \text{Length}(X) &= \text{if}_N(\text{is_nil}(X), 0, s(\text{Length}(\text{cons}_2^{-1}(X)))) \\ \text{Nth}(X, Y) &= \text{if}_N(\text{is_0}(X), \text{cons}_1^{-1}(Y), \text{Nth}(s_1^{-1}(X), \text{cons}_2^{-1}(Y))) \\ \text{Cat}(X, Y) &= \text{if}_{NL}(\text{is_nil}(X), Y, \text{cons}(\text{cons}_1^{-1}(X), \text{Cat}(\text{cons}_2^{-1}(X), Y))) \\ \text{Rev}(X) &= \text{if}_{NL}(\text{is_nil}(X), \\ &\quad X, \text{Cat}(\text{Rev}(\text{cons}_2^{-1}(X)), \\ &\quad \text{cons}(\text{cons}_1^{-1}(X), \text{nil}))). \end{aligned}$$

Then $\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Length})$ evaluates the length of a list. More precisely, if $\text{len}((m_1, \dots, m_n)) = n$ for any list $(m_1, \dots, m_n) \in \text{TUP}(N)$, then $\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Length}) = \text{len}_\perp$. It is clear that $\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Length})$ is strict, so we argue by induction on the length of lists:

$$\begin{aligned} \mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Length})(()) &= \mu\Phi_{\text{LIST}}(\text{Length})(()) \\ &= \Phi_{\text{LIST}}(\mu\Phi_{\text{LIST}})(\text{Length})(()) \\ &= \overline{\mu\Phi_{\text{LIST}} \cup \alpha_{()}}(\text{if}_N(\text{is_nil}(X), 0, s(\text{Length}(\text{cons}_2^{-1}(X))))) \\ &= \mathcal{I}_{NL}(\text{if}_N)(\mathcal{I}_{NL}(\text{is_nil})(()), 0, \overline{\mu\Phi_{\text{LIST}} \cup \alpha_{()}}(s(\text{Length}(\text{cons}_2^{-1}(X))))) \\ &= 0, \end{aligned}$$

and

$$\begin{aligned} \mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Length})((m_1, \dots, m_{n+1})) &= \overline{\mu\Phi_{\text{LIST}} \cup \alpha_{(m_1, \dots, m_{n+1})}}(\text{if}_N(\text{is_nil}(X), 0, s(\text{Length}(\text{cons}_2^{-1}(X))))) \\ &= \mathcal{I}_{NL}(s)(\mu\Phi_{\text{LIST}}(\text{Length})(\mathcal{I}_{NL}(\text{cons}_2^{-1})((m_1, \dots, m_{n+1})))) \\ &= \mathcal{I}_{NL}(s)(\mu\Phi_{\text{LIST}}(\text{Length})((m_2, \dots, m_{n+1}))) \\ &= \mathcal{I}_{NL}(s)(n) \quad \text{by the induction hypothesis} \\ &= n + 1. \end{aligned}$$

We leave it to the reader to verify that $\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Nth})(n, l)$ returns the n th element, starting from 0, of list l (if it exists); $\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Cat})$ concatenates two lists; and $\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Rev})$ reverses a list.

In the previous paragraph we referred to lists rather than tuples. Why? The real question is, what is a list? The vocabulary \mathbf{W}_{NL} was created to let us name lists of numbers, based on our intuitive understanding of the nature of lists. Our point of view is that a list of numbers is just an element of $\mathcal{T}(\text{NL})$, where $(\mathcal{I}, \mathcal{S})$ is any \mathbf{W}_{NL} -structure. This is essentially an *axiomatic* approach, where we express the properties we expect from our data objects, without specifying just what those objects are. So $\text{TUP}(N)_{\perp}$ is one set of objects that can serve as lists of numbers, but there are others. An alternative \mathbf{W}_{NL} -structure will be given by the construction preceding Theorem 5.1.

For a third example, we show that \mathcal{S} programs, defined in Chapter 2, can be incorporated into a simple data structure system. We start with type variables **Bool**, **N**, **V**, **L**, **S**, **I**, and **P**, to be assigned truth values, numbers, variables (of \mathcal{S}), labels, statements, instructions, and \mathcal{S} programs, respectively. For a constructor vocabulary we take

$$\mathbf{W}_c = \{\text{tt, ff, 0, s, var, lab, skip, incr, decr, goto, unlab_instr, lab_instr, empty, cons}\},$$

where

$$\begin{array}{ll} \tau(0) = \mathbf{N} & \tau(s) = \mathbf{N} \rightarrow \mathbf{N} \\ \tau(var) = \mathbf{N} \rightarrow \mathbf{V} & \tau(goto) = \mathbf{V} \times \mathbf{L} \rightarrow \mathbf{S} \\ \tau(lab) = \mathbf{N} \rightarrow \mathbf{L} & \tau(unlab_instr) = \mathbf{S} \rightarrow \mathbf{I} \\ \tau(skip) = \mathbf{V} \rightarrow \mathbf{S} & \tau(lab_instr) = \mathbf{L} \times \mathbf{S} \rightarrow \mathbf{I} \\ \tau(incr) = \mathbf{V} \rightarrow \mathbf{S} & \tau(empty) = \mathbf{P} \\ \tau(decr) = \mathbf{V} \rightarrow \mathbf{S} & \tau(cons) = \mathbf{I} \times \mathbf{P} \rightarrow \mathbf{P}, \end{array}$$

and we set $\mathbf{W}_{\mathcal{S}} = \mathbf{W}_c \cup B(\mathbf{W}_c)$. Now we set $\mathcal{T}_{\mathcal{S}}(\tau) = \mathcal{T}_{\mathbf{N}}(\tau)$ for $\tau = \text{Bool, N, V, L, S, I, P}$, and we set $\mathcal{T}_{\mathcal{S}}(\tau)$ to be the flat cpo on \mathcal{S} variables, labels, statements, instructions, and \mathcal{S} programs for $\tau = \mathbf{V, L, S, I, P}$, respectively. Let V_0, V_1, \dots and L_0, L_1, \dots enumerate the \mathcal{S} variables and labels, as in the beginning of Chapter 4 (except that we begin counting at 0 rather than 1). Then for the constructor symbols we define $\mathcal{I}_{\mathcal{S}}$ as follows:

$$\begin{array}{ll} \mathcal{I}_{\mathcal{S}}(0) = 0 & \mathcal{I}_{\mathcal{S}}(s) = s_{\perp} \\ \mathcal{I}_{\mathcal{S}}(var) = \text{var } \perp & \mathcal{I}_{\mathcal{S}}(goto) = \text{goto } \perp \\ \mathcal{I}_{\mathcal{S}}(lab) = \text{lab } \perp & \mathcal{I}_{\mathcal{S}}(unlab_instr) = \text{unlab_instr } \perp \\ \mathcal{I}_{\mathcal{S}}(skip) = \text{skip } \perp & \mathcal{I}_{\mathcal{S}}(lab_instr) = \text{lab_instr } \perp \\ \mathcal{I}_{\mathcal{S}}(incr) = \text{incr } \perp & \mathcal{I}_{\mathcal{S}}(empty) = \text{empty program} \\ \mathcal{I}_{\mathcal{S}}(decr) = \text{decr } \perp & \mathcal{I}_{\mathcal{S}}(cons) = \text{cons } \perp, \end{array}$$

where

$$\begin{array}{ll}
 \text{var}(n) = V_n & \text{goto}(V, L) = \text{IF } V \neq 0 \text{ GOTO } L \\
 \text{lab}(n) = L_n & \text{unlab_instr}(S) = S \\
 \text{skip}(n) = V \leftarrow V & \text{lab_instr}(L, S) = [L] \quad S \\
 \text{incr}(V) = V \leftarrow V + 1 & \\
 \text{decr}(V) = V \leftarrow V - 1 & \text{cons}(I, (I_1, \dots, I_n)) = (I, I_1, \dots, I_n).
 \end{array}$$

For example,

$$\begin{aligned}
 & \overline{\Omega}(\text{cons}(\text{lab_instr}(\text{lab}(0), \text{decr}(\text{var}(1))), \\
 & \quad \text{cons}(\text{unlab_instr}(\text{incr}(\text{var}(0)))), \\
 & \quad \text{cons}(\text{unlab_instr}(\text{goto}(\text{var}(1), \text{lab}(0))) \\
 & \quad \text{empty}))) \\
 = & \quad [A] \quad X \leftarrow X - 1 \\
 & \quad Y \leftarrow Y + 1 \\
 & \quad \text{IF } X \neq 0 \text{ GOTO } A
 \end{aligned}$$

a

$\Delta_{\mathcal{S}} = (\text{rep}(\mathcal{T}_{\mathcal{S}}), \text{rep}(\mathcal{I}_{\mathcal{S}})) = (\mathcal{T}_{\mathcal{S}}, \mathcal{I}_{\mathcal{S}})$ is the simple data structure system of \mathcal{S} programs.

One of the standard problems in programming language theory is to show the existence of structures that satisfy a given set of conditions. We can ask, for example, is there a strict data structure system for every standard vocabulary? We will show that there is. We define $\Sigma_{\mathcal{H}(W)} = (\mathcal{T}_{\mathcal{H}(W)}, \mathcal{I}_{\mathcal{H}(W)})$, the *simple Herbrand W-structure*, as follows.⁵ For each $\tau \in TV(W)$, let $\mathcal{T}_{\mathcal{H}(W)}(\tau)$ be the flat cpo on TM_W^τ . We will write \perp_τ for the bottom element of $\mathcal{T}_{\mathcal{H}(W)}(\tau)$. Next we define $\mathcal{I}_{\mathcal{H}(W)}$:

$$\mathcal{I}_{\mathcal{H}(W)}(\mathbf{c}) = \mathbf{c} \quad \text{for each constant symbol } \mathbf{c} \in W_c$$

$$\mathcal{I}_{\mathcal{H}(W)}(\mathbf{f})(\mathbf{t}_1, \dots, \mathbf{t}_n) = \begin{cases} \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n) & \text{if } \mathbf{t}_i \neq \perp_\tau, 1 \leq i \leq n \\ \perp_\tau & \text{otherwise,} \end{cases}$$

for each $\mathbf{f} \in W_c$ with $\tau(\mathbf{f}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$. For each $\mathbf{f} \in B(W_c)$, $\mathcal{I}_{\mathcal{H}(W)}(\mathbf{f})$ is defined according to conditions 4–6 on \mathcal{T} -interpretations. It is clear that $\Sigma_{\mathcal{H}(W)}$ is a simple W-structure, and an easy structural induction shows that

⁵ The idea of creating structures based on terms comes from the field of mathematical logic. For example, Herbrand *universes* are defined and play a significant role in Chapter 13.

$\bar{\Omega}(\mathbf{t}) = \mathbf{t}$ for all $\mathbf{t} \in TM_W$, so every element in $\Sigma_{\mathcal{K}(W)}$ is representable. Therefore, $\Delta_{\mathcal{K}(W)} = \text{rep}(\Sigma_{\mathcal{K}(W)}) = \Sigma_{\mathcal{K}(W)}$ is a simple data structure system for W , and we have proved

Theorem 5.1. There is a simple data structure system for every standard vocabulary.

Exercises

1. Show that any simple W -structure is complete and continuous.
2. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a simple W -structure, and let $\mathbf{f} \in W_c$ with $\tau(\mathbf{f}) = \tau \rightarrow \tau$. What is $\mu \mathcal{I}(\mathbf{f})$?
3. Show that Σ_{NL} is a simple W_{NL} -structure.
4. Show by induction on n that

$$\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Nth})(i, (m_0, \dots, m_n)) = \begin{cases} m_i & \text{if } 0 \leq i \leq n \\ \perp_N & \text{if } i > n. \end{cases}$$

5. Show by induction on n that

$$\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Cat})((l_1, \dots, l_n), (m_1, \dots, m_r)) = (l_1 \dots l_n, m_1, \dots, m_r).$$

6. Using Exercise 5, show by induction on n that

$$\mathcal{D}_{\Delta_{NL}}(\text{LIST})(\text{Rev})((m_1, \dots, m_n)) = (m_n, \dots, m_1).$$

7. Give a W_{NL} -program P such that

$$\mathcal{D}_{\Delta_{NL}}(P)(F)((m_1, \dots, m_n)) = \sum_{i=1}^n m_i.$$

8. Give the $W_{\mathcal{K}}$ -term \mathbf{t} such that $\bar{\Omega}(\mathbf{t})$ is

$$\begin{aligned} [C_1] \quad Z_2 &\leftarrow Z_3 - 1 \\ &\text{IF } Z_3 \neq 0 \text{ GOTO } C_1 \end{aligned}$$

9. Describe $\Sigma_{\mathcal{K}(W_{NL})}$.
10. Complete the proof of Theorem 5.1 by showing that $\text{rep}(\Sigma_{\mathcal{K}(W)}) = \Sigma_{\mathcal{K}(W)}$ for any standard vocabulary W .
- 11.* Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a W -structure. For any $\tau \in TV(W)$, we will say that an element $d \in D_\tau$ is *ground* if $d = \perp_\tau$ or $d = \bar{\Omega}(\mathbf{t})$ for some

$t \in TM_{W_c}$, and we will say that Σ is *term-generated* if for all $\tau \in TV(W)$, every element of D_τ is ground.

- (a) Show that there is a term-generated W -structure for every standard vocabulary W .
 - (b) Give a standard vocabulary W and a simple W -structure that is not term-generated.
 - (c) Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a simple W -structure, and let P be a W -program. Show by induction on i that for all $V \subseteq VAR_1$, all $\beta \in \mathcal{A}_\mathcal{I}(V)$ such that $\beta(X)$ is ground for all $X \in V$, and all $t \in TM_W(FV(P) \cup V)$, $\Phi_P^i(\Omega) \cup \beta(t)$ is ground. [Hint: For cases $i = 0$ and $i = k + 1$, argue by structural induction on t .]
 - (d) Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a simple W -structure. Show that $rep(\Sigma)$ is term-generated. [Hint: Use part (c) with $V = \emptyset$.]
- 12.* Let $\Sigma = (\mathcal{T}, \mathcal{I})$, $\Sigma' = (\mathcal{T}', \mathcal{I}')$ be W -structures. We say that Σ , Σ' are *isomorphic* if there is a set of functions $\{f_\tau \mid \tau \in TV(W)\}$ such that

- for all $\tau \in TV(W)$, f_τ is an isomorphism from $\mathcal{T}(\tau)$ to $\mathcal{T}'(\tau)$,
- for all constant symbols $c \in W_c$ with $\tau(c) = \tau$, $f_\tau(\mathcal{I}(c)) = \mathcal{T}'(c)$, and
- for all proper function symbols $f \in W_c$ with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$,

$$f_\tau(\mathcal{I}(f)(d_1, \dots, d_n)) = \mathcal{T}'(f)(f_{\tau_1}(d_1), \dots, f_{\tau_n}(d_n))$$

for all $(d_1, \dots, d_n) \in D_{\delta(f)}$.

[See Exercise 3.18 in Chapter 16 for the definition of *isomorphic partial orders*.]

- (a) Show that Σ_N , $\Sigma_{\mathcal{K}(W_N)}$ are isomorphic.
- (b) Show that Σ_{NL} , $\Sigma_{\mathcal{K}(W_{NL})}$ are isomorphic.
- (c) Let Σ be a simple W -structure. Use Exercise 11 to show that $rep(\Sigma)$, $\Sigma_{\mathcal{K}(W)}$ are isomorphic.

6. Infinitary Data Structure Systems

Simple data structure systems are too elementary to demonstrate the power of the framework we developed in Chapter 16, so in this section we look at more complex systems.

Definition. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous W -structure. Σ is an *infinitary* W -structure if for every proper constructor function symbol

$\mathbf{f} \in \mathbf{W}_c$, $\perp_{\rho(\mathbf{f})} \notin \text{ran } \mathcal{I}(\mathbf{f})$. $\Delta = \text{rep}(\Sigma)$ is an *infinitary data structure system* for \mathbf{W} if Σ is an infinitary \mathbf{W} -structure.

Suppose we try to define, as an interpretation $\mathcal{I}(s)$ for $s \in \mathbf{W}_N$, a continuous successor function s^∞ for N_\perp such that $\perp_N \notin \text{ran } s^\infty$. If s^∞ and s , the ordinary successor function, agree on N then we still get the positive natural numbers $s^\infty(0), s^\infty(s^\infty(0)), \dots$ by repeated application of s^∞ to 0, but we get other objects as well. Since $\perp_N \notin \text{ran } s^\infty$, we get $\perp_N \sqsubset s^\infty(\perp_N)$, which implies $s^\infty(\perp_N) \sqsubset s^\infty(s^\infty(\perp_N))$ by the monotonicity of s^∞ and by condition 2b on \mathcal{F} -interpretations. The idea is that we know that $s^\infty(\perp_N)$ is the successor of *something*, but that is all we know. Now, $s^\infty(s^\infty(\perp_N))$ is also the successor of something, but we also know that it is the successor of something that is the successor of something, so $s^\infty(s^\infty(\perp_N))$ is more defined than $s^\infty(\perp_N)$. Moreover, an object like $s^\infty(\perp_N)$ must be different than every natural number: $s^\infty(\perp_N) = 0$ would violate condition 3 on \mathcal{F} -interpretations, and $s^\infty(\perp_N) = n + 1 = s^\infty(n)$ would violate condition 2b. What we get, therefore, is an infinite chain of distinct new objects:

$$\dots \perp_N \sqsubset s^\infty(\perp_N) \sqsubset s^\infty(s^\infty(\perp_N)) \sqsubset s^\infty(s^\infty(s^\infty(\perp_N))) \sqsubset \dots$$

But then we need yet another new object $\sqcup\{(s^\infty)^i(\perp_N) \mid i \in N\}$ if we are to have a cpo.

It is not at all obvious, then, that infinitary \mathbf{W} -structures exist. In fact, we will show that they do. We begin by defining a variation on the Herbrand structures of Section 5. For each $\tau \in TV(\mathbf{W})$, we create a new constant symbol⁶ \perp_τ with $\tau(\perp_\tau) = \tau$, and we set

$$\mathbf{W}^+ = \mathbf{W}_c \cup \{\perp_\tau \mid \tau \in TV(\mathbf{W})\}.$$

For each $\tau \in TV(\mathbf{W})$ we define the ordering \sqsubseteq_{τ^+} on $TM_{\mathbf{W}^+}^\tau$ as follows:

$$t \sqsubseteq_{\tau^+} u \quad \text{if and only if} \quad \begin{cases} t = \perp_\tau \text{ or } t = u \text{ or} \\ [t = f(t_1, \dots, t_n) \text{ and } u = f(u_1, \dots, u_n), \text{ for} \\ \text{some } f \in \mathbf{W}_c \text{ with } \tau(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau, \text{ and} \\ t_i \sqsubseteq_{\tau_i^+} u_i, 1 \leq i \leq n]. \end{cases}$$

For \mathbf{W}_N we have $\mathbf{W}_N^+ = \{\text{tt}, \text{ff}, 0, s, \perp_{\text{Bool}}, \perp_N\}$, $TM_{\mathbf{W}_N^+}^{\text{Bool}} = \{\perp_{\text{Bool}}, \text{tt}, \text{ff}\}$, and

⁶ Note that the symbol \perp_τ is introduced into the *semantics* of \mathbf{W} -programs. The vocabulary \mathbf{W} remains the same. That is, \perp_τ cannot appear in a \mathbf{W} -program.

$\text{TM}_{\mathbf{W}^+}^N = \{s^i(\mathbf{0}) \mid i \in N\} \cup \{s^i(\perp_N) \mid i \in N\}$. The definition of \sqsubseteq_{N^+} implies, for example,

$$\perp_N \sqsubseteq_{N^+} s^i(\mathbf{0}) \quad \text{for all } i \in N,$$

$$\perp_N \sqsubseteq_{N^+} s(\perp_N) \sqsubseteq_{N^+} s(s(\perp_N)) \sqsubseteq_{N^+} s(s(s(\perp_N))), \text{ and}$$

$$\perp_N \sqsubseteq_{N^+} s(\perp_N) \sqsubseteq_{N^+} s(s(\perp_N)) \sqsubseteq_{N^+} s(s(s(\perp_N))) \sqsubseteq_{N^+} \dots.$$

Now, for each $\tau \in TV(\mathbf{W})$, $(\text{TM}_{\mathbf{W}^+}^\tau, \sqsubseteq_{\tau^+})$ is a partial order with bottom element \perp_τ , but we still have the problem that $(\text{TM}_{\mathbf{W}^+}^\tau, \sqsubseteq_{\tau^+})$ is not, in general, a cpo. For example, $\{s^i(\perp_N) \mid i \in N\}$ is a chain in $(\text{TM}_{\mathbf{W}^+}^N, \sqsubseteq_{N^+})$ without a least upper bound. Here is where we apply the ideal construction.

Definition. The *Herbrand ideal type assignment* for \mathbf{W} , denoted $\mathcal{T}_{\mathcal{H}^*(\mathbf{W})}$, is defined, for all $\tau \in TV(\mathbf{W})$,

$$\mathcal{T}_{\mathcal{H}^*(\mathbf{W})}(\tau) = (\text{id}(\text{TM}_{\mathbf{W}^+}^\tau), \sqsubseteq_{\text{id}(\text{TM}_{\mathbf{W}^+}^\tau)})$$

(As usual, we are writing $\text{id}(\text{TM}_{\mathbf{W}^+}^\tau)$ for $\text{id}(\text{TM}_{\mathbf{W}^+}^\tau, \sqsubseteq_{\tau^+})$.) The *Herbrand ideal $\mathcal{T}_{\mathcal{H}^*(\mathbf{W})}$ -interpretation*, denoted $\mathcal{I}_{\mathcal{H}^*(\mathbf{W})}$, is defined

- for all constant symbols $c \in \mathbf{W}_c$, with $\tau(c) = \tau$,

$$\mathcal{I}_{\mathcal{H}^*(\mathbf{W})}(c) = \{\perp_\tau, c\};$$

- for all proper function symbols $f \in \mathbf{W}_c$, with $\tau(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$,

$$\mathcal{I}_{\mathcal{H}^*(\mathbf{W})}(f)(I_1, \dots, I_n) = \{f(t_1, \dots, t_n) \mid t_i \in I_i, 1 \leq i \leq n\} \cup \{\perp_\tau\}$$

for all $(I_1, \dots, I_n) \in \text{id}(\text{TM}_{\mathbf{W}^+}^{\tau_1}) \times \dots \times \text{id}(\text{TM}_{\mathbf{W}^+}^{\tau_n})$;

- for all $f \in B(\mathbf{W}_c)$, $\mathcal{I}_{\mathcal{H}^*(\mathbf{W})}(f)$ is defined according to conditions 4–6 on \mathcal{I} -interpretations.

The *Herbrand ideal \mathbf{W} -structure*, denoted $\Sigma_{\mathcal{H}^*(\mathbf{W})}$, is $(\mathcal{T}_{\mathcal{H}^*(\mathbf{W})}, \mathcal{I}_{\mathcal{H}^*(\mathbf{W})})$.

When \mathbf{W} is understood we will write $(\mathcal{T}^\infty, \mathcal{I}^\infty)$ for $(\mathcal{T}_{\mathcal{H}^*(\mathbf{W})}, \mathcal{I}_{\mathcal{H}^*(\mathbf{W})})$, $(D_{\tau^\infty}, \sqsubseteq_{\tau^\infty})$ for $\mathcal{T}_{\mathcal{H}^*(\mathbf{W})}(\tau)$, and \perp_{τ^∞} for the bottom element of $(D_{\tau^\infty}, \sqsubseteq_{\tau^\infty})$. As usual, we will write tt for $\mathcal{I}^\infty(\mathbf{tt})$ and ff for $\mathcal{I}^\infty(\mathbf{ff})$.

For a first example we consider $\Sigma_{\mathcal{H}^*(\mathbf{W}_N)} = (\mathcal{T}_{\mathcal{H}^*(\mathbf{W}_N)}, \mathcal{I}_{\mathcal{H}^*(\mathbf{W}_N)})$, which we will write as Σ_N^∞ . There are three kinds of elements in D_{τ^∞} , including two kinds of principal ideals. For each numeral n there is $\text{pid}(n) = \{s^i(\perp_N) \mid i \leq n\} \cup \{n\}$, and for each term of the form $s^n(\perp_N)$ there is $\text{pid}(s^n(\perp_N)) = \{s^i(\perp_N) \mid i \leq n\}$. For all numerals n we will write n for $\text{pid}(n)$, e.g., $3 = \text{pid}(3)$, where $\text{pid}(n)$ is distinguished from the natural

number n by context. Somewhat ambiguously, perhaps, we will call these objects *numbers*. [Indeed, by our discussion about lists in Section 5, we could make a case that $\text{pid}(3)$ is the natural number 3.] Also, we will write n_{\perp} for $\text{pid}(s^n(\perp_N))$. These two kinds of objects look very similar, but the significant difference is that no object n can occur in an infinite chain. This is because \mathbf{n} is the greatest element of n , so n cannot be a proper subset of any larger ideal. Therefore, there is no object d distinct from n such that $n \sqsubseteq_{N^{\infty}} d$. In other words, n is not an approximation of any other element, so we can say that n is *completely defined*. On the other hand, $\{n_{\perp} \mid n \in N\}$ is an infinite chain, and

$$\begin{aligned}\sqcup\{n_{\perp} \mid n \in N\} &= \cup\{n_{\perp} \mid n \in N\} \\ &= \cup\{\text{pid}(s^n(\perp_N)) \mid n \in N\} \\ &= \{s^n(\perp_N) \mid n \in N\}.\end{aligned}$$

In fact, $\{s^n(\perp_N) \mid n \in N\}$ is the unique infinite ideal of $\mathcal{T}^{\infty}(\mathbf{N})$, and we will write it as ω . Clearly, ω is not a principal ideal.

We will now show that, for any standard vocabulary \mathbf{W} , $\Sigma_{\mathcal{K}^{\infty}(\mathbf{W})}$ is an infinitary \mathbf{W} -structure.

Lemma 1. Let $I \in D_{\tau^{\infty}}$ for some $\tau \in TV(\mathbf{W})$.

1. If $\mathbf{c} \in I$, where $\mathbf{c} \in \mathbf{W}_c$ is some constant symbol, then $I = \mathcal{I}^{\infty}(\mathbf{c})$.
2. If $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n) \in I$, where $\mathbf{f} \in \mathbf{W}_c$ is some proper function symbol, then $I = \mathcal{I}^{\infty}(\mathbf{f})(I_1, \dots, I_n)$, where, for $1 \leq i \leq n$,

$$I_i = \{\mathbf{u} \in TM_{\mathbf{W}^+}^{\tau_i} \mid \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{u}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \in I\}.$$

Proof. Let $\mathbf{c} \in I$ be some constant symbol. I is directed, so no term $\mathbf{u} \in TM_{\mathbf{W}^+}^{\tau}$ of the form \mathbf{g} or $\mathbf{g}(\mathbf{u}_1, \dots, \mathbf{u}_m)$, where $\mathbf{g} \in \mathbf{W}_c$ is distinct from \mathbf{c} , can be in I since, by definition of \sqsubseteq_{τ^+} , there is no term $\mathbf{v} \in TM_{\mathbf{W}^+}^{\tau}$ such that $\mathbf{c}, \mathbf{u} \sqsubseteq_{\tau^+} \mathbf{v}$. Also, $\perp_{\tau} \in I$ since I is downward closed, so $I = \{\perp_{\tau}, \mathbf{c}\} = \mathcal{I}^{\infty}(\mathbf{c})$.

Now, let $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n) \in I$, for some $\mathbf{f} \in \mathbf{W}_c$ with $\tau(\mathbf{f}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, and for $1 \leq i \leq n$ let I_1, \dots, I_n be as defined in the statement of the lemma. First we show that I_1, \dots, I_n are ideals. For $1 \leq i \leq n$, $\mathbf{t}_i \in I_i$, so I_i is nonempty. If $\mathbf{u} \in I_i$, $\mathbf{v} \in TM_{\mathbf{W}^+}^{\tau_i}$, and $\mathbf{v} \sqsubseteq_{\tau_i^+} \mathbf{u}$, then

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{u}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \in I$$

and

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{v}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \sqsubseteq_{\tau^+} \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{u}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n),$$

which implies $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{v}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \in I$ since I is downward closed. Therefore, $\mathbf{v} \in I_i$ and I_i is downward closed. If $\mathbf{u}, \mathbf{v} \in I_i$, there is a term $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{w}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \in I$ such that

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{u}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \sqsubseteq_{\tau^+} \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{w}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n),$$

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{v}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \sqsubseteq_{\tau^+} \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{w}, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n),$$

since I is directed. Then $\mathbf{w} \in I_i$ and $\mathbf{u}, \mathbf{v} \sqsubseteq_{\tau^+} \mathbf{w}$, so I_i is directed. Therefore, I_i is an ideal, $1 \leq i \leq n$, i.e., $I_i \in D_{\tau^*}$.

We claim that $I = \mathcal{I}^\infty(\mathbf{f})(I_1, \dots, I_n)$. Let $\mathbf{u} \in I$. If $\mathbf{u} = \perp_\tau$, then $\mathbf{u} \in \mathcal{I}^\infty(\mathbf{f})(I_1, \dots, I_n)$ by definition of $\mathcal{I}^\infty(\mathbf{f})$. Otherwise, \mathbf{u} must be of the form $\mathbf{f}(\mathbf{u}_1, \dots, \mathbf{u}_n)$ since I is directed. Then there is a term $\mathbf{f}(\mathbf{v}_1, \dots, \mathbf{v}_n) \in I$ such that

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n), \mathbf{f}(\mathbf{u}_1, \dots, \mathbf{u}_n) \sqsubseteq_{\tau^+} \mathbf{f}(\mathbf{v}_1, \dots, \mathbf{v}_n),$$

which implies, for $1 \leq i \leq n$,

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{u}_i, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \sqsubseteq_{\tau^+} \mathbf{f}(\mathbf{v}_1, \dots, \mathbf{v}_n),$$

so that

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{u}_i, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \in I.$$

Therefore, we have $\mathbf{u}_i \in I_i$, $1 \leq i \leq n$, which implies $\mathbf{f}(\mathbf{u}_1, \dots, \mathbf{u}_n) \in \mathcal{I}^\infty(\mathbf{f})(I_1, \dots, I_n)$, and so $I \subseteq \mathcal{I}^\infty(\mathbf{f})(I_1, \dots, I_n)$. Now, let $\mathbf{u} \in \mathcal{I}^\infty(\mathbf{f})(I_1, \dots, I_n)$. If $\mathbf{u} = \perp_\tau$ then $\mathbf{u} \in I$. Otherwise, \mathbf{u} is of the form $\mathbf{f}(\mathbf{u}_1, \dots, \mathbf{u}_n)$, where $\mathbf{u}_i \in I_i$, $1 \leq i \leq n$. Therefore, by definition of I_i ,

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{u}_i, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \in I, \quad 1 \leq i \leq n,$$

and since I is directed, a simple induction on n shows that there is a term $\mathbf{f}(\mathbf{v}_1, \dots, \mathbf{v}_n) \in I$ such that

$$\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, \mathbf{u}_i, \mathbf{t}_{i+1}, \dots, \mathbf{t}_n) \sqsubseteq_{\tau^+} \mathbf{f}(\mathbf{v}_1, \dots, \mathbf{v}_n), \quad 1 \leq i \leq n.$$

Then $\mathbf{u}_i \sqsubseteq_{\tau^+} \mathbf{v}_i$, $1 \leq i \leq n$, which implies $\mathbf{f}(\mathbf{u}_1, \dots, \mathbf{u}_n) \sqsubseteq_{\tau^+} \mathbf{f}(\mathbf{v}_1, \dots, \mathbf{v}_n)$, so $\mathbf{f}(\mathbf{u}_1, \dots, \mathbf{u}_n) \in I$, since I is downward closed. Therefore, $\mathcal{I}^\infty(\mathbf{f})(I_1, \dots, I_n) \subseteq I$, and so $I = \mathcal{I}^\infty(\mathbf{f})(I_1, \dots, I_n)$. ■

Theorem 6.1. $\Sigma_{\mathcal{I}^\infty(\mathbf{W})}$ is an infinitary \mathbf{W} -structure.

Proof. It is clear that $\mathcal{T}^{+\infty}$ is a type assignment for \mathbf{W} , and it is complete by Theorem 16.3.12. Therefore, we begin by showing that \mathcal{I}^∞ is a \mathcal{T}^∞ -interpretation for \mathbf{W} . Note that \perp_τ occurs in every ideal of $(\mathbf{TM}_{\mathbf{W}^+}, \sqsubseteq_{\tau^+})$, and $\{\perp_\tau\}$ is an ideal, so $\{\perp_\tau\}$ is the bottom element \perp_{τ^*} .

of $(D_{\tau^*}, \sqsubseteq_{\tau^*})$. Now, for any constant symbol $c \in W_c$ with $\tau(c) = \tau$, $\mathcal{I}^\infty(c) = \{\perp_\tau, c\} \neq \perp_{\tau^*}$ is an ideal of $(TM_{W^+}^*, \sqsubseteq_{\tau^*})$, so condition 1 is satisfied. Let $f \in W_c$ be a proper function symbol with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, let $(I_1, \dots, I_n) \in D_{\tau_1^*} \times \cdots \times D_{\tau_n^*}$, and let $I = \mathcal{I}^\infty(f)(I_1, \dots, I_n)$. I is nonempty since $\perp_\tau \in I$. Suppose that $f(t_1, \dots, t_n) \in I$, $u \in TM_{W^+}^*$, and $u \sqsubseteq_{\tau^*} f(t_1, \dots, t_n)$. If $u = \perp_\tau$ or $u = f(t_1, \dots, t_n)$ then $u \in I$. Otherwise, by the definition of \sqsubseteq_{τ^*} , u must be a term of the form $f(u_1, \dots, u_n)$, where $u_i \sqsubseteq_{\tau^*} t_i$, $1 \leq i \leq n$. Then $u_i \in I_i$, $1 \leq i \leq n$, since I_i is downward closed, so $f(u_1, \dots, u_n) \in I$ and I is downward closed. Suppose $t, u \in I$. If $t \sqsubseteq_{\tau^*} u$ or $u \sqsubseteq_{\tau^*} t$ then either $t, u \sqsubseteq_{\tau^*} t \in I$ or $t, u \sqsubseteq_{\tau^*} u \in I$, so suppose otherwise. Then t, u must be terms of the form $f(t_1, \dots, t_n), f(u_1, \dots, u_n)$, respectively, and $t_i, u_i \in I_i$ implies there is a $w_i \in I_i$ such that $t_i, u_i \sqsubseteq_{\tau^*} w_i$, $1 \leq i \leq n$, so $f(t_1, \dots, t_n), f(u_1, \dots, u_n) \sqsubseteq_{\tau^*} f(w_1, \dots, w_n) \in I$. Therefore, I is directed and it is an ideal, so $I \in D_{\tau^*}$ and condition 2a is satisfied. If $(I_1, \dots, I_n), (J_1, \dots, J_n) \in D_{\tau_1^*} \times \cdots \times D_{\tau_n^*}$ are distinct, then clearly $\mathcal{I}^\infty(f)(I_1, \dots, I_n) \neq \mathcal{I}^\infty(f)(J_1, \dots, J_n)$, so $\mathcal{I}^\infty(f)$ is one-one and condition 2b is satisfied. Conditions 2c, 3, 4, 5, and 6 follow immediately from the definition of $\mathcal{I}^\infty(f)$, so $(\mathcal{I}^\infty, \mathcal{I}^\infty)$ is a complete W -structure.

It is immediate from the definition that $\perp_{\rho(f)} \notin \text{ran } \mathcal{I}^\infty(f)$ for every proper constructor function symbol f , so it remains for us only to show that $\mathcal{I}^\infty(f)$ is continuous for every proper $f \in W$. Let $f \in W_c$ be a proper function symbol with $\tau(f) = \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, and let \mathcal{C} be a chain in the cpo $(D_{\tau_1^*} \times \cdots \times \tau_n^*, \sqsubseteq_{\tau_1^*} \times \cdots \times \tau_n^*)$. It is easy to see $\mathcal{I}^\infty(f)$ is monotonic, so by Theorem 16.4.3 we just need to show that $\mathcal{I}^\infty(f)(\sqcup \mathcal{C}) \sqsubseteq_{\tau^*} \sqcup \mathcal{I}^\infty(f)(\mathcal{C})$; that is,

$$\mathcal{I}^\infty(f)(\sqcup \mathcal{C}) \subseteq \sqcup \mathcal{I}^\infty(f)(\mathcal{C}).$$

Let $t \in \mathcal{I}^\infty(f)(\sqcup \mathcal{C}) = \mathcal{I}^\infty(f)(\cup(\mathcal{C} \downarrow 1), \dots, \cup(\mathcal{C} \downarrow n))$. If $t = \perp_\tau$ then $t \in \mathcal{I}^\infty(f)(I)$ for all $I \in \mathcal{C}$, so that $t \in \sqcup \mathcal{I}^\infty(f)(\mathcal{C})$. Otherwise, t is some term $f(t_1, \dots, t_n)$, where $t_i \in \cup(\mathcal{C} \downarrow i)$, $1 \leq i \leq n$. Now, for $1 \leq i \leq n$, if $t_i \in \cup(\mathcal{C} \downarrow i)$, then $t_i \in (I_1^i, \dots, I_n^i) \downarrow i$ for some $(I_1^i, \dots, I_n^i) \in \mathcal{C}$, and since $\{(I_1^i, \dots, I_n^i) \mid 1 \leq i \leq n\}$ is a finite subset of the chain \mathcal{C} , there is some $(I_1, \dots, I_n) \in \mathcal{C}$ such that

$$(I_1^i, \dots, I_n^i) \sqsubseteq_{\tau_1^*} \times \cdots \times \tau_n^* (I_1, \dots, I_n), \quad 1 \leq i \leq n.$$

Then $t_i \in I_i$, $1 \leq i \leq n$, which implies $f(t_1, \dots, t_n) \in \mathcal{I}^\infty(f)(I_1, \dots, I_n)$, so that $f(t_1, \dots, t_n) \in \cup \mathcal{I}^\infty(f)(\mathcal{C}) = \sqcup \mathcal{I}^\infty(f)(\mathcal{C})$. Therefore, $\mathcal{I}^\infty(f)(\sqcup \mathcal{C}) \subseteq \sqcup \mathcal{I}^\infty(f)(\mathcal{C})$.

We turn now to the built-in function symbols. Again, it is easy to see that each $\mathcal{I}^\infty(\text{if}_\tau)$, $\mathcal{I}^\infty(\text{is_f})$, and $\mathcal{I}^\infty(f_i^{-1})$ is monotonic, using Lemma 1 in the latter two cases. Let $\tau \in TV(W)$, and let \mathcal{C} be a chain in

$(D_{\text{Bool}^x \times \tau^* \times \tau^*}, \sqsubseteq_{\text{Bool}^x \times \tau^* \times \tau^*})$. If $(\sqcup \mathcal{C}) \downarrow 1 = \perp_{\text{Bool}^x}$ then $I \downarrow 1 = \perp_{\text{Bool}^x}$ for all $I \in \mathcal{C}$ and we have

$$\mathcal{I}^\infty(\mathbf{if}_\tau)(\sqcup \mathcal{C}) = \perp_{\tau^*} = \sqcup \mathcal{I}^\infty(\mathbf{if}_\tau)(\mathcal{C}).$$

If $(\sqcup \mathcal{C}) \downarrow 1 = \text{tt}$ then $\mathcal{I}^\infty(\mathbf{if}_\tau)(\sqcup \mathcal{C}) = (\sqcup \mathcal{C}) \downarrow 2 = \cup(\mathcal{C} \downarrow 2)$. Now, if $t \in \cup(\mathcal{C} \downarrow 2)$, then $t \in I$ for some $(b, I, J) \in \mathcal{C}$, and there is some $(\text{tt}, I', J') \in \mathcal{C}$ such that $I \sqsubseteq_{\tau^*} I'$, so

$$t \in I' = \mathcal{I}^\infty(\mathbf{if}_\tau)(\text{tt}, I', J') \subseteq \sqcup \mathcal{I}^\infty(\mathbf{if}_\tau)(\mathcal{C}).$$

Therefore, $\mathcal{I}^\infty(\mathbf{if}_\tau)(\sqcup \mathcal{C}) \subseteq \sqcup \mathcal{I}^\infty(\mathbf{if}_\tau)(\mathcal{C})$. Similarly, if $(\sqcup \mathcal{C}) \downarrow 1 = \text{ff}$ then $\mathcal{I}^\infty(\mathbf{if}_\tau)(\sqcup \mathcal{C}) = (\sqcup \mathcal{C}) \downarrow 3 \subseteq \sqcup \mathcal{I}^\infty(\mathbf{if}_\tau)(\mathcal{C})$, so $\mathcal{I}^\infty(\mathbf{if}_\tau)$ is continuous.

Next, let $\mathbf{f} \in \mathbf{W}_c$ with $\rho(\mathbf{f}) = \tau$, and let \mathcal{C} be a chain in $(D_{\tau^*}, \sqsubseteq_{\tau^*})$. If $\mathcal{C} = \{\perp_{\tau^*}\}$ then $\mathcal{I}^\infty(\mathbf{is_f})(\sqcup \mathcal{C}) = \perp_{\text{Bool}^x} = \sqcup \mathcal{I}^\infty(\mathbf{is_f})(\mathcal{C})$, so assume \mathcal{C} contains some $I \neq \perp_{\tau^*}$. Then there is some term in $I \subseteq \sqcup \mathcal{C}$ of the form \mathbf{g} or $\mathbf{g}(\mathbf{u}_1, \dots, \mathbf{u}_m)$, where $\mathbf{g} \in \mathbf{W}_c$, so $I, \sqcup \mathcal{C} \in \text{ran } \mathcal{I}^\infty(\mathbf{g})$ by Lemma 1. Moreover, for all $J \in \mathcal{C}$, if $J \neq \perp_{\tau^*}$ then $J \subseteq \sqcup \mathcal{C}$ implies that J also contains a term of the form \mathbf{g} or $\mathbf{g}(\mathbf{v}_1, \dots, \mathbf{v}_m)$, so $J \in \text{ran } \mathcal{I}^\infty(\mathbf{g})$. Therefore, if \mathbf{f}, \mathbf{g} are the same then $\mathcal{I}^\infty(\mathbf{is_f})(\sqcup \mathcal{C}) = \text{tt} = \sqcup \mathcal{I}^\infty(\mathbf{is_f})(\mathcal{C})$, and if they are distinct then, by condition 3 on \mathcal{T} -interpretations, $\mathcal{I}^\infty(\mathbf{is_f})(\sqcup \mathcal{C}) = \text{ff} = \sqcup \mathcal{I}^\infty(\mathbf{is_f})(\mathcal{C})$.

We conclude with the functions $\mathcal{I}^\infty(\mathbf{f}_i^{-1})$. Let $\mathbf{f} \in \mathbf{W}_c$ with $\tau(\mathbf{f}) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, and let \mathcal{C} be a chain in $(D_{\tau^*}, \sqsubseteq_{\tau^*})$. If $\sqcup \mathcal{C} \notin \text{ran } \mathcal{I}^\infty(\mathbf{f})$, then $\mathcal{I}^\infty(\mathbf{f}_i^{-1})(\sqcup \mathcal{C}) = \perp_{\tau_i^*}$, and, for any $I \in \mathcal{C}$, if there were a term $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n) \in I$, then $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n) \in \sqcup \mathcal{C}$ would imply by Lemma 1 that $\sqcup \mathcal{C} \in \text{ran } \mathcal{I}^\infty(\mathbf{f})$, so $\mathcal{I}^\infty(\mathbf{f}_i^{-1})(I) = \perp_{\tau_i^*}$ and $\sqcup \mathcal{I}^\infty(\mathbf{f}_i^{-1})(\mathcal{C}) = \perp_{\tau_i^*}$. Suppose, then, that $\sqcup \mathcal{C} = \mathcal{I}^\infty(\mathbf{f})(I_1, \dots, I_n)$ for some $(I_1, \dots, I_n) \in D_{\tau_1^*} \times \dots \times \tau_n^*$, so that $\mathcal{I}^\infty(\mathbf{f}_i^{-1})(\sqcup \mathcal{C}) = I_i$, and let $\mathbf{t} \in I_i$. Then there is a term \mathbf{u} of the form $\mathbf{f}(\mathbf{u}_1, \dots, \mathbf{u}_{i-1}, \mathbf{t}, \mathbf{u}_{i+1}, \dots, \mathbf{u}_n) \in \sqcup \mathcal{C}$, which implies $\mathbf{u} \in I$ for some $I \in \mathcal{C}$. Therefore, by Lemma 1, $\mathbf{t} \in \mathcal{I}^\infty(\mathbf{f}_i^{-1})(I) \subseteq \sqcup \mathcal{I}^\infty(\mathbf{f}_i^{-1})(\mathcal{C})$, and we have $\mathcal{I}^\infty(\mathbf{f}_i^{-1})(\sqcup \mathcal{C}) \subseteq \sqcup \mathcal{I}^\infty(\mathbf{f}_i^{-1})(\mathcal{C})$, so $\mathcal{I}^\infty(\mathbf{f}_i^{-1})$ is continuous. ■

We immediately get

Corollary 6.2. There is an infinitary data structure system for every standard vocabulary.

For example, $\Delta_N^\infty = \text{rep}(\Sigma_N^\infty)$ is an infinitary data structure system for W_N . It is clear that each element in D_{Bool^x} is representable. It turns out, moreover, that every element in D_{N^*} is representable. Certainly each

$n \in D_{N^\infty}$ is representable, since $\bar{\Omega}(n) = n$. For example,

$$\begin{aligned}\bar{\Omega}(2) &= \mathcal{I}^\infty(s)(\mathcal{I}^\infty(s)(\mathcal{I}^\infty(0))) = \mathcal{I}^\infty(s)(\mathcal{I}^\infty(s)(\{\perp_N, 0\})) \\ &= \mathcal{I}^\infty(s)(\{\perp_N, s(\perp_N), s(0)\}) \\ &= \{\perp_N, s(\perp_N), s(s(\perp_N)), s(s(0))\} \\ &= 2.\end{aligned}$$

Now, let P be the W_N -program with equations

$$\begin{aligned}B(X) &= B(X) \\ G(X) &= s(G(X)).\end{aligned}$$

Then for any $n_\perp \in D_{N^\infty}$, $n_\perp = \overline{\mu \Phi_P(s''(B(0)))}$. For example,

$$\begin{aligned}\overline{\mu \Phi_P(s(s(B(0))))} &= \mathcal{I}^\infty(s)(\mathcal{I}^\infty(s)(\Omega(B)(\mathcal{I}^\infty(0)))) \\ &= \mathcal{I}^\infty(s)(\mathcal{I}^\infty(s)(\{\perp_N\})) \\ &= \mathcal{I}^\infty(s)(\{\perp_N, s(\perp_N)\}) \\ &= \{\perp_N, s(\perp_N), s(s(\perp_N))\} \\ &= 2_\perp.\end{aligned}$$

Also, an easy induction on $n \in N$ shows that $\Phi_P^n(\Omega)(G)(0) = n_\perp$ for all $n \in N$ (where $0 = \{\perp_N, 0\}$), so

$$\begin{aligned}\overline{\mu \Phi_P(G(0))} &= \sqcup \{\overline{\Phi_P^n(\Omega)}(G(0)) \mid n \in N\} \\ &= \sqcup \{\Phi_P^n(\Omega)(G)(0) \mid n \in N\} \\ &= \sqcup \{n_\perp \mid n \in N\} \\ &= \omega.\end{aligned}$$

Therefore, every element of D_{N^∞} is representable, and $\Delta_N^\infty = \Sigma_N^\infty$.

The situation is more interesting when we consider

$$\Sigma_{NL}^\infty = (\mathcal{T}_{\mathcal{K}^\infty(W_{NL})}, \mathcal{I}_{\mathcal{K}^\infty(W_{NL})})$$

and $\Delta_{NL}^\infty = \text{rep}(\Sigma_{NL}^\infty)$. As in the previous example, $\mathcal{T}^\infty(N) = \text{rep}(\mathcal{T}^\infty)(N)$, but now $\mathcal{T}^\infty(NL) \neq \text{rep}(\mathcal{T}^\infty)(NL)$. Again there are three kinds of elements in D_{NL^∞} . For each term of the form $\text{cons}(t_1, \dots, \text{cons}(t_n, \text{nil}) \dots)$, $n \geq 0$,

where $t_i \in TM_{W_{NL}^+}^N$, $1 \leq i \leq n$, there is a principal ideal. For example,

$$\begin{aligned} \text{pid}(\text{cons}(1, \text{cons}(\perp_N, \text{nil}))) \\ = & \{\perp_{NL}, \text{cons}(\perp_N, \perp_{NL}), \text{cons}(s(\perp_N), \perp_{NL}), \text{cons}(1, \perp_{NL}), \\ & \text{cons}(\perp_N, \text{cons}(\perp_N, \perp_{NL})), \text{cons}(s(\perp_N), \text{cons}(\perp_N, \perp_{NL})), \\ & \text{cons}(1, \text{cons}(\perp_N, \perp_{NL})), \text{cons}(\perp_N, \text{cons}(\perp_N, \text{nil})), \\ & \text{cons}(s(\perp_N), \text{cons}(\perp_N, \text{nil})), \text{cons}(1, \text{cons}(\perp_N, \text{nil})))\}. \end{aligned}$$

Note that $\text{pid}(\perp_N) = \{\perp_N\}$, $\text{pid}(\text{nil}) = \{\perp_{NL}, \text{nil}\} = \mathcal{I}^\infty(\text{nil})$,

$$\begin{aligned} \text{pid}(\text{cons}(\perp_N, \text{nil})) &= \{\perp_{NL}, \text{cons}(\perp_N, \perp_{NL}), \text{cons}(\perp_N, \text{nil})\} \\ &= \mathcal{I}^\infty(\text{cons})(\text{pid}(\perp_N), \text{pid}(\text{nil})), \end{aligned}$$

and, as the reader can verify,

$$\text{pid}(\text{cons}(1, \text{cons}(\perp_N, \text{nil}))) = \mathcal{I}^\infty(\text{cons})(\text{pid}(1), \text{pid}(\text{cons}(\perp_N, \text{nil}))).$$

That is, $\text{pid}(\text{cons}(1, \text{cons}(\perp_N, \text{nil})))$ is built up from $\text{pid}(\text{nil})$, the list with no elements, by applying $\mathcal{I}^\infty(\text{cons})$ to $(\text{pid}(\perp_N), \text{pid}(\text{nil}))$ to get $\text{pid}(\text{cons}(\perp_N, \text{nil}))$, a list with one element (namely, $\text{pid}(\perp_N)$), and then applying $\mathcal{I}^\infty(\text{cons})$ again to $(\text{pid}(1), \text{pid}(\text{cons}(\perp_N, \text{nil})))$ to get a list with two elements (namely, $\text{pid}(1)$ and $\text{pid}(\perp_N)$). Therefore, we write $\text{pid}(\text{cons}(1, \text{cons}(\perp_N, \text{nil})))$ as $\langle \text{pid}(1), \text{pid}(\perp_N) \rangle$, or simply $\langle 1, \perp_{N^x} \rangle$. In general, we write $\text{pid}(\text{nil})$ as the *empty list* $\langle \rangle$ and $\text{pid}(\text{cons}(t_1, \dots, \text{cons}(t_n, \text{nil}) \dots))$ as the *finite list* $\langle d_1, \dots, d_n \rangle$, where $d_i = \text{pid}(t_i)$, $1 \leq i \leq n$.

For each term of the form $\text{cons}(t_1, \dots, \text{cons}(t_n, \perp_{NL}) \dots)$, $n \geq 0$, where $t_i \in TM_{W_{NL}^+}^N$, $1 \leq i \leq n$, there is also a principal ideal. For example, $\text{pid}(\text{cons}(1, \text{cons}(\perp_N, \perp_{NL})))$ is

$$\begin{aligned} &\{\perp_{NL}, \text{cons}(\perp_N, \perp_{NL}), \text{cons}(s(\perp_N), \perp_{NL}), \\ &\text{cons}(1, \perp_{NL}), \text{cons}(\perp_N, \text{cons}(\perp_N, \perp_{NL})), \\ &\text{cons}(s(\perp_N), \text{cons}(\perp_N, \perp_{NL})), \text{cons}(1, \text{cons}(\perp_N, \perp_{NL})))\}. \end{aligned}$$

Here we have

$$\text{pid}(\text{cons}(1, \text{cons}(\perp_N, \perp_{NL}))) = \mathcal{I}^\infty(\text{cons})(1, \mathcal{I}^\infty(\text{cons})(\perp_{N^x}, \perp_{NL^x})),$$

which we write $\langle 1, \perp_{N^x} \rangle_\perp$. In general we write elements of the form $\text{pid}(\text{cons}(t_1, \dots, \text{cons}(t_n, \perp_{NL}) \dots))$ as $\langle d_1, \dots, d_n \rangle_\perp$, where $d_i = \text{pid}(t_i)$,

$1 \leq i \leq n$. We call these objects *prefix lists*, since

$$\langle d_1, \dots, d_n \rangle_{\perp} \sqsubseteq_{\text{NL}^{\infty}} \langle d_1, \dots, d_n, e_1, \dots, e_m \rangle$$

for any $e_1, \dots, e_m \in D_{\text{NL}^{\infty}}$.

We also have elements in $D_{\text{NL}^{\infty}}$ which are nonprincipal ideals. The difference between $\langle d_1, \dots, d_n \rangle$ and $\langle d_1, \dots, d_n \rangle_{\perp}$ is that the former is built up from the completely defined empty list $\mathcal{I}^{\infty}(\text{nil})$ and the latter is built up from the completely undefined list $\perp_{\text{NL}^{\infty}}$. Now, we can have infinite chains of finite lists, e.g., $\{\langle n_{\perp} \rangle \mid n \in N\}$, where we write $\sqcup \{\langle n_{\perp} \rangle \mid n \in N\}$ as $\langle \omega \rangle$, but the least upper bound is always a finite list (though not a principal ideal). On the other hand, chains of prefix lists can lead to infinite lists. For example, $\{\langle 0, \dots, n \rangle_{\perp} \mid n \in N\}$ is a chain, since

$$\langle 0 \rangle_{\perp} \sqsubseteq_{\text{NL}^{\infty}} \langle 0, 1 \rangle_{\perp} \sqsubseteq_{\text{NL}^{\infty}} \langle 0, 1, 2 \rangle_{\perp} \sqsubseteq_{\text{NL}^{\infty}} \dots,$$

and the least upper bound of $\{\langle 0, \dots, n \rangle_{\perp} \mid n \in N\}$ is the infinite ideal which we write as the *infinite list* $\langle 0, 1, 2, 3, \dots \rangle$. There are other interesting kinds of nonprincipal ideals, such as $\sqcup \{\langle n_{\perp} \rangle_{\perp} \mid n \in N\}$ and

$$\sqcup \left\{ \overbrace{\langle n_{\perp}, \dots, n_{\perp} \rangle_{\perp}}^n \mid n \in N \right\},$$

which we leave to the reader to explore.

It is interesting to observe that even these infinite lists, or the representable ones, more precisely, can be quite useful. In fact, a family of programming languages known as *lazy functional languages* has been developed based on the use of such objects. The typical use of infinite lists in these languages is to define an infinite list of desired objects and then to select some particular object from the list. The word *lazy* refers to the fact that, in practice, it is not necessary to generate an entire infinite list before performing the selection: it is necessary to generate only *enough* of the list so that the desired object appears. For a simple example, let **P** be the program with equations

$$F(X) = \text{cons}(X, F(s(X)))$$

$$\text{Nth}(X, Y) = \text{if}_{\text{NL}}(\text{is_0}(X), \text{cons}_1^{-1}(Y), \text{Nth}(s_1^{-1}(X), \text{cons}_2^{-1}(Y))).$$

Then

$$\overline{\mu \Phi_P(F(0))} = \langle 0, 1, 2, \dots \rangle \text{ and } \overline{\mu \Phi_P(\text{Nth}(n, F(0)))} = n \text{ for all } n \in N.$$

In the next chapter we will show that the infinite list $\langle p_1, p_2, \dots \rangle$ of all primes is also representable.

Unlike Δ_N^∞ , we now have objects that are not representable. For example, the infinite list $L = \langle \text{HALT}(0,0), \text{HALT}(1,1), \dots \rangle$ is not representable, where $\text{HALT}(x, x)$ is the predicate defined in Chapter 4. If it were representable, say $L = \underline{\mu\Phi_Q(t)}$, then we would have

$$\underline{\mu\Phi_{P \cup Q}}(\text{Nth}(n, t)) = \text{HALT}(n, n) \quad \text{for all } n \in N,$$

which, informally at least, would imply that $\text{HALT}(x, x)$ is computable.

For our final example, we take a vocabulary, W_R , suitable for representing the decimal expansions of real numbers x in the interval $0 \leq x < 1$. (This interval is usually written $[0, 1)$.) This time we begin with constant symbols d_0, d_1, \dots, d_9 , with $\tau(d_0) = \dots = \tau(d_9) = D$, to represent decimal digits. We will build up lists of decimal digits with **dnil** and **dcons**, where $\tau(\text{dnil}) = \text{DL}$ and $\tau(\text{dcons}) = D \times \text{DL} \rightarrow \text{DL}$. We also include in W_R the symbols of W_{NL} . We call the elements of D_{D^∞} *decimal digits*, which we write as $\perp_{D^\infty}, 0, 1, \dots, 9$. Again we have three kinds of elements in D_{DL^∞} : (1) finite lists of decimal digits, which we write $.d_1 d_2 \dots d_n$, $n \geq 0$; (2) prefix lists of decimal digits, which we write $.d_1 d_2 \dots d_n \perp$, $n \geq 0$; and (3) infinite lists of decimal digits, which we write $.d_1 d_2 \dots$. It is clear that there is an object in D_{DL^∞} for every real number in $[0, 1)$. (Actually, there is more than one object for some real numbers, since, for example, $.29999\dots$ and $.3$ are distinct in D_{DL^∞} , but that problem will not concern us here.) We call the elements of D_{DL^∞} *computable real numbers*⁷ in $[0, 1)$. It is a basic mathematical fact that there are more real numbers than there are W_R -programs and W_R -terms, so there are certainly objects in D_{DL^∞} that are not computable real numbers. It is clear that every nonrepeating rational number in $[0, 1)$ is computable, e.g., $.33 = \overline{\Omega}(\text{dcons}(d_3, \text{dcons}(d_3, \text{dnil})))$. It is also clear, intuitively, that we could write a W_R -program to define long division, so every repeating rational number in $[0, 1)$ is computable. In the next chapter we will show, moreover, that some irrational numbers are computable as well.

Exercises

- Let $W_c = \{\text{tt}, \text{ff}, \mathbf{c}, \mathbf{f}\}$ be a standard constructor vocabulary with $\tau_c(\mathbf{c}) = \tau$ and $\tau_c(\mathbf{f}) = \tau \times \tau \rightarrow \tau$, and let $W = W_c \cup B(W_c)$.
 - Give an infinite chain in $(\text{TM}_{W^+}^\tau, \sqsubseteq_\tau)$.
 - Let $\mathcal{I}^\infty = \mathcal{I}_{W^+}^\tau$. What is $\overline{\Omega}_{\mathcal{I}^\infty}(\mathbf{f}(\mathbf{X}, \mathbf{f}(\mathbf{c}, \mathbf{c})))$?

⁷ Really we should call these objects *representable real numbers* at this point, but the operational semantics in the next chapter will justify the more traditional name.

2. What is $\text{pid}(3)$ in Δ_N^∞ ? What is $\text{pid}(s^3(\perp_N))$?
3. Show that $n_\perp \sqsubseteq_{N^\infty} n$ for all $n \in N$. [Note that n has two different meanings here.]
4. Let $\mathcal{I}^\infty = \mathcal{I}_{\mathcal{H}^\infty(W_N)}$.
 - (a) Show that $\mathcal{I}^\infty(s)(2) = 3$.
 - (b) Show that $\mathcal{I}^\infty(s)(2_\perp) = 3_\perp$.
 - (c) Show that $\mathcal{I}^\infty(s_1^{-1})(3_\perp) = 2_\perp$.
5. (a) What is $\mathcal{D}_{\Delta_N^\infty}(\text{ADD})(+)\(3, 2)$?

(b) What is $\mathcal{D}_{\Delta_N^\infty}(\text{ADD})(+)\(3_\perp, 2)$?

(c) What is $\mathcal{D}_{\Delta_N^\infty}(\text{ADD})(+)\(3, 2_\perp)$?
6. What is $\langle \perp_{N^\infty}, 1 \rangle$ in Δ_{NL}^∞ ? What is $\langle \perp_{N^\infty}, 1 \rangle_\perp$?
7. Show that $\langle n \rangle_\perp \sqsubseteq_{NL^\infty} \langle n \rangle$ for all $n \in N$.
8. Let $\mathcal{I}^\infty = \mathcal{I}_{\mathcal{H}^\infty(W_{NL})}$.
 - (a) Show that $\mathcal{I}^\infty(\text{cons})(0, \langle 1 \rangle) = \langle 0, 1 \rangle$.
 - (b) Show that $\mathcal{I}^\infty(\text{cons})(0, \langle 1 \rangle_\perp) = \langle 0, 1 \rangle_\perp$.
 - (c) Show that $\mathcal{I}^\infty(\text{cons}_2^{-1})(\langle 0, 1 \rangle_\perp) = \langle 1 \rangle_\perp$.
9. (a) What is $\mathcal{D}_{\Delta_{NL}^\infty}(\text{LIST})(\text{Length})(\langle 2, 2 \rangle)$?

(b) What is $\mathcal{D}_{\Delta_{NL}^\infty}(\text{LIST})(\text{Length})(\langle 2, 2, 2, \dots \rangle)$?
10. (a) What is $\mathcal{D}_{\Delta_{NL}^\infty}(\text{LIST})(\text{Cat})(\langle 2, 3 \rangle, \langle 4, 5 \rangle)$?

(b) What is $\mathcal{D}_{\Delta_{NL}^\infty}(\text{LIST})(\text{Cat})(\langle 2, 3 \rangle, \langle 4, 5, 6, \dots \rangle)$?
11. Write a W_{NL} -program P with $F \in FV(P)$ such that

$$\mathcal{D}_{\Delta_{NL}^\infty}(P)(F)(\langle n_1, n_2, \dots \rangle) = \langle n_1 + 1, n_2 + 1, \dots \rangle$$

for all representable lists of numbers $\langle n_1, n_2, \dots \rangle \in D_{NL^\infty}$.

12. Write a W_{NL} -program P with $F \in FV(P)$ such that
- $$\mathcal{D}_{\Delta_{NL}^\infty}(P)(F)(\langle m_1, m_2, \dots \rangle, \langle n_1, n_2, \dots \rangle) = \langle m_1 + n_1, m_2 + n_2, \dots \rangle$$
- for all representable lists of numbers $\langle m_1, m_2, \dots \rangle, \langle n_1, n_2, \dots \rangle \in D_{NL^\infty}$.
13. Show that $\langle 1, 1, 1, \dots \rangle$ is representable.
 14. Show that $\langle 1_\perp, 1_\perp, 1_\perp, \dots \rangle$ is representable.
 15. Show that for any $n \in N$, $\langle n, n + 1, n + 2, \dots \rangle$ is representable.
 16. Let $\langle \omega \rangle_\perp = \sqcup \{\langle n_\perp \rangle_\perp \mid n \in N\}$. Describe $\langle \omega \rangle_\perp$, and show that it is representable.

17. Let

$$\langle \omega, \omega, \dots \rangle = \sqcup \left\{ \overbrace{\langle n_\perp, \dots, n_\perp \rangle_\perp}^n \mid n \in N \right\}.$$

Describe $\langle \omega, \omega, \dots \rangle$, and show that it is representable.

18. Prove the assertion made in the proof of Theorem 6.1: for every $f \in W$, $\mathcal{I}^\infty(f)$ is monotonic.

18

Operational Semantics of Recursion Equations

1. Operational Semantics for Simple Data Structure Systems

The definition of \mathcal{D}_Δ accomplishes our goal of directly assigning a meaning to programs without the intermediary notion of a computation. On the other hand, at this point we have no *a priori* reason for believing that the functions defined by recursion programs are (partially) computable. It turns out, though, that they are computable in a very reasonable sense. In this chapter we go back to basics and define a notion of computation, one appropriate for recursion programs, which has much in common with computations of \mathcal{S} programs. This new kind of computation will be the basis for the operational semantics of W-programs. The idea is that the operational semantics will give us a way to compute the functions defined by the denotational semantics. We say that an operational semantics is *correct* with respect to the denotational semantics \mathcal{D}_Δ for a data structure system Δ if it gives every W-program P the same meaning given by \mathcal{D}_Δ , that is, $\mathcal{D}_\Delta(P)$. The precise details depend on the nature of the particular data structure systems in which we wish to compute, so in this section we concentrate on an operational semantics appropriate for simple data structure systems.

An \mathcal{S} program computation is a sequence of snapshots, and the relation between a snapshot and its successor is easily described. Computations of recursion programs are similar in nature. The idea, as before, is that we

treat an equation like $\mathbf{F}(\mathbf{X}) = \mathbf{G}(\mathbf{H}(\mathbf{X}))$ as a definition of \mathbf{F} in terms of $\mathbf{G}(\mathbf{H}(\mathbf{X}))$. Given a term $\mathbf{F}(3)$, for example, we attempt to determine its value by replacing it with $\mathbf{G}(\mathbf{H}(3))$. Of course, \mathbf{G} and \mathbf{H} should also be defined, so we replace them by their definitions as well, and continue replacing until we get a numeral. We formalize this idea as follows.

Again we let \mathbf{W} be some arbitrary standard vocabulary throughout the chapter. A \mathbf{W} -substitution is a finite function $\{\langle \mathbf{X}_1, \mathbf{t}_1 \rangle, \dots, \langle \mathbf{X}_n, \mathbf{t}_n \rangle\}$ such that $\mathbf{X}_1, \dots, \mathbf{X}_n$ are distinct individual variables and $\mathbf{t}_1, \dots, \mathbf{t}_n$ are \mathbf{W} -terms such that $\tau(\mathbf{t}_i) = \tau(\mathbf{X}_i)$, $1 \leq i \leq n$. The application of a \mathbf{W} -substitution θ to a \mathbf{W} -term \mathbf{t} is written $\mathbf{t}\theta$, and the result of applying θ to \mathbf{t} is the \mathbf{W} -term obtained from \mathbf{t} by simultaneously replacing each occurrence of \mathbf{X}_i by \mathbf{t}_i , $1 \leq i \leq n$. Note that each variable in the domain of θ is replaced by a term of the same type, so if \mathbf{t} is a \mathbf{W} -term, then $\mathbf{t}\theta$ is also a \mathbf{W} -term. We can give a more formal definition as follows. Let θ be a \mathbf{W} -substitution. Then

$$\begin{aligned} \mathbf{c}\theta &= \mathbf{c} && \text{for constant symbols } \mathbf{c} \in \mathbf{W} \\ \mathbf{X}\theta &= \mathbf{t} && \text{for } \mathbf{X} \in \text{VAR}_1 \text{ such that } (\mathbf{X}, \mathbf{t}) \in \theta \\ \mathbf{X}\theta &= \mathbf{X} && \text{for } \mathbf{X} \in \text{VAR}_1 \text{ such that } \mathbf{X} \notin \text{the domain of } \theta \\ \mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)\theta &= \mathbf{f}(\mathbf{t}_1\theta, \dots, \mathbf{t}_n\theta) && \text{where } \mathbf{f} \in \mathbf{W} \\ \mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)\theta &= \mathbf{F}(\mathbf{t}_1\theta, \dots, \mathbf{t}_n\theta) && \text{where } \mathbf{F} \in \text{VAR}_F. \end{aligned}$$

The following useful lemma shows that we can sometimes trade in part of a variable assignment for a substitution.

Substitution Lemma. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a \mathbf{W} -structure, let $V_i \subseteq \text{VAR}_1$ and $V_f \subseteq \text{VAR}_F$, let $\alpha \in \mathcal{A}_{\mathcal{T}}(V_f)$, and let $\beta \in \mathcal{A}_{\mathcal{T}}(V_i)$ be a variable assignment and θ a substitution such that, for all $\mathbf{X} \in V_i$, $V(\mathbf{X}\theta) \subseteq V_f$ and $\beta(\mathbf{X}) = \bar{\alpha}(\mathbf{X}\theta)$. Then for all $\mathbf{t} \in \text{TM}_{\mathbf{W}}(V_i \cup V_f)$, $\overline{\alpha \cup \beta}(\mathbf{t}) = \bar{\alpha}(\mathbf{t}\theta)$.

Proof. We argue by structural induction on \mathbf{t} . If \mathbf{t} is a constant symbol $\mathbf{c} \in \mathbf{W}$, then

$$\overline{\alpha \cup \beta}(\mathbf{c}) = \mathcal{I}(\mathbf{c}) = \mathcal{I}(\mathbf{c}\theta) = \bar{\alpha}(\mathbf{c}\theta),$$

and if \mathbf{t} is $\mathbf{X} \in V_i$, then $\overline{\alpha \cup \beta}(\mathbf{X}) = \beta(\mathbf{X}) = \bar{\alpha}(\mathbf{X}\theta)$ by assumption. If \mathbf{t} is $\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)$, where $\mathbf{f} \in \mathbf{W}$, then

$$\begin{aligned} \overline{\alpha \cup \beta}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)) &= \mathcal{I}(\mathbf{f})(\overline{\alpha \cup \beta}(\mathbf{t}_1), \dots, \overline{\alpha \cup \beta}(\mathbf{t}_n)) \\ &= \mathcal{I}(\mathbf{f})(\bar{\alpha}(\mathbf{t}_1\theta), \dots, \bar{\alpha}(\mathbf{t}_n\theta)) && \text{by the induction hypothesis} \\ &= \bar{\alpha}(\mathbf{f}(\mathbf{t}_1\theta, \dots, \mathbf{t}_n\theta)) \\ &= \bar{\alpha}(\mathbf{f}(\mathbf{t}_1, \dots, \mathbf{t}_n)\theta). \end{aligned}$$

If t is $F(t_1, \dots, t_n)$, where $F \in V_f$, then

$$\begin{aligned} & \overline{\alpha \cup \beta(F(t_1, \dots, t_n))} \\ &= \alpha(F(\overline{\alpha \cup \beta(t_1)}, \dots, \overline{\alpha \cup \beta(t_n)})) \\ &= \alpha(F(\bar{\alpha}(t_1\theta), \dots, \bar{\alpha}(t_n\theta))) \quad \text{by the induction hypothesis} \\ &= \bar{\alpha}(F(t_1\theta, \dots, t_n\theta)) \\ &= \bar{\alpha}(F(t_1, \dots, t_n)\theta). \end{aligned}$$
■

A **W-term rewrite rule** is a pair of W-terms, written $u \rightarrow v$, such that $IV(v) \subseteq IV(u)$ and such that no individual variable occurs more than once in u . A **W-term rewriting system** is a set of W-term rewrite rules.¹ We say that a W-term t *matches* a rewrite rule $u \rightarrow v$ with substitution θ if $t = u\theta$. A W-term rewriting system T is *deterministic* if no W-term matches more than one rewrite rule in T .

In order to use a W-term rewriting system T , we associate with T a *rewriting strategy* σ which selects, for every W-term t , a (possibly empty) set of occurrences² of subterms of t . These are the (T, σ) -*redexes* of t . A W-term w is a T -*rewrite* of W-term t if t matches some rewrite rule $u \rightarrow v$ in T with substitution θ , and $w = v\theta$. Given a strategy σ , a W-term w is a (T, σ) -*rewrite* of a W-term t , denoted

$$t \xrightarrow[T, \sigma]{} w,$$

if w is the result of replacing every (T, σ) -redex t' of t by a T -rewrite of t' . A W-term t is (T, σ) -*normal* if the set of (T, σ) -redexes is empty. A (T, σ) -*computation* for W-term t is a (possibly infinite) sequence of W-terms t_0, t_1, \dots such that

1. t_0 is t ,
2. $t_i \xrightarrow[T, \sigma]{} t_{i+1}$ for all t_i occurring in the sequence, and
3. for all t_i in the sequence, t_i is (T, σ) -normal if and only if t_i is the last term in the sequence.

For our definition of a computation to be reasonable, it is crucial that the process of finding the term t_{i+1} that follows term t_i should itself be “mechanical,” that is, computable in some sense. Moreover, the test that a

¹ Although the definition permits rules of the form $c \rightarrow d$, where c, d are constant symbols, they play no role in our treatment of operational semantics, so we assume that such rules do not occur in any rewriting system referred to in this chapter.

² Note that we distinguish between subterms and *occurrences* of subterms. For example, $F(G(0), G(0))$ has two occurrences of the subterm $G(0)$, and a strategy σ might select the occurrence on the left without selecting the occurrence on the right.

term is (T, σ) -normal must also be computable, so that we know when a computation terminates. Now, three kinds of steps are involved in finding t_{i+1} :

- finding the set of (T, σ) -redexes of t_i ;
- for each (T, σ) -redex of t_i , finding a rule $u \rightarrow v$ in T and a substitution θ such that t_i matches $u \rightarrow v$ with θ ; and
- applying θ to v .

It is clear that finding and applying substitutions are fairly simple operations. (For a more detailed treatment see the unification algorithm in Chapter 13.) Therefore, we must look more closely at rewriting strategies and sets of rewrite rules. We begin with four commonly defined rewriting strategies.

Definition. Let T be a W -term rewriting system. A W -term is T -*rewritable* if it matches some rule in T . Let t be a W -term. An *innermost* occurrence of a T -rewritable subterm of t is one which has no T -rewritable subterms.³ An *outermost* occurrence of a T -rewritable subterm of t is one which is not a subterm of any T -rewritable subterm of t . The *leftmost innermost* strategy, denoted σ_{li} , selects the leftmost of the innermost occurrences of T -rewritable subterms of t . If there is no such subterm then σ_{li} selects the empty set. The *parallel innermost* strategy, denoted σ_{pi} , selects the (possibly empty) set of all innermost occurrences of T -rewritable subterms of t . The *leftmost outermost* strategy, denoted σ_{lo} , selects the leftmost of the outermost occurrences of T -rewritable subterms of t , or the empty set if there is no such subterm. The *parallel outermost* strategy, denoted σ_{po} , selects the (possibly empty) set of all outermost occurrences of T -rewritable subterms of t .

Note that in each of the four strategies, the choice of (T, σ) -redexes depends on the particular set T . Therefore, the computability of applying σ depends on the nature of T .

To illustrate, let T consist of the rewrite rules

$$\begin{aligned} F(X) &\rightarrow s(F(s(X))) \\ F(X) &\rightarrow X \\ G(0, Y) &\rightarrow Y \\ G(s(X), s(Y)) &\rightarrow s(G(X, Y)). \end{aligned}$$

Then the underlined subterm of $G(\underline{s(F(0))}, s(F(0)))$ is the (T, σ_{li}) -redex, and the underlined subterm of $G(F(\underline{G(0, 0)}), F(0))$ is the (T, σ_{lo}) -redex.

³ Note that we are not considering a term to be a subterm of itself.

Each of

$$\underline{F(2)} \xrightarrow[T, \sigma_{li}]{} 2$$

$$\underline{F(2)} \xrightarrow[T, \sigma_{li}]{} s(\underline{F(3)}) \xrightarrow[T, \sigma_{li}]{} s(s(\underline{F(4)})) \xrightarrow[T, \sigma_{li}]{} s(s(4)) = 6$$

$$\underline{F(2)} \xrightarrow[T, \sigma_{li}]{} s(\underline{F(3)}) \xrightarrow[T, \sigma_{li}]{} s(s(\underline{F(4)})) \xrightarrow[T, \sigma_{li}]{} s(s(s(\underline{F(5)}))) \xrightarrow[T, \sigma_{li}]{} \dots$$

is a (T, σ_{li}) -computation for $F(2)$, where the underlined terms are the (T, σ_{li}) -redexes, and

$$\begin{aligned} \underline{G(F(F(0)), F(2))} &\xrightarrow[T, \sigma_{po}]{} \underline{G(s(F(s(F(0)))), s(F(3))))} \\ &\xrightarrow[T, \sigma_{po}]{} s(\underline{G(F(s(F(0)))), F(3))}) \\ &\xrightarrow[T, \sigma_{po}]{} s(\underline{G(s(F(0)), s(F(4))))}) \\ &\xrightarrow[T, \sigma_{po}]{} s(s(\underline{G(F(0), F(4))})) \\ &\xrightarrow[T, \sigma_{po}]{} s(s(\underline{G(0, s(F(5))))})) \\ &\xrightarrow[T, \sigma_{po}]{} s(s(s(\underline{F(5)}))) \\ &\xrightarrow[T, \sigma_{po}]{} s(s(s(5))) = 8 \end{aligned}$$

is a (T, σ_{po}) -computation for $G(F(F(0)), F(2))$, where the underlined terms

are the (T, σ_{po}) -redexes.

It is clear from the example that computations for a given term t are not necessarily unique, which would make them unsuitable for the definition of functions. However, we can associate with each W-program a deterministic W-term rewriting system, which does give unique computations.

Definition. Let P be a W-program. The *W-term rewriting system associated with P for simple data structure systems*, denoted $T_s(P)$, consists of

$F(X_1, \dots, X_n) \rightarrow u$ for each equation $F(X_1, \dots, X_n) = u$ in P ,

together with

- for each $\tau \in TV(W)$,

$$\text{if}_\tau(tt, X, Y) \rightarrow X$$

$$\text{if}_\tau(ff, X, Y) \rightarrow Y$$

(The choice of particular individual variables X, Y is unimportant, as long as they are distinct and of the appropriate type.)

- for each constant symbol $c \in W_c^-$, with $\tau(c) = \tau$,

$$\text{is_c}(c) \rightarrow \text{tt}$$

$$\text{is_c}(t) \rightarrow \text{ff} \quad \text{for each } t \in TM_{W_c}^\tau - \{c\}$$

- for each proper function symbol $f \in W_c$, with $\text{ar}(f) = n$ and $\rho(f) = \tau$,

$$\text{is_f}(f(t_1, \dots, t_n)) \rightarrow \text{tt}$$

$$\text{for each } f(t_1, \dots, t_n) \in TM_{W_c}^\tau$$

$$\text{is_f}(u) \rightarrow \text{ff}$$

$$\text{for each } u \in TM_{W_c}^\tau \text{ not of the form } f(t_1, \dots, t_n)$$

$$f_i^{-1}(f(t_1, \dots, t_n)) \rightarrow t_i$$

$$\text{for each } f(t_1, \dots, t_n) \in TM_{W_c}^\tau$$

It is clear that for any W -program P , $T_s(P)$ is deterministic, so $T_s(P)$ gives a unique $(T_s(P), \sigma)$ -computation for any W -term t , where σ is any rewriting strategy. It is also easy to see that for any W -term t , the process of finding a matching rewrite rule in $T_s(P)$, if one exists, is straightforward. It follows that finding the set of $(T_s(P), \sigma)$ -redexes for t , where σ is any of the strategies just defined, is also straightforward. Moreover, the test that a term is $(T_s(P), \sigma)$ -normal is easy, so we have four reasonable notions of a computation.

To illustrate, $T_s(\text{ADD})$ has the rules

$$+ (X, Y) \rightarrow \text{if}_N(\text{is_0}(Y), X, s(+ (X, s_1^{-1}(Y))))$$

$$\text{if}_{\text{Bool}}(\text{tt}, X, Y) \rightarrow X$$

$$\text{if}_{\text{Bool}}(\text{ff}, X, Y) \rightarrow Y$$

$$\text{if}_N(\text{tt}, X, Y) \rightarrow X$$

$$\text{if}_N(\text{ff}, X, Y) \rightarrow Y$$

$$\text{is_0}(0) \rightarrow \text{tt}$$

$$\text{is_0}(s(n)) \rightarrow \text{ff} \quad \text{for all } n \in N$$

$$\text{is_s}(s(n)) \rightarrow \text{tt} \quad \text{for all } n \in N$$

$$\text{is_s}(0) \rightarrow \text{ff}$$

$$s_1^{-1}(s(n)) \rightarrow n \quad \text{for all } n \in N,$$

and

$\underline{+} \ (3, 2)$

$$\begin{aligned}
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{\text{if}_N(\text{is_0}(2), 3, s(+ (3, s_1^{-1}(2)))))} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{\text{if}_N(\text{ff}, 3, s(+ (3, s_1^{-1}(2)))))} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(+ (3, s_1^{-1}(2)))} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(\text{if}_N(\text{is_0}(s_1^{-1}(2)), 3, s(+ (3, s_1^{-1}(s_1^{-1}(2)))))))} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(\text{if}_N(\text{is_0}(1), 3, s(+ (3, s_1^{-1}(s_1^{-1}(2)))))))} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(\text{if}_N(\text{ff}, 3, s(+ (3, s_1^{-1}(s_1^{-1}(2)))))))} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(s(+ (3, s_1^{-1}(s_1^{-1}(2)))))} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(s(\text{if}_N(\text{is_0}(s_1^{-1}(s_1^{-1}(2))), 3, s(+ (3, s_1^{-1}(s_1^{-1}(s_1^{-1}(2))))))))})} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(s(\text{if}_N(\text{is_0}(s_1^{-1}(1)), 3, s(+ (3, s_1^{-1}(s_1^{-1}(s_1^{-1}(2))))))))})} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(s(\text{if}_N(\text{is_0}(0), 3, s(+ (3, s_1^{-1}(s_1^{-1}(s_1^{-1}(2))))))))})} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(s(\text{if}_N(\text{tt}, 3, s(+ (3, s_1^{-1}(s_1^{-1}(s_1^{-1}(2))))))))})} \\
 & \xrightarrow[T_s(\text{ADD}), \sigma_{lo}]{} \underline{s(s(3)) = 5}
 \end{aligned}$$

is the $(T_s(\text{ADD}), \sigma_{lo})$ -computation for $\underline{+} \ (3, 2)$.

Now, in this example, the leftmost outermost strategy gives us exactly what we want with respect to Δ_N , computing 5 from $\underline{+} \ (3, 2)$. The leftmost innermost strategy, on the other hand, is a different story. Consider the simple program

$$P = \{G(X) = G(X), H(X) = 3\}.$$

For $G(0)$ we get the infinite $(T_s(P), \sigma_{li})$ -computation

$$\underline{G(0)} \xrightarrow[T_s(P), \sigma_{li}]{} \underline{G(0)} \xrightarrow[T_s(P), \sigma_{li}]{} \cdots,$$

which is entirely appropriate since $\mathcal{D}_{\Delta_N}(P)(G)$ is the everywhere undefined function and $\mathcal{D}_{\Delta_N}(P)(G)(0) = \perp_N$. However, σ_{li} also gives the infinite computation

$$\underline{H(G(0))} \xrightarrow[T_s(P), \sigma_{li}]{} \underline{H(G(0))} \xrightarrow[T_s(P), \sigma_{li}]{} \underline{H(G(0))} \xrightarrow[T_s(P), \sigma_{li}]{} \cdots,$$

which is not what we want, with respect to Δ_N , since

$$\begin{aligned}
 D_{\Delta_N}(\mathbf{P})(\mathbf{H})(\perp_N) &= \mu\Phi_{\mathbf{P}}(\mathbf{H})(\perp_N) \\
 &= \sqcup(\{\Phi_{\mathbf{P}}^{i+1}(\Omega)(\mathbf{H})(\perp_N) \mid i \in N\} \cup \{\Phi_{\mathbf{P}}^0(\Omega)(\mathbf{H})(\perp_N)\}) \\
 &= \sqcup(\{\overline{\Phi_{\mathbf{P}}^i(\Omega) \cup \alpha_{\perp_N}}(3) \mid i \in N\} \cup \{\perp_N\}) \\
 &= 3.
 \end{aligned}$$

The problem is that the nonstrict function assigned to \mathbf{H} by $\mathcal{D}_{\Delta_N}(\mathbf{P})$ can completely *ignore* its input and produce an output value, but the leftmost (or parallel) innermost strategy requires the computation to try forever to compute $\mathbf{G}(\mathbf{0})$. On the other hand, the leftmost (or parallel) outermost strategy gives the finite computation

$$\underline{\mathbf{H}(\mathbf{G}(\mathbf{0}))} \xrightarrow[T_s(\mathbf{P}), \sigma_{lo}]{} 3,$$

which is exactly what we want.

The point is that innermost strategies may be fine in a context where all functions are strict, but they are not successful in general. For our purposes they are not appropriate since, even if we interpret all constructor function symbols with strict functions, the interpretations of the if_τ symbols are necessarily not strict. This is quite sensible, since we do not want $\mathcal{I}(\text{if}_\tau)(b, d, e)$ to depend on both d and e , but only on (at most) one of d or e , according to the value of b . We choose a strategy which is neither purely innermost nor outermost, but which is closer in spirit to an outermost strategy, since it does not depend on completing the computation of innermost subterms. It will be convenient to define it only for deterministic \mathbf{W} -term rewriting systems.

Definition. Let \mathbf{T} be a deterministic \mathbf{W} -term rewriting system, and let \mathbf{t} be a \mathbf{W} -term. In the *full rewriting strategy*, denoted σ_f , the (\mathbf{T}, σ_f) -rewrite⁴ of \mathbf{t}

⁴ Technically, the definition of (\mathbf{T}, σ_f) -rewrites varies somewhat from the general definition of (\mathbf{T}, σ) -rewrites given earlier, since we replace subterms $\phi(t_1, \dots, t_n)$ by the \mathbf{T} -rewrite of $\phi(r r_T(t_1), \dots, r r_T(t_n))$, i.e., $r_T(\phi(r r_T(t_1), \dots, r r_T(t_n)))$, rather than by the \mathbf{T} -rewrite of $\phi(t_1, \dots, t_n)$. The difference is of no concern, however, and the definition of (\mathbf{T}, σ_f) -computations, which depends only on (\mathbf{T}, σ_f) -rewrites and (\mathbf{T}, σ_f) -normality, conforms to the general definition of (\mathbf{T}, σ) -computations.

is $\text{rr}_T(t)$, where $\text{rr}_T(t)$ is defined in two stages:

$$\text{r}_T(\phi(t_1, \dots, t_n)) = \begin{cases} \mathbf{u}\theta & \text{if } \phi(t_1, \dots, t_n) \text{ matches a rule} \\ & \phi(u_1, \dots, u_n) \rightarrow u \text{ in } T \text{ with} \\ & \text{substitution } \theta \\ \phi(t_1, \dots, t_n) & \text{otherwise} \end{cases}$$

$$\text{rr}_T(c) = c \quad \text{for all constant symbols } c \in W$$

$$\text{rr}_T(\phi(t_1, \dots, t_n)) = \text{r}_T(\phi(\text{rr}_T(t_1), \dots, \text{rr}_T(t_n))).$$

The W -term t is (T, σ_f) -normal if there is no subterm $\phi(t_1, \dots, t_n)$ of t such that $\phi(\text{rr}_T(t_1), \dots, \text{rr}_T(t_n))$ matches a rewrite rule in T .

Throughout this section we will write r_p and rr_P for $\text{r}_{T_s(P)}$ and $\text{rr}_{T_s(P)}$, respectively. Just as with σ_{li} , σ_{pi} , σ_{lo} , and σ_{po} , the computability of applying σ_f depends on the rewriting system T , and $T_s(P)$ and σ_f give us another reasonable notion of a computation. Note that if t_0, t_1, t_2, \dots is an infinite $(T_s(P), \sigma_f)$ -computation, then $\text{rr}_P^i(t_0) = t_i$ for all $i \in N$. In other words, $\text{rr}_P^0(t_0), \text{rr}_P^1(t_0), \text{rr}_P^2(t_0), \dots$ is by definition the $(T_s(P), \sigma_f)$ -computation for t_0 when the computation is infinite. If t_0, t_1, \dots, t_n is a finite $(T_s(P), \sigma_f)$ -computation, then $\text{rr}_P^i(t_0) = t_i$ for $1 \leq i \leq n$, and $\text{rr}_P^i(t_0) = t_n$ for $i > n$. That is, $\text{rr}_P(t) = t$ if t is $(T_s(P), \sigma_f)$ -normal.

It is clear that for any term $f(t_1, \dots, t_n)$, where f is a constructor symbol,

$$\text{rr}_P(f(t_1, \dots, t_n)) = r_p(f(\text{rr}_P(t_1), \dots, \text{rr}_P(t_n))) = f(\text{rr}_P(t_1), \dots, \text{rr}_P(t_n)),$$

since there are no rewrite rules in $T_s(P)$ for f . In W_N , for example, we have $\text{rr}_P(s(t)) = s(\text{rr}_P(t))$, and, in particular, $\text{rr}_P(n) = n$ for any numeral n . To illustrate σ_f we give the $(T_s(\text{ADD}), \sigma_f)$ -computation for $+(3, 2)$:

$$\begin{aligned} + (3, 2) &\xrightarrow[T_s(\text{ADD}), \sigma_f]{} \text{rr}_P(+ (3, 2)) \\ &= r_p(+ (\text{rr}_P(3), \text{rr}_P(2))) \\ &= r_p(+ (3, 2)) \\ &= \text{if}_N (\text{is_0}(2), 3, s(+ (3, s_1^{-1}(2)))) \\ \\ &\xrightarrow[T_s(\text{ADD}), \sigma_f]{} \text{rr}_P(\text{if}_N (\text{is_0}(2), 3, s(+ (3, s_1^{-1}(2)))))) \\ &= r_p(\text{if}_N (\text{rr}_P(\text{is_0}(2)), \text{rr}_P(3), \text{rr}_P(s(+ (3, s_1^{-1}(2)))))) \\ &= r_p(\text{if}_N (r_p(\text{is_0}(\text{rr}_P(2))), 3, r_p(s(\text{rr}_P(+ (3, s_1^{-1}(2))))))) \\ &= r_p(\text{if}_N (r_p(\text{is_0}(2)), 3, r_p(s(r_p(+ (\text{rr}_P(3), \text{rr}_P(s_1^{-1}(2)))))))) \\ &= r_p(\text{if}_N (\text{ff}, 3, r_p(s(r_p(+ (3, r_p(s_1^{-1}(\text{rr}_P(2)))))))))) \end{aligned}$$

$$\begin{aligned}
&= r_p(\text{if}_N(\text{ff}, 3, r_p(s(r_p(+ (3, r_p(s_1^{-1}(2)))))))) \\
&= r_p(\text{if}_N(\text{ff}, 3, r_p(s(r_p(+ (3, 1)))))) \\
&= r_p(\text{if}_N(\text{ff}, 3, r_p(s(\text{if}_N(\text{is_0}(1), 3, s(+ (3, s_1^{-1}(1)))))))) \\
&= r_p(\text{if}_N(\text{ff}, 3, s(\text{if}_N(\text{is_0}(1), 3, s(+ (3, s_1^{-1}(1))))))) \\
&= s(\text{if}_N(\text{is_0}(1), 3, s(+ (3, s_1^{-1}(1)))) \\
\\
&\xrightarrow[T_s(\text{ADD}), \sigma_f]{} rr_p(s(\text{if}_N(\text{is_0}(1), 3, s(+ (3, s_1^{-1}(1)))))) \\
&= \dots = s(s(\text{if}_N(\text{is_0}(0), 3, s(+ (3, s_1^{-1}(0)))))) \\
\\
&\xrightarrow[T_s(\text{ADD}), \sigma_f]{} rr_p(s(s(\text{if}_N(\text{is_0}(0), 3, s(+ (3, s_1^{-1}(0))))))) \\
&= \dots = 5.
\end{aligned}$$

We use $(T_s(P), \sigma_f)$ -computations to define the operational semantics of W-programs with respect to simple data structure systems. In particular, we use the final terms in finite computations to determine the functions defined by P. Moreover, the value of these terms should be independent of the denotational semantics of P, so we use the least informative variable assignment, Ω , to interpret them.

Definition. Let $\Delta = \text{rep}(\Sigma)$ be a simple data structure system for W. The *operational meaning function* for Δ , denoted \mathcal{O}_Δ , is defined as follows. For all W-programs P_0 , all $F \in FV(P_0)$, and all $(d_1, \dots, d_n) \in D_{r(\delta(F))}$,

$$\mathcal{O}_\Delta(P_0)(F)(d_1, \dots, d_n) = \begin{cases} \overline{\Omega}(t) & \text{if the } (T_s(P), \sigma_f)\text{-computation for } F(t_1, \dots, t_n) \text{ is finite and ends with } t \\ \perp_{\rho(F)} & \text{otherwise,} \end{cases}$$

where

- $d_i = \overline{\mu \Phi_{P_i}^S}(t_i)$, $1 \leq i \leq n$,
- P_0, P_1, \dots, P_n are consistent, and
- $P = \bigcup_{i=0}^n P_i$.

The idea is that we can compute $\mathcal{O}_\Delta(P_0)(F)(d_1, \dots, d_n)$ by extending program P_0 with programs P_1, \dots, P_n and then carrying out the $(T_s(\bigcup_{i=0}^n P_i), \sigma_f)$ -computation for $F(t_1, \dots, t_n)$. In Δ_N , of course, $n = \overline{\Omega}(n)$ for all $n \in N$, so to compute a function on $(m_1, \dots, m_n) \in N^n$, we can simply let $P_i = \emptyset$, $1 \leq i \leq n$. Moreover, to represent \perp_N we can just include the equation $B(X) = B(X)$.

Note, however, that in general there are many different choices of programs P_1, \dots, P_n and terms t_1, \dots, t_n which characterize the same tuple

(d_1, \dots, d_n) , and these might give different computations. It is not obvious, then, that the definition of \mathcal{O}_Δ makes sense. We need to know that we get the same *result* in all cases, even if the computations differ. Theorem 1.1 later in this section, which shows that the operational and denotational semantics are equivalent for simple data structure systems, implies that we do.

We should note that the purist might object that the operational semantics of **W**-programs is not really independent of the denotational semantics since the initial term $F(t_1, \dots, t_n)$ and the program **P** depend on the condition $d_i = \mu \Phi_P^{\Sigma}(t_i)$, $1 \leq i \leq n$. Indeed, we could give an alternative operational semantics in which the input to a program **P** is simply a sequence such as $((P_1, t_1), \dots, (P_n, t_n))$ and the output is the final term in the $(T_s(\cup_{i=0}^n P_i), \sigma_f)$ -computation for $F(t_1, \dots, t_n)$. Of course, Theorem 1.1 would need to be reformulated in a suitable way. For our purposes, however, the important thing is that there is *some* term $F(t_1, \dots, t_n)$ from which the correct value of $\mathcal{D}_\Delta(P_0)(F)(d_1, \dots, d_n)$ can be computed. That fact is sufficient to justify calling the function $\mathcal{D}_\Delta(P_0)(F)$ *computable*.

We now turn to the proof of Theorem 1.1, beginning with four lemmas. In Lemma 1 we finally apply condition 2c on \mathcal{T} -interpretations. Lemma 3, which is proved by an induction based on Lemma 2, is the heart of the argument. It shows that the terms of a computation (interpreted by Ω) correctly approximate the value of the function being computed. Lemma 4 guarantees that, if a function has a non-bottom value for some given input, then the computation for that input will eventually terminate.

Lemma 1. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a **W**-structure and let α be any variable assignment based on \mathcal{T} . Then for any term $t \in TM_{W_c}$, $\bar{\alpha}(t) \neq \perp_{\tau(t)}$.

Proof. We argue by structural induction on t . If t is a constant symbol $c \in W_c$, then $\bar{\alpha}(c) = \mathcal{I}(c) \in D_{\tau(c)} - \{\perp_{\tau(c)}\}$ by condition 1 on \mathcal{T} -interpretations. Otherwise, t is of the form $f(t_1, \dots, t_n)$, where $f \in W_c$. Then $\bar{\alpha}(f(t_1, \dots, t_n)) = \mathcal{I}(f)(\bar{\alpha}(t_1), \dots, \bar{\alpha}(t_n)) \neq \perp_{\tau(t)}$ by the induction hypothesis and condition 2c on \mathcal{T} -interpretations. ■

Lemma 2. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous **W**-structure, and let **P** be a **W**-program. Then for all $t \in TM_W(FV(P))$ and for all $i \in N$, $\Phi_P^{i+1}(\Omega)(t) = \Phi_P^i(\Omega)(rr_P(t))$.

Proof. Let $i \in N$. We argue by structural induction on t . If t is a constant symbol $c \in W$, then

$$\overline{\Phi_P^{i+1}(\Omega)(c)} = \mathcal{I}(c) = \mathcal{I}(rr_P(c)) = \overline{\Phi_P^i(\Omega)}(rr_P(c)).$$

If t is of the form $f(t_1, \dots, t_n)$, where $f \in W$, then $\text{rr}_P(f(t_1, \dots, t_n)) = r_P(f(\text{rr}_P(t_1), \dots, \text{rr}_P(t_n)))$. If $f(\text{rr}_P(t_1), \dots, \text{rr}_P(t_n))$ does not match any rewrite rule in $T_s(P)$, then $\text{rr}_P(f(t_1, \dots, t_n)) = f(\text{rr}_P(t_1), \dots, \text{rr}_P(t_n))$, and

$$\begin{aligned}
 & \overline{\Phi_P^{i+1}(\Omega)}(f(t_1, \dots, t_n)) \\
 &= \mathcal{I}(f)(\overline{\Phi_P^{i+1}(\Omega)}(t_1), \dots, \overline{\Phi_P^{i+1}(\Omega)}(t_n)) \\
 &= \mathcal{I}(f)(\overline{\Phi_P^i(\Omega)}(\text{rr}_P(t_1)), \dots, \overline{\Phi_P^i(\Omega)}(\text{rr}_P(t_n))) \quad \text{by the induction hypothesis} \\
 &= \overline{\Phi_P^i(\Omega)}(f(\text{rr}_P(t_1), \dots, \text{rr}_P(t_n))) \\
 &= \overline{\Phi_P^i(\Omega)}(\text{rr}_P(f(t_1, \dots, t_n))).
 \end{aligned}$$

Otherwise, $f(\text{rr}_P(t_1), \dots, \text{rr}_P(t_n))$ does match some rewrite rule in $T_s(P)$. Suppose t is $\text{if}_\tau(u, v, w)$. If $\text{rr}_P(u) = tt$ then

$$\text{rr}_P(\text{if}_\tau(u, v, w)) = r_P(\text{if}_\tau(\text{rr}_P(u), \text{rr}_P(v), \text{rr}_P(w))) = \text{rr}_P(v),$$

and

$$\begin{aligned}
 & \overline{\Phi_P^{i+1}(\Omega)}(\text{if}_\tau(u, v, w)) \\
 &= \mathcal{I}(\text{if}_\tau)(\overline{\Phi_P^{i+1}(\Omega)}(u), \overline{\Phi_P^{i+1}(\Omega)}(v), \overline{\Phi_P^{i+1}(\Omega)}(w)) \\
 &= \mathcal{I}(\text{if}_\tau)(\overline{\Phi_P^i(\Omega)}(\text{rr}_P(u)), \overline{\Phi_P^i(\Omega)}(\text{rr}_P(v)), \overline{\Phi_P^i(\Omega)}(\text{rr}_P(w))) \\
 & \quad \text{by the induction hypothesis} \\
 &= \mathcal{I}(\text{if}_\tau)(tt, \overline{\Phi_P^i(\Omega)}(\text{rr}_P(v)), \overline{\Phi_P^i(\Omega)}(\text{rr}_P(w))) \\
 &= \overline{\Phi_P^i(\Omega)}(\text{rr}_P(v)) \\
 &= \overline{\Phi_P^i(\Omega)}(\text{rr}_P(\text{if}_\tau(u, v, w))).
 \end{aligned}$$

Similarly, if $\text{rr}_P(u) = ff$ then

$$\overline{\Phi_P^{i+1}(\Omega)}(\text{if}_\tau(u, v, w)) = \overline{\Phi_P^i(\Omega)}(\text{rr}_P(w)) = \overline{\Phi_P^i(\Omega)}(\text{rr}_P(\text{if}_\tau(u, v, w))).$$

Next, suppose, t is $\text{is_c}(u)$ for some constant symbol $c \in W_c^-$. If $\text{rr}_P(u) = c$ then

$$\text{rr}_P(\text{is_c}(u)) = r_P(\text{is_c}(\text{rr}_P(u))) = r_P(\text{is_c}(c)) = tt,$$

and

$$\begin{aligned}
 & \overline{\Phi_p^{i+1}(\Omega)}(\mathbf{is_c}(u)) \\
 &= \mathcal{I}(\mathbf{is_c})(\overline{\Phi_p^{i+1}(\Omega)}(u)) \\
 &= \mathcal{I}(\mathbf{is_c})(\overline{\Phi_p^i(\Omega)}(\mathbf{rr}_p(u))) \quad \text{by the induction hypothesis} \\
 &= \mathcal{I}(\mathbf{is_c})(\mathcal{I}(c)) \\
 &= \mathbf{tt} \quad \text{by conditions 1 and 5 on } \mathcal{T}\text{-interpretations} \\
 &= \overline{\Phi_p^i(\Omega)}(\mathbf{rr}_p(\mathbf{is_c}(u))).
 \end{aligned}$$

Otherwise, $\mathbf{rr}_p(u) = g$ or $g(u_1, \dots, u_m) \in \text{TM}_{W_c}$ for some constant symbol or proper function symbol g distinct from c , so that

$$\mathbf{rr}_p(\mathbf{is_c}(u)) = r_p(\mathbf{is_c}(\mathbf{rr}_p(u))) = \mathbf{ff}.$$

Suppose $\mathbf{rr}_p(c) = g$. Then

$$\begin{aligned}
 & \overline{\Phi_p^{i+1}(\Omega)}(\mathbf{is_c}(u)) \\
 &= \mathcal{I}(\mathbf{is_c})(\overline{\Phi_p^{i+1}(\Omega)}(u)) \\
 &= \mathcal{I}(\mathbf{is_c})(\overline{\Phi_p^i(\Omega)}(\mathbf{rr}_p(u))) \quad \text{by the induction hypothesis} \\
 &= \mathcal{I}(\mathbf{is_c})(\mathcal{I}(g)) \\
 &= \mathbf{ff} \quad \text{by conditions 1, 3, and 5 on } \mathcal{T}\text{-interpretations} \\
 &= \overline{\Phi_p^i(\Omega)}(\mathbf{rr}_p(\mathbf{is_c}(u))).
 \end{aligned}$$

Similarly, if $\mathbf{rr}_p(u) = g(u_1, \dots, u_m)$ then

$$\begin{aligned}
 & \overline{\Phi_p^{i+1}(\Omega)}(\mathbf{is_c}(u)) \\
 &= \mathcal{I}(\mathbf{is_c})(\overline{\Phi_p^{i+1}(\Omega)}(u)) \\
 &= \mathcal{I}(\mathbf{is_c})(\overline{\Phi_p^i(\Omega)}(\mathbf{rr}_p(u))) \quad \text{by the induction hypothesis} \\
 &= \mathcal{I}(\mathbf{is_c})(\mathcal{I}(g)(\overline{\Phi_p^i(\Omega)}(u_1), \dots, \overline{\Phi_p^i(\Omega)}(u_m))) \\
 &= \mathbf{ff} \quad \text{by Lemma 1 and conditions 2c, 3, and 5 on} \\
 & \quad \mathcal{T}\text{-interpretations} \\
 &= \overline{\Phi_p^i(\Omega)}(\mathbf{rr}_p(\mathbf{is_c}(u))).
 \end{aligned}$$

The argument is nearly the same if t is $\text{is_f}(u)$, where f is a proper function symbol in \mathbf{W}_c .

Now suppose t is $f_j^{-1}(u)$ and $\text{rr}_P(u) = f(t_1, \dots, t_n) \in \text{TM}_{\mathbf{W}_c}$. Then

$$\text{rr}_P(f_j^{-1}(u)) = r_P(f_j^{-1}(\text{rr}_P(u))) = t_j,$$

and

$$\begin{aligned} & \overline{\Phi_P^{i+1}(\Omega)}(f_j^{-1}(u)) \\ &= \mathcal{I}(f_j^{-1})(\overline{\Phi_P^{i+1}(\Omega)}(u)) \\ &= \mathcal{I}(f_j^{-1})(\overline{\Phi_P^i(\Omega)}(\text{rr}_P(u))) \quad \text{by the induction hypothesis} \\ &= \mathcal{I}(f_j^{-1})(\mathcal{I}(f)(\overline{\Phi_P^i(\Omega)}(t_1), \dots, \overline{\Phi_P^i(\Omega)}(t_n))) \\ &= \overline{\Phi_P^i(\Omega)}(t_j) \quad \text{by Lemma 1 and conditions 2c on 6 on } \\ & \quad \mathcal{I}\text{-interpretations} \\ &= \overline{\Phi_P^i(\Omega)}(\text{rr}_P(f_j^{-1}(u))). \end{aligned}$$

Finally, suppose t is $F(t_1, \dots, t_n)$, where $F \in FV(P)$, and let $F(X_1, \dots, X_n) = u$ be the defining equation for F in P . Then

$$\text{rr}_P(F(t_1, \dots, t_n)) = r_P(F(\text{rr}_P(t_1), \dots, \text{rr}_P(t_n))) = u\theta,$$

where $X_j\theta = \text{rr}_P(t_j)$, $1 \leq j \leq n$, and

$$\begin{aligned} & \overline{\Phi_P^{i+1}(\Omega)}(F(t_1, \dots, t_n)) \\ &= \Phi_P^{i+1}(\Omega)(F)(\overline{\Phi_P^{i+1}(\Omega)}(t_1), \dots, \overline{\Phi_P^{i+1}(\Omega)}(t_n)) \\ &= \Phi_P^{i+1}(\Omega)(F)(\overline{\Phi_P^i(\Omega)}(\text{rr}_P(t_1)), \dots, \overline{\Phi_P^i(\Omega)}(\text{rr}_P(t_n))) \\ & \quad \text{by the induction hypothesis} \\ &= \overline{\Phi_P^i(\Omega) \cup \alpha}(u) \quad \text{where } \alpha(X_j) = \overline{\Phi_P^i(\Omega)}(\text{rr}_P(t_j)), 1 \leq j \leq n \\ &= \overline{\Phi_P^i(\Omega)}(u\theta) \quad \text{by the substitution lemma} \\ &= \overline{\Phi_P^i(\Omega)}(\text{rr}_P(F(t_1, \dots, t_n))). \end{aligned}$$
■

Lemma 3. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a complete, continuous **W**-structure, and let \mathbf{P} be a **W**-program. Then for all $t \in TM_{\mathbf{W}}(FV(\mathbf{P}))$ and for $i \in N$, $\overline{\Phi_p^i(\Omega)(t)} = \overline{\Omega}(rr_p^i(t))$.

Proof. We argue by induction on i . If $i = 0$ then for all $t \in TM_{\mathbf{W}}(FV(\mathbf{P}))$,

$$\overline{\Phi_p^0(\Omega)(t)} = \overline{\Omega}(t) = \overline{\Omega}(rr_p^0(t)),$$

so assume the lemma is true for $i = k$. Then for all $t \in TM_{\mathbf{W}}(FV(\mathbf{P}))$,

$$\begin{aligned} \overline{\Phi_p^{k+1}(\Omega)(t)} &= \overline{\Phi_p^k(\Omega)}(rr_p(t)) && \text{by Lemma 2} \\ &= \overline{\Omega}(rr_p^k(rr_p(t))) && \text{by the induction hypothesis} \\ &= \overline{\Omega}(rr_p^{k+1}(t)). && \text{(See Exercise 12.)} \end{aligned}$$

■

Lemma 4. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be a simple **W**-structure, let \mathbf{P} be a **W**-program, and let $t \in TM_{\mathbf{W}}(FV(\mathbf{P}))$. If $\overline{\Omega}(t) \neq \perp_{\tau(t)}$, then $\overline{\Omega}(rr_p(t)) = \overline{\Omega}(t)$ and $rr_p(t) \in TM_{\mathbf{W}_c}$.

Proof. Let $\tau(t) = \tau$. We argue by structural induction on t , assuming throughout that $\overline{\Omega}(t) \neq \perp_{\tau}$. If t is a constant symbol $c \in \mathbf{W}$, then $rr_p(c) = c \in TM_{\mathbf{W}_c}$. If t is $f(t_1, \dots, t_n)$, where $f \in \mathbf{W}_c$ and $\tau(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$, then $\overline{\Omega}(t_i) \neq \perp_{\tau_i}$, $1 \leq i \leq n$, by the strictness of $\mathcal{I}(f)$, so $\overline{\Omega}(rr_p(t_i)) = \overline{\Omega}(t_i)$ and $rr_p(t_i) \in TM_{\mathbf{W}_c}$ by the induction hypothesis, $1 \leq i \leq n$, and we have

$$\begin{aligned} \overline{\Omega}(f(t_1, \dots, t_n)) &= \mathcal{I}(f)(\overline{\Omega}(t_1), \dots, \overline{\Omega}(t_n)) \\ &= \mathcal{I}(f)(\overline{\Omega}(rr_p(t_1)), \dots, \overline{\Omega}(rr_p(t_n))) \\ &= \overline{\Omega}(f(rr_p(t_1), \dots, rr_p(t_n))) \\ &= \overline{\Omega}(rr_p(f(t_1, \dots, t_n))), \end{aligned}$$

where $f(t_1, \dots, t_n) \in TM_{\mathbf{W}_c}$.

Suppose t is $\mathbf{if}_{\tau}(u, v, w)$. Then $\overline{\Omega}(u) \neq \perp_{\text{Bool}}$, so $\overline{\Omega}(rr_p(u)) = \overline{\Omega}(u)$ and $rr_p(u) \in TM_{\mathbf{W}_c}$ by the induction hypothesis. If $\overline{\Omega}(u) = tt$ then $rr_p(u)$ must be tt by condition 3 on \mathcal{I} -interpretations, and $rr_p(\mathbf{if}_{\tau}(u, v, w)) = rr_p(v)$.

Therefore,

$$\begin{aligned}
 & \overline{\Omega}(\text{if}_\tau(u, v, w)) \\
 &= \mathcal{I}(\text{if}_\tau)(\overline{\Omega}(u), \overline{\Omega}(v), \overline{\Omega}(w)) \\
 &= \overline{\Omega}(v) \\
 &= \overline{\Omega}(\text{rr}_P(v)) \quad \text{by the induction hypothesis, since} \\
 & \quad \overline{\Omega}(v) = \overline{\Omega}(\text{if}_\tau(u, v, w)) \neq \perp_\tau \\
 &= \overline{\Omega}(\text{rr}_P(\text{if}_\tau(u, v, w))),
 \end{aligned}$$

and by the induction hypothesis, $\text{rr}_P(\text{if}_\tau(u, v, w)) = \text{rr}_P(v) \in TM_{W_c}$. If $\overline{\Omega}(u) = \text{ff}$ then we get $\overline{\Omega}(\text{if}_\tau(u, v, w)) = \overline{\Omega}(\text{rr}_P(\text{if}_\tau(u, v, w)))$ and $\text{rr}_P(\text{if}_\tau(u, v, w)) = \text{rr}_P(w) \in TM_{W_c}$ by a similar argument.

Next, suppose t is $\text{is_f}(u)$ for some $f \in W_c$ with $\tau(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$. If $\overline{\Omega}(\text{is_f}(u)) = \text{tt}$, then $\perp_\tau \neq \overline{\Omega}(u) \in \text{ran } \mathcal{I}(f)$, so $\overline{\Omega}(\text{rr}_P(u)) = \overline{\Omega}(u)$ and $\text{rr}_P(u) \in TM_{W_c}$ by the induction hypothesis. Moreover, $\overline{\Omega}(\text{rr}_P(u)) \in \text{ran } \mathcal{I}(f)$ implies $\text{rr}_P(u)$ must be of the form $f(t_1, \dots, t_n)$ by condition 3 on \mathcal{I} -interpretations, so $\text{rr}_P(\text{is_f}(u)) = \text{tt} \in TM_{W_c}$ and $\overline{\Omega}(\text{rr}_P(\text{is_f}(u))) = \text{tt} = \overline{\Omega}(\text{is_f}(u))$. If $\overline{\Omega}(\text{is_f}(u)) = \text{ff}$, then $\perp_\tau \neq \overline{\Omega}(u) \notin \text{ran } \mathcal{I}(f)$, so $\overline{\Omega}(\text{rr}_P(u)) = \overline{\Omega}(u)$ and $\text{rr}_P(u) \in TM_{W_c}$ by the induction hypothesis. Moreover, $\overline{\Omega}(\text{rr}_P(u)) \notin \text{ran } \mathcal{I}(f)$ implies that $\text{rr}_P(u)$ cannot be of the form $f(t_1, \dots, t_n)$, so $\text{rr}_P(\text{is_f}(u)) = \text{ff} \in TM_{W_c}$ and $\overline{\Omega}(\text{rr}_P(\text{is_f}(u))) = \text{ff} = \overline{\Omega}(\text{is_f}(u))$. The argument is similar if t is $\text{is_c}(u)$ for some constant symbol $c \in W_c^-$.

Now suppose t is $f_i^{-1}(u)$, where $\tau(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau$. Then $\perp_\tau \neq \overline{\Omega}(u) \in \text{ran } \mathcal{I}(f)$, so $\overline{\Omega}(\text{rr}_P(u)) = \overline{\Omega}(u)$ and $\text{rr}_P(u) \in TM_{W_c}$ by the induction hypothesis. Again, $\text{rr}_P(u)$ must be of the form $f(t_1, \dots, t_n)$, so $\text{rr}_P(f_i^{-1}(u)) = t_i \in TM_{W_c}$, and

$$\begin{aligned}
 \overline{\Omega}(f_i^{-1}(u)) &= \mathcal{I}(f_i^{-1})(\overline{\Omega}(u)) \\
 &= \mathcal{I}(f_i^{-1})(\overline{\Omega}(\text{rr}_P(u))) \\
 &= \mathcal{I}(f_i^{-1})(\overline{\Omega}(f(t_1, \dots, t_n))) \\
 &= \mathcal{I}(f_i^{-1})(\mathcal{I}(f)(\overline{\Omega}(t_1), \dots, \overline{\Omega}(t_n))) \\
 &= \overline{\Omega}(t_i) \quad \text{since } \mathcal{I}(f)(\overline{\Omega}(t_1), \dots, \overline{\Omega}(t_n)) \neq \perp_\tau \\
 &= \overline{\Omega}(\text{rr}_P(f_i^{-1}(u))).
 \end{aligned}$$

Finally, if t is of the form $F(t_1, \dots, t_n)$, where $F \in FV(P)$, then $\overline{\Omega}(F(t_1, \dots, t_n)) = \perp_\tau$ and there is nothing to prove in this case. ■

Theorem 1.1. Let $\Delta = \text{rep}(\Sigma)$ be a simple data structure system for \mathbf{W} . Then $\mathcal{O}_\Delta = \mathcal{D}_\Delta$.

Proof. Let \mathbf{P}_0 be a \mathbf{W} -program, let $\mathbf{F} \in FV(\mathbf{P}_0)$, and let $(d_1, \dots, d_n) \in D_{r(\delta(\mathbf{F}))}$. For $1 \leq i \leq n$ let $d_i = \overline{\mu \Phi_{\mathbf{P}_i}}(\mathbf{t}_i)$ for some \mathbf{W} -program \mathbf{P}_i and some $\mathbf{t}_i \in TM_{\mathbf{W}}(FV(\mathbf{P}_i))$ such that $\mathbf{P}_0, \mathbf{P}_1, \dots, \mathbf{P}_n$ are consistent, and let $\mathbf{P} = \bigcup_{i=0}^n \mathbf{P}_i$. Then

$$\begin{aligned} & \mathcal{D}_\Delta(\mathbf{P}_0)(\mathbf{F})(d_1, \dots, d_n) \\ &= \mathcal{D}_\Delta(\mathbf{P}_0)(\mathbf{F})(\overline{\mu \Phi_{\mathbf{P}_1}}(\mathbf{t}_1), \dots, \overline{\mu \Phi_{\mathbf{P}_n}}(\mathbf{t}_n)) \\ &= \mu \Phi_{\mathbf{P}_0}(\mathbf{F})(\overline{\mu \Phi_{\mathbf{P}_1}}(\mathbf{t}_1), \dots, \overline{\mu \Phi_{\mathbf{P}_n}}(\mathbf{t}_n)) \\ &= \mu \Phi_{\mathbf{P}}(\mathbf{F})(\overline{\mu \Phi_{\mathbf{P}}}(\mathbf{t}_1), \dots, \overline{\mu \Phi_{\mathbf{P}}}(\mathbf{t}_n)) \quad \text{by the extension lemma} \\ &= \overline{\mu \Phi_{\mathbf{P}}}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \\ &= \overline{\sqcup \{\Phi_{\mathbf{P}}^i(\Omega) \mid i \in N\}}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \\ &= \overline{\sqcup \{\overline{\Phi_{\mathbf{P}}^i(\Omega)}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \mid i \in N\}} \quad \text{by Theorem 17.2.3.} \end{aligned}$$

By Theorem 17.2.3 the set $\{\overline{\Phi_{\mathbf{P}}^i(\Omega)}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \mid i \in N\}$ is a chain in the flat cpo $(D_{\rho(\mathbf{F})}, \sqsubseteq_{\rho(\mathbf{F})})$, so if $\overline{\sqcup \{\overline{\Phi_{\mathbf{P}}^i(\Omega)}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \mid i \in N\}} \neq \perp_{\rho(\mathbf{F})}$, then $\overline{\sqcup \{\overline{\Phi_{\mathbf{P}}^i(\Omega)}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \mid i \in N\}} = \overline{\Phi_{\mathbf{P}}^{i_0}(\Omega)}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n))$ for some smallest $i_0 \in N$. Now,

$$\begin{aligned} \overline{\Phi_{\mathbf{P}}^{i_0}(\Omega)}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) &= \overline{\Omega}(\text{rr}_{\mathbf{P}}^{i_0}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n))) \quad \text{by Lemma 3} \\ &= \overline{\Omega}(\text{rr}_{\mathbf{P}}^{i_0+1}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n))) \quad \text{by Lemma 4,} \end{aligned}$$

where $\text{rr}_{\mathbf{P}}^{i_0+1}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \in TM_{\mathbf{W}_t}$, so that $\text{rr}_{\mathbf{P}}^{i_0+1}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n))$ is a $(T_s(\mathbf{P}), \sigma_f)$ -normal term, and

$$\begin{aligned} \mathcal{D}_\Delta(\mathbf{P}_0)(\mathbf{F})(d_1, \dots, d_n) &= \overline{\Omega}(\text{rr}_{\mathbf{P}}^{i_0+1}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n))) \\ &= \mathcal{O}_\Delta(\mathbf{P}_0)(\mathbf{F})(d_1, \dots, d_n). \end{aligned}$$

Otherwise, $\overline{\sqcup \{\overline{\Phi_{\mathbf{P}}^i(\Omega)}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \mid i \in N\}} = \perp_{\rho(\mathbf{F})}$, and

$$\overline{\sqcup \{\overline{\Phi_{\mathbf{P}}^i(\Omega)}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)) \mid i \in N\}} = \overline{\sqcup \{\overline{\Omega}(\text{rr}_{\mathbf{P}}^i(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n))) \mid i \in N\}}$$

by Lemma 3, so either the $(T_s(\mathbf{P}), \sigma_f)$ -computation for $\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n)$ is infinite or it ends with a $(T_s(\mathbf{P}), \sigma_f)$ -normal term $\text{rr}_{\mathbf{P}}^{i_0}(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n))$ such

that $\overline{\Omega}(\text{rr}_P^{t_0}(F(t_1, \dots, t_n))) = \perp_{\rho(F)}$, and in either case

$$\mathcal{D}_\Delta(P_0)(F)(d_1, \dots, d_n) = \perp_{\rho(F)} = \mathcal{O}_\Delta(P_0)(F)(d_1, \dots, d_n).$$

The choice of P_0 , F , and (d_1, \dots, d_n) was arbitrary, so $\mathcal{D}_\Delta = \mathcal{O}_\Delta$. ■

Thus \mathcal{O}_Δ is correct with respect to \mathcal{D}_Δ for every simple data structure system Δ . Moreover, there is no ambiguity in the definition of \mathcal{D}_Δ , so Theorem 1.1 implies that the definition of \mathcal{O}_Δ is independent of the choice of programs and terms used to denote values in Δ . Theorem 1.1 also justifies the following

Definition. Let Δ be a simple data structure system for \mathbf{W} . A function f is Δ -computable if there is a \mathbf{W} -program \mathbf{P} and $\mathbf{F} \in FV(\mathbf{P})$ such that $f = \mathcal{D}_\Delta(\mathbf{P})(\mathbf{F})$.

It follows, then, from our work in the previous chapter that $+_\perp$ is a Δ_N -computable function. We have also seen some examples of Δ_{NL} -computable functions. In the next section we will examine the Δ_N -computable functions more closely.

Exercises

- Let $\theta = \{(X_1, F(0)), (X_3, s(X_2))\}$. What is $G(s(X_2), X_3)\theta$?
- Let α be a variable assignment such that $\alpha(X) = 1$, $\alpha(Y) = 2$. Give a substitution θ such that $\overline{\mu\Phi_{ADD} \cup \alpha(+ (s(X), Y))} = \overline{\mu\Phi_{ADD} (+ (s(X), Y)\theta)}$ in Σ_N .
- We have left open the possibility of a \mathbf{W} -term rewriting system T , a strategy σ , and a \mathbf{W} -term t such that t is not (T, σ) -normal but there is no (T, σ) -rewrite of t . Verify that this situation does not occur for $T_s(\mathbf{P})$ and σ , where \mathbf{P} is any \mathbf{W} -program and σ is any of the five strategies we have defined.
- Let T be the \mathbf{W}_N -term rewriting system with rewrite rules

$$F(X, Y) \rightarrow s(Y)$$

$$F(X, Y) \rightarrow F(Y, Y),$$

and let $t = F(F(2, 3), F(4, 5))$. Give two distinct (T, σ) -computations for t , where σ is (a) σ_{li} ; (b) σ_{lo} ; (c) σ_{pi} ; (d) σ_{po} .

- Give the $(T_s(\text{ADD}), \sigma)$ -computation for $+(3, 2)$, where σ is (a) σ_{li} ; (b) σ_{pi} ; (c) σ_{po} .
- Let \mathbf{P} be a \mathbf{W} -program and let $t \in TM_{\mathbf{W}}(FV(\mathbf{P}))$. Show that for all $i \in N$, $\text{rr}_P^i(t) \in TM_{\mathbf{W}}(FV(\mathbf{P}))$.

7. Let P be the W_N -program with the equation

$$F(X) = \text{if}_N(\text{is_}0(X), 1, s(F(s_1^{-1}(X)))).$$

- (a) Describe $T_s(P)$.
 - (b) Give the $(T_s(P), \sigma_f)$ -computation for $F(2)$.
8. Let $P = \text{ADD} \cup \{B(X) = B(X)\}$, where $\tau(B) = N \rightarrow N$.
- (a) Give the $(T_s(P), \sigma_f)$ -computation for $+(3, s(s(B(0))))$.
 - (b) Give the $(T_s(P), \sigma_f)$ -computation for $+(s(s(s(B(0)))), 2)$.
9. Without using Theorem 1.1, give the value of each of the following.
- (a) $\mathcal{O}_{\Delta_N}(\text{ADD})(+)(3, 2)$.
 - (b) $\mathcal{O}_{\Delta_N}(\text{ADD})(+)(\perp_N, 2)$.
 - (c) $\mathcal{O}_{\Delta_N}(\text{ADD})(+)(3, \perp_N)$.
10. Let t be the W_{NL} -term $\text{cons}(1, \text{cons}(2, \text{nil}))$. Describe $T_s(\text{LIST})$, and give the $(T_s(\text{LIST}), \sigma_f)$ -computation for each of the following W_{NL} -terms. [LIST is defined in Section 5 of Chapter 17.]
- (a) $\text{Length}(t)$.
 - (b) $\text{Nth}(1, t)$.
 - (c) $\text{Cat}(t, t)$.
 - (d) $\text{Rev}(t)$.
11. Without using Theorem 1.1, give the value of each of the following.
- (a) $\mathcal{O}_{\Delta_{NL}}(\text{LIST})(\text{Length})(\langle 2, 3 \rangle)$.
 - (b) $\mathcal{O}_{\Delta_{NL}}(\text{LIST})(\text{Length})(\perp_{NL})$.
 - (c) $\mathcal{O}_{\Delta_{NL}}(\text{LIST})(\text{Nth})(2, \langle 2, 3, 4 \rangle)$.
 - (d) $\mathcal{O}_{\Delta_{NL}}(\text{LIST})(\text{Nth})(\perp_{NL}, \langle 2, 3, 4 \rangle)$.
 - (e) $\mathcal{O}_{\Delta_{NL}}(\text{LIST})(\text{Cat})(\langle 2, 3 \rangle, \langle 4, 5 \rangle)$.
 - (f) $\mathcal{O}_{\Delta_{NL}}(\text{LIST})(\text{Cat})(\perp_{NL}, \langle 4, 5 \rangle)$.
 - (g) $\mathcal{O}_{\Delta_{NL}}(\text{LIST})(\text{Cat})(\langle 2, 3 \rangle, \perp_{NL})$.
 - (h) $\mathcal{O}_{\Delta_{NL}}(\text{LIST})(\text{Rev})(\langle 2, 3 \rangle)$.
12. Let D be a set, let $f \in D \rightarrow D$, and let $d \in D$. Show that for all $n \in N$, $f^{n+1}(d) = f^n(f(d))$.

2. Computable Functions

Now that we have a new class of computable numeric functions, it is reasonable to compare it to the partially computable functions defined in Part 1 of the book. We will show that they are essentially the same, just as

we showed that a function is partially computable if and only if it is computable by Turing machines or Post–Turing programs. The difference here is that, technically, the two classes contain different kinds of functions, since in this chapter we have defined computable functions on N_{\perp}^n , $n \geq 1$, rather than partially computable functions on N^n . However, this distinction is easily overcome.

We begin with the primitive recursive functions.

Lemma 1. If f is primitive recursive, then f_{\perp} is Δ_N -computable.

Proof. We argue by induction on the number of compositions and recursions by which f is obtained from the initial functions. The Δ_N -computability of the initial functions is given by the following programs:

$$\mathbf{P}_s = \{\mathbf{S}(X) = s(X)\}$$

$$\mathbf{P}_n = \{\mathbf{N}(X) = \mathbf{if}_N(\mathbf{is_0}(X), 0, 0)\}$$

$$\mathbf{P}_{u_i^n} = \{\mathbf{U}_i^n(X_1, \dots, X_n) = \mathbf{if}_N(\mathbf{Def}^n(X_1, \dots, X_n), X_i, 0)\} \cup \mathbf{P}_{\mathbf{Def}^n},$$

where $\mathbf{P}_{\mathbf{Def}^1}$ is

$$\{\mathbf{Def}^1(X) = \mathbf{if}_{\text{Bool}}(\mathbf{is_0}(X), \mathbf{tt}, \mathbf{tt})\}$$

and, for $n \geq 1$, $\mathbf{P}_{\mathbf{Def}^{n+1}}$ is $\mathbf{P}_{\mathbf{Def}^n}$ together with the equation

$$\mathbf{Def}^n +^1(X_1, \dots, X_{n+1}) = \mathbf{if}_{\text{Bool}}(\mathbf{Def}^n(X_1, \dots, X_n), \mathbf{Def}^1(X_{n+1}), \mathbf{tt}).$$

Note that we include \mathbf{Def}^i , $1 \leq i \leq n$, to enforce the strictness of $\mu\Phi_{\mathbf{P}_{u_i^n}}(\mathbf{U}_i^n)$.

Now let

$$h(x_1, \dots, x_n) = f(g^1(x_1, \dots, x_n), \dots, g^k(x_1, \dots, x_n)),$$

where f, g^1, \dots, g^k are primitive recursive. By the induction hypothesis there are programs $\mathbf{P}_0, \mathbf{P}_1, \dots, \mathbf{P}_k$ with function variables $\mathbf{F}, \mathbf{G}_1, \dots, \mathbf{G}_k$ such that $f_{\perp} = \mu\Phi_{\mathbf{P}_0}(\mathbf{F})$ and $g^i_{\perp} = \mu\Phi_{\mathbf{P}_i}(\mathbf{G}_i)$, $1 \leq i \leq n$. We assume that $\mathbf{P}_0, \mathbf{P}_1, \dots, \mathbf{P}_k$ are consistent and do not contain the function variable \mathbf{H} , and we set \mathbf{P} to

$$\bigcup_{i=0}^n \mathbf{P}_i \cup \{\mathbf{H}(X_1, \dots, X_n) = \mathbf{F}(\mathbf{G}_1(X_1, \dots, X_n), \dots, \mathbf{G}_k(X_1, \dots, X_n))\}.$$

Then for any $(x_1, \dots, x_n) \in N_\perp^n$ we have

$$\begin{aligned}
 & \mu\Phi_{\mathbf{P}}(\mathbf{H})(x_1, \dots, x_n) \\
 &= \Phi_{\mathbf{P}}(\mu\Phi_{\mathbf{P}})(\mathbf{H})(x_1, \dots, x_n) \\
 &= \mu\Phi_{\mathbf{P}}(\mathbf{F})(\mu\Phi_{\mathbf{P}}(\mathbf{G}_1)(x_1, \dots, x_n), \dots, \mu\Phi_{\mathbf{P}}(\mathbf{G}_k)(x_1, \dots, x_n)) \\
 &= \mu\Phi_{\mathbf{P}_0}(\mathbf{F})(\mu\Phi_{\mathbf{P}_1}(\mathbf{G}_1)(x_1, \dots, x_n), \dots, \mu\Phi_{\mathbf{P}_k}(\mathbf{G}_k)(x_1, \dots, x_n))
 \end{aligned}$$

by the extension lemma

$$= f_\perp(g_\perp^1(x_1, \dots, x_n), \dots, g_\perp^k(x_1, \dots, x_n)).$$

If $(x_1, \dots, x_n) \in N^n$ then

$$\begin{aligned}
 & f_\perp(g_\perp^1(x_1, \dots, x_n), \dots, g_\perp^k(x_1, \dots, x_n)) \\
 &= f(g^1(x_1, \dots, x_n), \dots, g^k(x_1, \dots, x_n)) \\
 &= h(x_1, \dots, x_n),
 \end{aligned}$$

and if not then

$$f_\perp(g_\perp^1(x_1, \dots, x_n), \dots, g_\perp^k(x_1, \dots, x_n)) = \perp_N,$$

so $\mu\Phi_{\mathbf{P}}(\mathbf{H}) = h_\perp$.

Finally, let

$$\begin{aligned}
 h(x_1, \dots, x_n, 0) &= f(x_1, \dots, x_n) \\
 h(x_1, \dots, x_n, y+1) &= g(y, h(x_1, \dots, x_n, y), x_1, \dots, x_n),
 \end{aligned}$$

where g, h are primitive recursive. By the induction hypothesis there are programs $\mathbf{P}_f, \mathbf{P}_g$ with function variables \mathbf{F}, \mathbf{G} such that $\mu\Phi_{\mathbf{P}_f}(\mathbf{F}) = f_\perp$ and $\mu\Phi_{\mathbf{P}_g}(\mathbf{G}) = g_\perp$. We assume that $\mathbf{P}_f, \mathbf{P}_g$ are consistent and do not contain the function variable \mathbf{H} , and we set \mathbf{P} to $\mathbf{P}_f \cup \mathbf{P}_g \cup \{\mathbf{H}(\mathbf{X}_1, \dots, \mathbf{X}_n, \mathbf{Y}) = \mathbf{t}\}$, where \mathbf{t} is

$$\begin{aligned}
 & \text{if}_N(\text{is_0}(\mathbf{Y}), \\
 & \quad \mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n), \\
 & \quad \mathbf{G}(s_1^{-1}(\mathbf{Y}), \mathbf{H}(\mathbf{X}_1, \dots, \mathbf{X}_n, s_1^{-1}(\mathbf{Y})), \mathbf{X}_1, \dots, \mathbf{X}_n)).
 \end{aligned}$$

Let $(x_1, \dots, x_n) \in N_\perp^n$. It is clear that $\mu\Phi_{\mathbf{P}}(\mathbf{H})(x_1, \dots, x_n, \perp_N) = \perp_N$, so to conclude the proof we argue by induction on y that

$\mu\Phi_{\mathbf{P}}(\mathbf{H})(x_1, \dots, x_n, y) = h_{\perp}(x_1, \dots, x_n, y)$ for all $y \in N$. If $y = 0$ then

$$\begin{aligned}\mu\Phi_{\mathbf{P}}(\mathbf{H})(x_1, \dots, x_n, 0) &= \Phi_{\mathbf{P}}(\mu\Phi_{\mathbf{P}})(\mathbf{H})(x_1, \dots, x_n, 0) \\ &= \overline{\mu\Phi_{\mathbf{P}} \cup \alpha_{(x_1, \dots, x_n, 0)}}(\mathbf{t}) \\ &= \mu\Phi_{\mathbf{P}}(\mathbf{F})(x_1, \dots, x_n) \\ &= \mu\Phi_{\mathbf{P}_f}(\mathbf{F})(x_1, \dots, x_n) \\ &= f_{\perp}(x_1, \dots, x_n) \\ &= h_{\perp}(x_1, \dots, x_n, 0).\end{aligned}$$

Assume, now, that $\mu\Phi_{\mathbf{P}}(\mathbf{H})(x_1, \dots, x_n, y) = h_{\perp}(x_1, \dots, x_n, y)$. Then

$$\begin{aligned}\mu\Phi_{\mathbf{P}}(\mathbf{H})(x_1, \dots, x_n, y + 1) &= \overline{\mu\Phi_{\mathbf{P}} \cup \alpha_{(x_1, \dots, x_n, y + 1)}}(\mathbf{t}) \\ &= \mu\Phi_{\mathbf{P}}(\mathbf{G})(y, \mu\Phi_{\mathbf{P}}(\mathbf{H})(x_1, \dots, x_n, y), x_1, \dots, x_n) \\ &= g_{\perp}(y, h_{\perp}(x_1, \dots, x_n, y), x_1, \dots, x_n) \\ &= h_{\perp}(x_1, \dots, x_n, y + 1).\end{aligned}$$
■

Theorem 2.1. If f is partially computable, then f_{\perp} is Δ_N -computable.

Proof. Let f be a partially computable n -ary function. By Theorem 3.3 in Chapter 4, there is a primitive recursive predicate $R(x_1, \dots, x_n, y)$ such that

$$f(x_1, \dots, x_n) = l(\min_z R(x_1, \dots, x_n, z)),$$

and by Lemma 1 there are \mathbf{W}_N -programs $\mathbf{P}_R, \mathbf{P}_l$ with function variables \mathbf{R} and \mathbf{L} such that $\mu\Phi_{\mathbf{P}_R}(\mathbf{R}) = R_{\perp}$ and $\mu\Phi_{\mathbf{P}_l}(\mathbf{L}) = l_{\perp}$. We assume that $\mathbf{P}_R, \mathbf{P}_l$ are consistent and do not include function variables \mathbf{F}, \mathbf{G} , and we set \mathbf{P} to $\mathbf{P}_R \cup \mathbf{P}_l$ together with the equations

$$\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{L}(\mathbf{G}(\mathbf{X}_1, \dots, \mathbf{X}_n, \mathbf{0}))$$

$$\mathbf{G}(\mathbf{X}_1, \dots, \mathbf{X}_n, \mathbf{Y}) = \mathbf{if}_{\mathbf{N}}(\mathbf{is_s}(R(\mathbf{X}_1, \dots, \mathbf{X}_n, \mathbf{Y})), \mathbf{Y}, \mathbf{G}(\mathbf{X}_1, \dots, \mathbf{X}_n, \mathbf{s}(\mathbf{Y}))).$$

Let $(x_1, \dots, x_n) \in N_\perp^n$. It is clear that, for all $y \in N$,

$$\mu\Phi_p(\mathbf{G})(x_1, \dots, x_n, y)$$

$$= \begin{cases} \perp_N & \text{if } (x_1, \dots, x_n, y) \notin N^{n+1} \\ y & \text{if } R(x_1, \dots, x_n, y) \\ \mu\Phi_p(\mathbf{G})(x_1, \dots, x_n, y + 1) & \text{otherwise,} \end{cases}$$

and so

$$\mu\Phi_p(\mathbf{G})(x_1, \dots, x_n, 0)$$

$$= \begin{cases} \perp_N & \text{if } (x_1, \dots, x_n, 0) \notin N^{n+1} \\ \perp_N & \text{if } f(x_1, \dots, x_n) \uparrow \\ \min_z R(x_1, \dots, x_n, z) & \text{otherwise.} \end{cases}$$

Therefore, if $(x_1, \dots, x_n) \in N^n$ and $f(x_1, \dots, x_n) \downarrow$, then

$$\begin{aligned} \mu\Phi_p(\mathbf{F})(x_1, \dots, x_n) &= l_\perp(\min_z R(x_1, \dots, x_n, z)) \\ &= l(\min_z R(x_1, \dots, x_n, z)) \\ &= f(x_1, \dots, x_n), \end{aligned}$$

and

$$\mu\Phi_p(\mathbf{F})(x_1, \dots, x_n) = l_\perp(\perp_N) = \perp_N,$$

otherwise, so that

$$\mu\Phi_p(\mathbf{F})(x_1, \dots, x_n) = f_\perp(x_1, \dots, x_n)$$

for all $(x_1, \dots, x_n) \in N_\perp^n$, i.e., $\mu\Phi_p(\mathbf{F}) = f_\perp$. ■

To prove a result in the other direction, we need a way of going from functions on N_\perp^n to partial functions on N^n .

Definition. For any function $f: N_\perp^n \rightarrow N_\perp$, let f_π be the partial n -ary function on N^n defined

$$f_\pi(x_1, \dots, x_n) = \begin{cases} f(x_1, \dots, x_n) & \text{if } f(x_1, \dots, x_n) \neq \perp_N \\ \uparrow & \text{otherwise.} \end{cases}$$

Theorem 2.2. If f is Δ_N -computable, then f_π is partially computable.

Proof. The proof is similar to the proof in Chapter 4 that the $\text{STP}^{(n)}$ predicates are primitive recursive. We encode \mathbf{W}_N -terms, \mathbf{W}_N -programs, and \mathbf{W}_N -substitutions as numbers and give a numeric version of the rr_P

functions. All \mathbf{W}_N -terms are words in the 20 symbol alphabet $A \cup \mathbf{W}_N$, so for any word $w \in (A \cup \mathbf{W}_N)^*$, we let $\#_{20}(w)$ be the numeric value of w in base 20 notation, as defined in Chapter 5. We will use $\#_{20}$ to encode variables and function symbols. We encode each \mathbf{W}_N -term t as $\#(t)$, where

$$\begin{aligned} \#(\phi) &= \langle \#_{20}(\phi), 0 \rangle && \text{if } \phi \text{ is a constant symbol or individual variable} \\ \#(\phi(t_1, \dots, t_n)) &= \langle \#_{20}(\phi), [\#(t_1), \dots, \#(t_n)] \rangle && \text{if } \phi \text{ is a proper function symbol or function variable.} \end{aligned}$$

If \mathbf{P} is a \mathbf{W}_N -program

$$\{F_1(X_1^1, \dots, X_{n_1}^1) = t_1, \dots, F_m(X_1^m, \dots, X_{n_m}^m) = t_m\},$$

then we associate with \mathbf{P} the finite numeric function

$$\phi_{\mathbf{P}} = \left\{ (\#_{20}(F_i), \langle [\#(X_1^i), \dots, \#(X_{n_i}^i)], \#(t_i) \rangle) \mid 1 \leq i \leq m \right\},$$

and if θ is a \mathbf{W}_N -substitution $\{(X_1, t_1), \dots, (X_n, t_n)\}$, then we associate with θ the finite numeric function

$$\phi_{\theta} = \{(\#(X_1), \#(t_1)), \dots, (\#(X_n), \#(t_n))\}.$$

We encode any finite numeric function $\phi = \{(x_1, y_1), \dots, (x_n, y_n)\}$ as

$$\tilde{\phi} = \prod_{i=1}^n p_{x_i}^{y_i+1},$$

and we set $\#(\mathbf{P}) = \tilde{\phi}_{\mathbf{P}}$ and $\#(\theta) = \tilde{\phi}_{\theta}$ for any \mathbf{W}_N -program \mathbf{P} and \mathbf{W}_N -substitution θ .

Now we define some numeric functions for handling \mathbf{W}_N -terms, their values, and their encoding numbers:

$$\text{NUM}(0) = \#(\mathbf{0})$$

$$\text{NUM}(x + 1) = \langle \#_{20}(s), [\text{NUM}(x)] \rangle$$

$$\text{TERM}(z, x_1, \dots, x_n) = \langle z, [\text{NUM}(x_1), \dots, \text{NUM}(x_n)] \rangle$$

$$\text{EVAL}(x) = \begin{cases} 0 & \text{if } x = \#(\mathbf{0}) \\ \uparrow & \text{if } l(x) = \#_{20}(s) \\ \text{EVAL}((r(x))_1) + 1 & \text{otherwise} \end{cases}$$

$$\text{IS_NUM}(x) = \begin{cases} 1 & \text{if } x = \#(\mathbf{0}) \\ 0 & \text{if } l(x) = \#_{20}(s) \\ \uparrow & \text{otherwise.} \end{cases}$$

It is clear that $\text{NUM}(n) = \#\mathbf{n}$ and $\text{EVAL}(\#\mathbf{n}) = n$ for any $n \in N$; $\text{TERM}(\#_{20}(\mathbf{F}), m_1, \dots, m_n) = \#(\mathbf{F}(m_1, \dots, m_n))$ for any function variable \mathbf{F} ; and $\text{IS_NUM}(x)$ is the predicate that tests whether $x = \#\mathbf{n}$ for some numeral \mathbf{n} .

Next we define two functions for handling \mathbf{W}_N -substitutions:

$$\begin{aligned} \text{MAKE_SUB}(y, t) &= \prod_{i=1}^{\text{Lt}(r(t))} p_i^{(r(t))_i + 1}_{l((y)_{l(t)} - 1)_i} \\ \text{APPLY_SUB}(s, t) &= \begin{cases} (s)_i \doteq 1 & \text{if } r(t) = 0 \text{ and } (s)_i \neq 0 \\ t & \text{if } r(t) = 0 \text{ and } (s)_i = 0 \\ \langle l(t), \prod_{i=1}^{\text{Lt}(r(t))} p_i^{\text{APPLY_SUB}(s, (r(t))_i)} \rangle & \text{otherwise.} \end{cases} \end{aligned}$$

If \mathbf{P} is a \mathbf{W}_N -program and $\mathbf{F}(\mathbf{X}_1, \dots, \mathbf{X}_n) = \mathbf{u}$ is an equation in \mathbf{P} , then

$$\text{MAKE_SUB}(\#(\mathbf{P}), \#(\mathbf{F}(\mathbf{t}_1, \dots, \mathbf{t}_n))) = \#(\{(\mathbf{X}_1, \mathbf{t}_1), \dots, (\mathbf{X}_n, \mathbf{t}_n)\}).$$

Also, if θ is a \mathbf{W}_N -substitution and \mathbf{t} is a \mathbf{W}_N -term, then

$$\text{APPLY_SUB}(\#(\theta), \#(\mathbf{t})) = \#(\mathbf{t}\theta).$$

Finally, we define some functions for handling $(\mathbf{T}_s(\mathbf{P}), \sigma_f)$ -rewriting:

$$\begin{aligned} \text{APPLY_SUB}(\text{MAKE_SUB}(y, t), r((y)_{l(t)} - 1)) \\ \text{if } (y)_{l(t)} \neq 0 \\ (r(t))_2 &\quad \text{if } l(t) = \#_{20}(\mathbf{if}_{\mathbf{BOOL}}) \text{ and } (r(t))_1 = \#(\mathbf{tt}) \\ (r(t))_3 &\quad \text{if } l(t) = \#_{20}(\mathbf{if}_{\mathbf{BOOL}}) \text{ and } (r(t))_1 = \#(\mathbf{ff}) \\ (r(t))_2 &\quad \text{if } l(t) = \#_{20}(\mathbf{if}_N) \text{ and } (r(t))_1 = \#(\mathbf{tt}) \\ (r(t))_3 &\quad \text{if } l(t) = \#_{20}(\mathbf{if}_N) \text{ and } (r(t))_1 = \#(\mathbf{ff}) \\ \#(\mathbf{tt}) &\quad \text{if } l(t) = \#_{20}(\mathbf{is_0}) \text{ and } (r(t))_1 = \#(\mathbf{0}) \\ \#(\mathbf{ff}) &\quad \text{if } l(t) = \#_{20}(\mathbf{is_0}) \text{ and } l((r(t))_1) = \#_{20}(\mathbf{s}) \\ &\quad \text{and } \text{IS_NUM}(t) \\ \#(\mathbf{tt}) &\quad \text{if } l(t) = \#_{20}(\mathbf{is_s}) \text{ and } l((r(t))_1) = \#_{20}(\mathbf{s}) \\ &\quad \text{and } \text{IS_NUM}(t) \\ \#(\mathbf{ff}) &\quad \text{if } l(t) = \#_{20}(\mathbf{is_s}) \text{ and } (r(t))_1 = \#(\mathbf{0}) \\ (r((r(t))_1))_1 &\quad \text{if } l(t) = \#_{20}(\mathbf{s}_1^{-1}) \text{ and } l((r(t))_1) = \#_{20}(\mathbf{s}) \\ &\quad \text{and } \text{IS_NUM}(t) \\ t &\quad \text{otherwise} \end{aligned}$$

$$\text{RRP}(y, t) = \text{RP}\left(y, \langle l(t), \prod_{i=1}^{\text{Lt}(r(t))} p_i^{\text{RRP}(y, (r(t))_i)} \rangle\right)$$

$$\text{RRP}^*(y, t, 0) = t$$

$$\text{RRP}^*(y, t, r + 1) = \text{RRP}(y, \text{RRP}^*(y, t, r))$$

$$\text{RRPT}^*(y, z, x_1, \dots, x_n, r) = \text{RRP}^*(y, \text{TERM}(z, x_1, \dots, x_n), r)$$

$$\begin{aligned} \text{END}(y, z, x_1, \dots, x_n) &= \min_r [\text{RRPT}^*(y, z, x_1, \dots, x_n, r) \\ &= \text{RRPT}^*(y, z, x_1, \dots, x_n, r + 1)]. \end{aligned}$$

If \mathbf{P} is a \mathbf{W}_N -program and t is a \mathbf{W}_N -term, then $\text{RP}(\#(\mathbf{P}), \#(t)) = \#(r_p(t))$, $\text{RRP}(\#(\mathbf{P}), \#(t)) = \#(rr_p(t))$, and $\text{RRP}^*(\#(\mathbf{P}), \#(t), i) = \#(rr_p^i(t))$. Also, if F is a function variable and $(m_1, \dots, m_n) \in N^n$, then

$$\text{RRPT}^*(\#(\mathbf{P}), \#_{20}(F), m_1, \dots, m_n, i) = \#(rr_p^i(F(m_1, \dots, m_n))),$$

and $\text{END}(\#(\mathbf{P}), \#_{20}(F), m_1, \dots, m_n)$ is the smallest i such that $rr_p^i(F(m_1, \dots, m_n))$ is $(T_s(\mathbf{P}), \sigma_f)$ -normal, if such an i exists, and is undefined otherwise.

Now, let $f: N_\perp^n \rightarrow N_\perp$ be a Δ_N -computable function, let $f = \mu \Phi_F(F)$, let $a = \#_{20}(\mathbf{P})$, and let $b = \#_{20}(F)$. Then it is clear that

$f_\pi(x_1, \dots, x_n) = \text{EVAL}(\text{RRPT}^*(a, b, x_1, \dots, x_n, \text{END}(a, b, x_1, \dots, x_n)))$ for all $(x_1, \dots, x_n) \in N^n$. EVAL , RRPT^* , and END are partially computable, so f_π is partially computable. ■

Exercises

1. Show that for all $n > 0$ and all $(x_1, \dots, x_n) \in N_\perp^n$,

$$\mathcal{D}_{\Delta_N}(\mathbf{P}_{\text{Def}^n})(\mathbf{Def}^n)(x_1, \dots, x_n) = \begin{cases} \text{TRUE} & \text{if } (x_1, \dots, x_n) \in N^n \\ \perp_{\text{Bool}} & \text{otherwise.} \end{cases}$$

2. For each of the following functions f from Chapter 3, give a \mathbf{W}_N -program \mathbf{P} with $F \in FV(\mathbf{P})$ such that $\mathcal{D}_{\Delta_N}(\mathbf{P})(F) = f_\perp$.

- (a) $f(x, y) = x \cdot y$.
- (b) $f(x) = x!$.
- (c) $f(x, y) = x^y$.
- (d) $f(x) = p(x)$.
- (e) $f(x, y) = x \div y$.
- (f) $f(x, y) = |x - y|$.
- (g) $f(x) = \alpha(x)$.

3. Let $P(x), Q(x)$ be primitive recursive predicates, and let \mathbf{P}, \mathbf{Q} be \mathbf{W}_N -programs such that $\mathcal{D}_{\Delta_N}(\mathbf{P})(\mathbf{F}_P) = P_\perp$ and $\mathcal{D}_{\Delta_N}(\mathbf{P})(\mathbf{F}_Q) = Q_\perp$. For each of the following predicates $R(x)$, give a \mathbf{W}_N -program \mathbf{R} with $\mathbf{F}_R \in FV(\mathbf{R})$ such that $\mathcal{D}_{\Delta_N}(\mathbf{R})(\mathbf{F}_R) = R_\perp$. [Also see Exercise 3.12 in Chapter 17.]
- $R(x) \Leftrightarrow \sim P(x).$
 - $R(x) \Leftrightarrow P(x) \ \& \ Q(x).$
 - $R(x) \Leftrightarrow P(x) \vee Q(x).$
 - $R(x) \Leftrightarrow (\exists z)_{\leq x} P(z).$
 - $R(x) \Leftrightarrow (\forall z)_{\leq x} P(z).$
4. Let $P(x, y)$ be a primitive recursive predicate, and let \mathbf{P} be a \mathbf{W}_N -program such that $\mathcal{D}_{\Delta_N}(\mathbf{P})(\mathbf{F}_P) = P_\perp$. Give a \mathbf{W}_N -program \mathbf{R} with $\mathbf{Min}_R \in FV(\mathbf{R})$ such that $\mathcal{D}_{\Delta_N}(\mathbf{R})(\mathbf{Min}_R)$ is the strict extension of $\min_{z \leq x} P(z, y)$.
5. For each of the following predicates P from Chapter 3, give a \mathbf{W}_N -program \mathbf{P} with $\mathbf{F}_P \in FV(\mathbf{P})$ such that $\mathcal{D}_{\Delta_N}(\mathbf{P})(\mathbf{F}_P) = P_\perp$.
- $P(x, y) \Leftrightarrow x = y.$
 - $P(x, y) \Leftrightarrow x \leq y.$
 - $P(x, y) \Leftrightarrow x < y.$
 - $P(x, y) \Leftrightarrow x \mid y.$
 - $P(x) \Leftrightarrow \text{Prime}(x).$
6. For each of the following functions f from Chapter 3, give a \mathbf{W}_N -program \mathbf{P} with $\mathbf{F} \in FV(\mathbf{P})$ such that $\mathcal{D}_{\Delta_N}(\mathbf{P})(\mathbf{F}) = f_\perp$.
- $f(x, y) = \lfloor x/y \rfloor.$
 - $f(x, y) = R(x, y).$
 - $f(x) = p_x.$
 - $f(x, y) = \langle x, y \rangle.$
 - $f(x) = l(x).$
 - $f(x) = r(x).$
 - $f(x_1, \dots, x_n) = [x_1, \dots, x_n].$
 - $f(x, y) = (x)_y.$
 - $f(x) = \text{Lt}(x).$
7. Show that RRPT* is primitive recursive.
8. Let $\# : \{\mathcal{S} \text{ programs}\} \rightarrow N$ be the coding function for \mathcal{S} programs given in Chapter 4. Give a $\mathbf{W}_{\mathcal{S}}$ -program \mathbf{P} with $\mathbf{C} \in FV(\mathbf{P})$ such that $\mathcal{D}_{\Delta_{\mathcal{S}}}(\mathbf{P})(\mathbf{C}) = \#_\perp$. [See Section 5 in Chapter 17 for the definitions of $\mathbf{W}_{\mathcal{S}}$ and $\Delta_{\mathcal{S}}$.]

- 9.* (a)** Give a \mathbf{W}_N -program \mathbf{SMN} with $S \in FV(\mathbf{SMN})$ such that $\mathcal{D}_{\Delta_N}(\mathbf{SMN})(S)$ is the strict extension of S_1^1 . [See the parameter theorem in Chapter 4 for the definition of S_1^1 .]
- (b)** Let $s_1^1: N \times \{\mathcal{S} \text{ programs}\} \rightarrow \{\mathcal{S} \text{ programs}\}$ be defined: For all \mathcal{S} programs \mathcal{P} and all $u \in N$,

$$\Phi^{(2)}(x, u, \#(\mathcal{P})) = \Phi(x, \#(s_1^1(u, \mathcal{P}))).$$

Give a $\mathbf{W}_{\mathcal{S}}$ -program \mathbf{SMN} with $S \in FV(\mathbf{SMN})$ such that $\mathcal{D}_{\Delta_{\mathcal{S}}}(\mathbf{SMN})(S)$ is the strict extension of s_1^1 .

3. Operational Semantics for Infinitary Data Structure Systems

We turn now to the operational semantics for infinitary data structure systems. It differs in two respects from the operational semantics we gave for simple data structure systems. First, for a term such as $\mathbf{is_f(f(t))}$, we cannot be sure that $\overline{\mu\Phi_P(is_f(f(t)))} = tt$ when $\mathcal{I}(f)$ is strict because $\overline{\mu\Phi_P(t)}$ might be $\perp_{\tau(t)}$. Therefore, we defined $T_s(P)$ so that we rewrite $\mathbf{is_f(f(t))}$ to tt only when t is free of variables, which guarantees that $\overline{\mu\Phi_P(t)} \neq \perp_{\tau(t)}$. Moreover, if $\overline{\mu\Phi_P(t)} \neq \perp_{\tau(t)}$, then $\overline{\Phi_P^{i_0}(\Omega)}(t) \neq \perp_{\tau(t)}$ for some smallest i_0 , and Lemmas 3 and 4 guarantee that t will eventually be rewritten to some term which is free of variables. In an infinitary data structure system, however, this problem does not arise because we always have $\overline{\mu\Phi_P(f(t))} \neq \perp_{\rho(f)}$. Therefore, we can replace $T_s(P)$ with a simpler, indeed finite, term rewriting system.

Definition. Let P be a \mathbf{W} -program. The *infinitary \mathbf{W} -term rewriting system associated with P* , denoted $T_i(P)$, consists of

$$F(X_1, \dots, X_n) \rightarrow u \quad \text{for each equation } F(X_1, \dots, X_n) = u \text{ in } P,$$

together with

- for each $\tau \in TV(W)$,

$$\begin{aligned} \mathbf{if}_{\tau}(tt, X, Y) &\rightarrow X \\ \mathbf{if}_{\tau}(ff, X, Y) &\rightarrow Y \end{aligned}$$

- for each constant symbol $c \in W_c^-$, with $\tau(c) = \tau$,

$$\begin{aligned} & \text{is_c}(c) \rightarrow tt \\ & \text{is_c}(d) \rightarrow ff \quad \text{for each } d \in W_c^- - \{c\} \text{ with } \tau(d) = \tau \\ & \text{is_c}(f(X_1, \dots, X_n)) \rightarrow ff \quad \text{for each } f \in W_c \text{ with} \\ & \qquad \qquad \qquad \tau(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau \end{aligned}$$

- for each proper function symbol $f \in W_c$, with $\text{ar}(f) = n$ and $\rho(f) = \tau$,

$$\begin{aligned} & \text{is_f}(f(X_1, \dots, X_n)) \rightarrow tt \\ & \text{is_f}(c) \rightarrow ff \quad \text{for each } c \in W_c \text{ with } \tau(c) = \tau \\ & \text{is_f}(g(X_1, \dots, X_m)) \rightarrow ff \quad \text{for each } g \in W_c - \{f\} \text{ with} \\ & \qquad \qquad \qquad \text{ar}(g) = m \text{ and } \rho(g) = \tau \\ & f_i^{-1}(f(X_1, \dots, X_n)) \rightarrow X_i \quad \text{for } 1 \leq i \leq n. \end{aligned}$$

Again, the choice of particular individual variables is unimportant, as long as they are of the appropriate type and in each rewrite rule the variables are distinct.

It is easy to see that all term rewriting systems $T_i(P)$ are deterministic and that $(T_i(P), \sigma_f)$ -computations are a reasonable sort of computation. In this section we will write r_P and rr_P for $r_{T_i(P)}$ and $rr_{T_i(P)}$, respectively.

The other difference we need to address is that there are infinite chains in infinitary data structure systems. In particular, there are terms t such that $\overline{\Omega(rr_P^i(t))} = \overline{\Phi_P^i(\Omega)}(t) \sqsubset_{\tau(t)} \overline{\mu\Phi_P}(t)$ for all $i \in N$, so that we get an infinite computation

$$t \xrightarrow[T_i(P), \sigma_f]{} rr_P(t) \xrightarrow[T_i(P), \sigma_f]{} rr_P^2(t) \xrightarrow[T_i(P), \sigma_f]{} \dots$$

which never reaches the desired value $\overline{\mu\Phi_P}(t)$. Therefore, we cannot expect to base the operational semantics on the final terms of finite computations. Instead, we take the point of view that an infinite computation produces ever better approximations to the actual value and that the entire computation gives the meaning of the function being computed.

Definition. Let $\Delta = \text{rep}(\Sigma)$ be an infinitary data structure system for W . The *operational meaning function* for Δ , denoted \mathcal{O}_Δ , is defined as follows. For all W -programs P_0 , all $F \in FV(P_0)$, and all $(d_1, \dots, d_n) \in D_{r(\delta(F))}$,

$$\mathcal{O}_\Delta(P_0)(F)(d_1, \dots, d_n) = \sqcup \{ \overline{\Omega(rr_P^i(F(t_1, \dots, t_n)))} \mid i \in N \},$$

where

- $d_i = \overline{\mu\Phi_{P_i}^{\Sigma}}(t_i)$, $1 \leq i \leq n$,
- P_0, P_1, \dots, P_n are consistent, and
- $P = \bigcup_{i=0}^n P_i$.

The proof that \mathcal{O}_{Δ} is correct with respect to \mathcal{D}_{Δ} for all infinitary data structure systems is very much like the proof of Theorem 1.1.

Lemma 1. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be an infinitary \mathbf{W} -structure, and let P be a \mathbf{W} -program. Then for all $t \in TM_{\mathbf{W}}(FV(P))$ and for all $i \in N$, $\overline{\Phi_P^{i+1}}(\Omega)(t) = \overline{\Phi_P^i}(\Omega)(rr_P(t))$.

Proof. The proof is almost identical to the proof of Lemma 2 in Section 1. The only differences occur in the cases where t is of the form $\text{is_f}(u)$ or $f_i^{-1}(u)$. In particular, if $\text{is_f}(rr_P(u))$ or $f_i^{-1}(rr_P(u))$ match a rewrite rule in $T_i(P)$, then we do not necessarily have $rr_P(u) \in TM_{\mathbf{W}_c}$, so we cannot appeal to Lemma 1 in Section 1 to show that $\overline{\Phi_P^i}(\Omega)(rr_P(u)) \neq \perp_{\tau(u)}$. However, $rr_P(u)$ must be of the form g or $g(u_1, \dots, u_m)$ for some $g \in \mathbf{W}_c$, so that $\overline{\Phi_P^i}(\Omega)(rr_P(u)) \neq \perp_{\tau(u)}$ is certainly true in an infinitary \mathbf{W} -structure, and the argument goes through unchanged but for the reference to Lemma 1 in Section 1. ■

Repeating the proof of Lemma 3 in Section 1 gives us

Lemma 2. Let $\Sigma = (\mathcal{T}, \mathcal{I})$ be an infinitary \mathbf{W} -structure, and let P be a \mathbf{W} -program. Then for all $t \in TM_{\mathbf{W}}(FV(P))$ and for all $i \in N$, $\overline{\Phi_P^i}(\Omega)(t) = \overline{\Omega}(rr_P^i(t))$.

Now the proof of Theorem 3.1 is even simpler than the proof of Theorem 1.1.

Theorem 3.1. Let $\Delta = \text{rep}(\Sigma)$ be an infinitary data structure system for \mathbf{W} . Then $\mathcal{O}_{\Delta} = \mathcal{D}_{\Delta}$.

Proof. Let P_0 be a \mathbf{W} -program, let $F \in FV(P_0)$, and let $(d_1, \dots, d_n) \in D_{r(\delta(F))}$. For $1 \leq i \leq n$ let $d_i = \overline{\mu\Phi_{P_i}^{\Sigma}}(t_i)$ for some \mathbf{W} -program P_i and some $t_i \in TM_{\mathbf{W}}(FV(P_i))$ such that P_0, P_1, \dots, P_n are consistent, and let $P = \bigcup_{i=0}^n P_i$. Then

$$\begin{aligned} \mathcal{D}_{\Delta}(P_0)(F)(d_1, \dots, d_n) &= \sqcup \{ \overline{\Phi_P^i}(\Omega)(F(t_1, \dots, t_n)) \mid i \in N \} \quad \text{as in the proof of Theorem 1.1} \\ &= \sqcup \{ \overline{\Omega}(rr_P^i(F(t_1, \dots, t_n))) \mid i \in N \} \quad \text{by Lemma 2} \\ &= \mathcal{O}_{\Delta}(P_0)(F)(d_1, \dots, d_n). \end{aligned}$$

■

As we did for simple data structure systems, we can now define computable functions in infinitary data structure systems.

Definition. Let Δ be an infinitary data structure system for \mathbf{W} . A function f is Δ -computable if there is a \mathbf{W} -program \mathbf{P} and $\mathbf{F} \in FV(\mathbf{P})$ such that $f = \mathcal{D}_\Delta(\mathbf{P})(\mathbf{F})$.

We can also use Lemma 2 to justify the name *computable real numbers* used in the previous chapter.

Definition. Let Σ be an infinitary \mathbf{W} -structure, and let $\tau \in TV(\mathbf{W})$. An element $d \in D_\tau$ is computable if there is some \mathbf{W} -program \mathbf{P} and some term $\mathbf{t} \in TM_{\mathbf{W}}(FV(\mathbf{P}))$ such that $d = \sqcup \{\overline{\Omega}(rr_P^i(\mathbf{t})) \mid i \in N\}$.

Now we can easily prove

Theorem 3.2. Let Σ be an infinitary \mathbf{W} -structure, and let $\tau \in TV(\mathbf{W})$. Then an element $d \in D_\tau$ is representable if and only if it is computable.

Proof. Let $d \in D_\tau$. If d is representable then there is a \mathbf{W} -program \mathbf{P} and a term $\mathbf{t} \in TM_{\mathbf{W}}(FV(\mathbf{P}))$ such that

$$\begin{aligned} d &= \overline{\mu \Phi_{\mathbf{P}}}(\mathbf{t}) \\ &= \overline{\sqcup \{\Phi_{\mathbf{P}}^i(\Omega) \mid i \in N\}}(\mathbf{t}) \\ &= \sqcup \{\overline{\Phi_{\mathbf{P}}^i(\Omega)}(\mathbf{t}) \mid i \in N\} \quad \text{by Theorem 17.2.3} \\ &= \sqcup \{\overline{\Omega}(rr_P^i(\mathbf{t})) \mid i \in N\} \quad \text{by Lemma 2,} \end{aligned}$$

so d is computable. Similarly, if d is computable then there is a \mathbf{W} -program \mathbf{P} and a term $\mathbf{t} \in TM_{\mathbf{W}}(FV(\mathbf{P}))$ such that

$$d = \sqcup \{\overline{\Omega}(rr_P^i(\mathbf{t})) \mid i \in N\} = \overline{\mu \Phi_{\mathbf{P}}}(\mathbf{t}),$$

and so d is representable. ■

Now that we have available all of the strict extensions of the partially computable functions, we conclude with two promised examples of computing with infinite objects. We will be working in infinitary data structures, but it is not hard to verify that if $f(x_1, \dots, x_n)$ is partially computable and \mathbf{P} is a \mathbf{W}_N -program obtained in the proof of Theorem 2.1 such that $\mathcal{D}_{\Delta_N}(\mathbf{P})(\mathbf{F}) = f_\perp$, then

$$\mathcal{D}_{\Delta_N^*}(\mathbf{P})(\mathbf{F})(x_1, \dots, x_n) = f_\perp(x_1, \dots, x_n) \tag{3.1}$$

$$\begin{aligned}
 \text{Primes}(X) &= \text{Sieve}(\text{Seq}(X)) \\
 \text{Seq}(X) &= \text{cons}(X, \text{Seq}(s(X))) \\
 \text{Sieve}(L) &= \text{cons}(\text{cons}_1^{-1}(L), \text{Sieve}(\text{Elim}(\text{cons}_1^{-1}(L), \text{cons}_2^{-1}(L)))) \\
 \text{Elim}(X, L) &= \text{if}_{NL}(X \mid \text{cons}_1^{-1}(L), \\
 &\quad \text{Elim}(X, \text{cons}_2^{-1}(L)), \\
 &\quad \text{cons}(\text{cons}_1^{-1}(L), \text{Elim}(X, \text{cons}_2^{-1}(L))))
 \end{aligned}$$

Figure 3.1. The main part of program PR.

for all $(x_1, \dots, x_n) \in N^n$. We will now freely write $f(x_1, \dots, x_n)$ as a macro in W_{NL} -programs and W_R -programs when $f(x_1, \dots, x_n)$ is partially computable. If $P(x_1, \dots, x_n)$ is a computable predicate, then when we write $P(x_1, \dots, x_n)$ in a program its range should be understood as $\{\perp_{Bool}, \text{tt}, \text{ff}\}$.

The first example is a W_{NL} -program for generating the list $\langle p_1, p_2, \dots \rangle$ of all prime numbers. It is based on the method known as Eratosthenes' sieve, where we start with the list $\langle 2, 3, 4, \dots \rangle$, eliminate all numbers divisible by 2, then eliminate all numbers divisible by 3, all numbers divisible by 5, etc. Let PR be the W_{NL} -program with the equations in Fig. 3.1 along with the definition of the predicate $x \mid y$, i.e., “ x divides y .” Then in Σ_{NL}^∞ we have

$$\begin{aligned}
 \overline{\mu\Phi_{PR}}(\text{Primes}(2)) &= \mu\Phi_{PR}(\text{Sieve})(\mu\Phi_{PR}(\text{Seq})(2)) \\
 &= \mu\Phi_{PR}(\text{Sieve})(\langle 2, 3, 4, \dots \rangle) \\
 &= \langle 2, \mu\Phi_{PR}(\text{Sieve})(\mu\Phi_{PR}(\text{Elim})(2, \langle 3, 4, 5, \dots \rangle)) \rangle \\
 &= \langle 2, \mu\Phi_{PR}(\text{Sieve})(\langle 3, 5, 7, \dots \rangle) \rangle \\
 &= \langle 2, 3, \mu\Phi_{PR}(\text{Sieve})(\mu\Phi_{PR}(\text{Elim})(3, \langle 5, 7, 9, \dots \rangle)) \rangle \\
 &= \langle 2, 3, \mu\Phi_{PR}(\text{Sieve})(\langle 5, 7, 11, \dots \rangle) \rangle \\
 &\quad \vdots \qquad \qquad \qquad \vdots \\
 &= \langle 2, 3, 5, \dots \rangle
 \end{aligned}$$

where the notation $\langle 2, \mu\Phi_{PR}(\text{Sieve})(\langle 3, 5, 7, \dots \rangle) \rangle$ means the list with 2 followed by the elements of the list $\mu\Phi_{PR}(\text{Sieve})(\langle 3, 5, 7, \dots \rangle)$. Therefore, the list of primes is representable. Moreover, Theorem 3.2 shows that it can be generated by a $(T_i(PR), \sigma_f)$ -computation.

Finally, we show that there are computable irrational numbers. It is a mathematical fact that the well-known irrational number $e = 2.7182\dots$ can be expressed as $2 + \frac{1}{2!} + \frac{1}{3!} + \dots$, and we can use this fact to show that $e/10 = .27182\dots$ is computable.⁵ The idea is to consider $\frac{n_1}{2!} + \frac{n_2}{3!} + \dots$ as a sort of base notation system analogous to the decimal system, where .354, for example, represents $\frac{3}{10} + \frac{5}{10^2} + \frac{4}{10^3}$. All we have to do, then, is to change $2 + \frac{1}{2!} + \frac{1}{3!} + \dots$ (divided by 10) into its decimal representation. The procedure consists of taking the integer part as the first decimal digit, multiplying the fractional part by 10, normalizing the result (i.e., reducing the numerators by carrying), and then repeating these steps with the normalized result. The correct method for carrying is given by the equations

$$\begin{aligned}\frac{m}{n+1!} &= \frac{(n+1)[m/(n+1)] + R(m, n+1)}{n+1!} \\ &= \frac{\lfloor m/(n+1) \rfloor}{n!} + \frac{R(m, n+1)}{n+1!}.\end{aligned}$$

For example, starting with $2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!}$ we get 2 as the first decimal digit, and then we get

$$\begin{aligned}10 \cdot \left(\frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!}\right) &= \frac{10}{2!} + \frac{10}{3!} + \frac{10}{4!} \\ &= \frac{10}{2!} + \frac{12}{3!} + \frac{2}{4!} \\ &= \frac{14}{2!} + \frac{0}{3!} + \frac{2}{4!} = 7 + \frac{0}{2!} + \frac{0}{3!} + \frac{2}{4!},\end{aligned}$$

so 7 is the second decimal digit. Next we get

$$\begin{aligned}10 \cdot \left(\frac{0}{2!} + \frac{0}{3!} + \frac{2}{4!}\right) &= \frac{0}{2!} + \frac{0}{3!} + \frac{20}{4!} \\ &= \frac{0}{2!} + \frac{5}{3!} + \frac{0}{4!} = 0 + \frac{1}{2!} + \frac{2}{3!} + \frac{0}{4!},\end{aligned}$$

so 0 is the third decimal digit. Notice, however, that if we start with $2 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!}$, we get .271 instead of .270. That is, we get more precision by starting with more terms. However, if we want to get the decimal expansion of the *infinite* sum $2 + \sum_{n=2}^{\infty} \frac{1}{n!}$, then at any given iteration we certainly cannot perform the entire multiplication by 10 before beginning the carry step. Fortunately, we need to perform only enough of it so that multiplying and normalizing any additional terms would not change the decimal digit produced by the current iteration.

⁵ This example is due to D. A. Turner, who credits E. W. Dijkstra with the idea.

$$\begin{aligned}
 K(X) &= \text{cons}(X, K(X)) \\
 \text{Convert}(L) &= \text{dcons}(\text{Digit}(\text{First}(L)), \\
 &\quad \text{Convert}(\text{Norm}(2, \text{cons}(0, \text{Mult10}(\text{Rest}(L)))))) \\
 \text{Norm}(C, L) &= \text{if}_{NL}(\text{Second}(L) + 9 < C, \\
 &\quad \text{cons}(\text{First}(L), \text{Norm}(s(C), \text{Rest}(L))), \\
 &\quad \text{Carry}(C, \text{cons}(\text{First}(L), \text{Norm}(s(C), \text{Rest}(L))))) \\
 \text{Carry}(C, L) &= \text{cons}(\text{First}(L) + \lfloor \text{Second}(L)/C \rfloor \\
 &\quad \text{cons}(R(\text{Second}(L), C), \text{Rest2}(L))) \\
 \text{Mult10}(L) &= \text{cons}(10 \cdot \text{First}(L), \text{Mult10}(\text{Rest}(L))) \\
 \text{First}(L) &= \text{cons}_1^{-1}(L) \\
 \text{Second}(L) &= \text{cons}_1^{-1}(\text{cons}_2^{-1}(L)) \\
 \text{Rest}(L) &= \text{cons}_2^{-1}(L) \\
 \text{Rest 2}(L) &= \text{cons}_2^{-1}(\text{cons}_2^{-1}(L))
 \end{aligned}$$

Figure 3.2. The main part of program E.

Now, the carry procedure leaves fractions of the form $R(m, n)/n!$, where $R(m, n) < n$, so when we multiply by 10 we get $10 \cdot R(m, n)/n!$, and then we carry $\lfloor 10 \cdot R(m, n)/n \rfloor < 10n/n = 10$. So the maximum possible carry is always 9, and if we have a term $m/n!$ such that $m + 9 < n$, there will never be a carry out of $m/n!$, regardless of what is carried into $m/n!$. Therefore, when we reach such a term, we can be sure that we have enough information to produce a correct decimal digit. Let **E** be the program with the equations in Fig. 3.2 together with the appropriate definitions of addition, multiplication, integer division, and remainder. Also, **Digit**, with $\tau(\text{Digit}) = N \rightarrow D$, must be defined so that $\mu\Phi_E(\text{Digit})$ turns natural numbers $0, \dots, 9$ into decimal digits $0, \dots, 9$. Now, $\overline{\mu\Phi_E(K(1))}$ is the infinite list $\langle 1, 1, \dots \rangle \in D_{NL^\omega}$, which we use to represent $\sum_{n=2}^{\infty} 1/n!$. We leave it as an exercise to check that $\mu\Phi_E(\text{Convert}(\text{cons}(2, K(1)))) = e/10 \in D_{DL^\omega}$. So $e/10$ is representable, and Theorem 3.2 justifies calling $e/10$ a computable real number.

Exercises

- Let $P = \text{ADD} \cup \{B(X) = B(X)\}$, where $\tau(B) = N \rightarrow N$. Give $T_i(P)$, and give the $(T_i(P), \sigma_f)$ -computation for each of the following.
 - $+ (3, 2)$.
 - $+ (3, s(s(B(0))))$.
 - $+ (s(s(s(B(0)))), 2)$.

2. Without using Theorem 3.1, give the value of each of the following.
 - (a) $\mathcal{O}_{\Delta_N^{\infty}}(\text{ADD})(+)(3, 2)$.
 - (b) $\mathcal{O}_{\Delta_N^{\infty}}(\text{ADD})(+)(3, 2_{\perp})$.
 - (c) $\mathcal{O}_{\Delta_N^{\infty}}(\text{ADD})(+)(3_{\perp}, 2)$.
3. Let $P = \text{LIST} \cup \{B(X) = B(X), BL(X) = BL(X)\}$, where $\tau(B) = N \rightarrow N$ and $\tau(BL) = NL \rightarrow NL$, let t be $\text{cons}(B(0), \text{cons}(1, \text{nil}))$, and let u be $\text{cons}(0, \text{cons}(1, BL(\text{nil})))$. [LIST is defined in Section 5 of Chapter 17.] Give $T_i(P)$, and give the $(T_i(P), \sigma_f)$ -computation for each of the following.
 - (a) $\text{Length}(t)$.
 - (b) $\text{Length}(u)$.
 - (c) $\text{Nth}(0, t)$.
 - (d) $\text{Nth}(1, u)$.
 - (e) $\text{Cat}(t, t)$.
 - (f) $\text{Cat}(t, u)$.
 - (g) $\text{Cat}(u, t)$.
 - (h) $\text{Reverse}(t)$.
 - (i) $\text{Reverse}(u)$.
 - (j) $\text{Length}(\text{Cat}(\text{Rev}(t), t))$.
4. Let $l_1 = \langle \perp_{N^{\infty}}, 1 \rangle, l_2 = \langle 0, 1 \rangle_{\perp}$. Without using Theorem 3.1, give the value of each of the following.
 - (a) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Length})(l_1)$.
 - (b) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Length})(l_2)$.
 - (c) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Nth})(0, l_1)$.
 - (d) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Nth})(1, l_2)$.
 - (e) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Cat})(l_1, l_1)$.
 - (f) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Cat})(l_1, l_2)$.
 - (g) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Cat})(l_2, l_1)$.
 - (h) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Rev})(l_1)$.
 - (i) $\mathcal{O}_{\Delta_{NL}^{\infty}}(\text{LIST})(\text{Rev})(l_2)$.
5. Give a W_N -program with $F \in FV(P)$ such that $\mathcal{D}_{\Delta_N^{\infty}}(P)(F)(0) = \perp_N$ and $\mathcal{D}_{\Delta_N^{\infty}}(P)(F)(0) = 0$. Verify that $\mathcal{O}_{\Delta_N^{\infty}}(P)(F)(0) = \perp_N$ and $\mathcal{O}_{\Delta_N^{\infty}}(P)(F)(0) = 0$.
6. Verify the sentence containing (3.1).
7. Give the $(T_i(\text{PR} \cup \text{LIST}), \sigma_f)$ -computation for $\text{Nth}(1, \text{Primes}(2))$.
8. Suppose we change the defining equation for Norm in E to

$$\text{Norm}(C, L) = \text{Carry}(C, \text{cons}(\text{First}(L), \text{Norm}(\text{s}(C), \text{Rest}(L)))).$$

Now what is $\overline{\mu\Phi_E}(\text{Convert}(\text{cons}(2, K(1))))$?

9. Show that if A is a nonempty r.e. set, then there is a computable list $l = \langle i_0, i_1, \dots \rangle$ in D_{NL^\times} such that $A = \{i \in N \mid i \text{ occurs in } l\}$.
- 10.* Show that if $l = \langle i_0, i_1, \dots \rangle$ is a computable list of numbers in D_{NL^\times} , then $\{i \in N \mid i \text{ occurs in } l\}$ is r.e. [*Hint:* Adapt the proof of Theorem 2.2.]

Suggestions for Further Reading

C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

A very readable treatise on resolution-based algorithms for satisfiability in quantification theory.

Martin Davis, *Computability and Unsolvability*. Dover, New York, 1983.

Originally published in 1958. The 1983 reprint includes an appendix on unsolvable problems in number theory.

Martin Davis (editor), *The Undecidable*. Raven, New York, 1965.

A collection of basic papers in computability theory. Included are the original papers in which Church announced his “thesis,” in which Turing defined his machines and produced a universal computer, in which Post stated his “problem,” and in which Turing introduced “oracles.”

Herbert P. Enderton, *A Mathematical Introduction to Logic*. Academic Press, New York, 1972.

An introductory textbook on mathematical logic for mathematically mature readers.

Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

This treatise includes a comprehensive list of NP-complete problems.

Carl A. Gunter, *Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1992.

A treatment of denotational semantics for the sophisticated reader.

Paul R. Halmos, *Naive Set Theory*. Van Nostrand, Princeton, New Jersey, 1964.

A short, classic introduction to set theory.

Michael Harrison, *Introduction to Formal Language Theory*. Addison-Wesley, Reading, Massachusetts, 1978.

A comprehensive, up-to-date, readable treatise on formal languages.

Karel Hrbacek and Thomas Jech, *Introduction to Set Theory*, second edition. Marcel Dekker, New York, 1984.

Another introduction to set theory, somewhat more detailed than the book by Halmos.

Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

Another introduction to theoretical computer science.

Jacques Loeckx and Kurt Sieber, *The Foundations of Program Verification*, second edition. John Wiley and Sons, New York, 1987.

A well-written treatment of programming language semantics with an emphasis on program verification.

Donald W. Loveland, *Automated Theorem Proving: A Logical Basis*. North-Holland Publ., Amsterdam, 1978.

A well-organized account of resolution theory.

Michael Machtey and Paul Young, *An Introduction to the General Theory of Algorithms*. North-Holland Publ., Amsterdam, 1978.

A well-written account of computability and complexity theory.

Hartley Rogers, *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

The classic comprehensive treatise on computability and noncomputability.

David Schmidt, *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

A good general introduction to the denotational semantics of programming languages.

Joseph R. Shoenfield, *Degrees of Unsolvability*. North-Holland Publ., Amsterdam, 1971.

A short and clearly written monograph on the subject going well beyond the material covered in this book.

Robert I. Soare, *Recursively Enumerable Sets and Degrees: The Study of Computable Functions and Computably Generated Sets*. Springer-Verlag, Berlin and New York, 1987.

A modern treatment of advanced recursive function theory.

Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Massachusetts, 1977.

An early treatment of denotational semantics.

Notation Index

\in	1	$(\exists t), (\forall t)$	7
\emptyset	1	■	9
\subseteq, \subset	1	\mathcal{S}	17
\cup, \cap	2	$\psi_{\varphi}^{(m)}(r_1, \dots, r_m)$	29
$R - S$	2	$s(x), n(x), u_i^n(x_1, \dots, x_n)$	42
\bar{S}	2	$x \doteq y$	46
$\{a_1, \dots, a_n\}$	2	$\alpha(x)$	47
(a_1, \dots, a_n)	2	$\min_{t \leq y} P(t, x_1, \dots, x_n)$	55
\times	2	$\lfloor x/y \rfloor, R(x, y)$	56
S^n	3	p_n	56
$f(a) \downarrow, f(a) \uparrow$	3	$\min_y P(x_1, \dots, x_n, y)$	57
$ u $	4	$\langle x, y \rangle$	59
A^*	4	$l(z), r(z)$	60
\widehat{uv}	4	$[a_1, \dots, a_n]$	60
$u^{[n]}$	4	$(x)_i$	61
$\sim, \&, \vee$	5	$\text{Lt}(x)$	61
$\{a \in S \mid P(a)\}$	6	$\#(\mathcal{P})$	65, 201
\Leftrightarrow	6	$\text{HALT}(X, Y)$	68
$(\exists t)_{\leq y}, (\forall t)_{\leq y}$	7	$\Phi^{(n)}(x_1, \dots, x_n, y)$	70

\mathcal{U}_n	70	$L_1 \cdot L_2$	253
$\Phi_y^{(n)}(x_1, \dots, x_n)$	73	L^*	253
$\Phi_y(x)$	73	$\mathbf{0}, \emptyset, \cup, \cdot, *$	256
$\text{STP}^{(n)}(x_1, \dots, x_n, y, t)$	74	$\langle \gamma \rangle$	257
W_n	81	\equiv_L	263
K	82	$X \rightarrow h$	269
$S_m^n(u_1, \dots, u_n, y)$	85, 205	$u \Rightarrow v$ Γ	270
\leq_m	91	$\ker(\Gamma)$	271
\equiv_m	93	\mathcal{T}^v	274
R_Γ	95	$\langle \mathcal{T} \rangle$	274
w^-	119	$\text{Er}_p(L), \text{Er}_p(\Gamma)$	294
\mathcal{S}_n	121	$\text{PAR}_n(A)$	301
\mathcal{T}	129	Γ_s	303
\vdash	159	$\vdash_{\mathcal{T}}$	311
$g \rightarrow h$	169	$L_A(\mathcal{M})$	331
$u \Rightarrow v, u \stackrel{*}{\Rightarrow} v$	170	$\neg, \wedge, \vee, \supset, \leftrightarrow$	347
$\Sigma(\mathcal{M})$	171	γ^v	348
$\Omega(\mathcal{M})$	176	$\gamma_1, \dots, \gamma_n \models \gamma$	352, 382
$\Theta(\mathcal{M})$	177	\square	360
$L(\Gamma)$	186, 270	$\alpha_\lambda^+, \alpha_\lambda^-, \alpha_\lambda^0$	361
$\psi_{\mathcal{R}, G}^{(m)}(r_1, \dots, r_m)$	198	$\text{POS}_\lambda(\alpha), \text{NEG}_\lambda(\alpha)$	361
$\text{STP}_G^{(n)}(x_1, \dots, x_n, y, t)$	201	$\text{RES}_\lambda(\alpha)$	367
$\{u\}(i)$	203	$\delta(t)$	375
$u \prec G$	204	\forall, \exists	375
\leq_t, \leq_1	207	$\lambda = \lambda(b_1, \dots, b_n)$	378
\equiv_Q	208	$\alpha'[d_1, \dots, d_n]$	378
co-W	208	$\Gamma \vDash \gamma$	382
$A \oplus B$	210	γ_s	384
W_n^G	213	\mathbf{T}_Γ	407
G'	213	$\vdash_T \gamma$	407
$G^{(n)}$	214	$w^\#$	411
$\Sigma_n, \Pi_n, \Delta_n$	215	$M_i(x)$	420
TOT	224	a.e.	421
f^{-1}	231	∞	421
$L(\mathcal{M})$	239, 243, 311	$O(g(n))$	439

P	444	F ($\mathbf{X}_1, \dots, \mathbf{X}_n$) = \mathbf{t}	507
SAT	446	<i>PF(E)</i> , <i>AF(E)</i>	508
NP	446	<i>rhs(E)</i>	508
\leq_p	448	Bool , tt , ff	508
$\nabla\{x_e \mid 1 \leq e \leq l\}$	453	if $_{\tau}$, is_f , \mathbf{f}_i^{-1}	509
\perp_N	471	($\mathbf{W}_c \cup B(\mathbf{W}_c)$, $\tau_c \cup \tau_{B(\mathbf{W}_c)}$)	509
\sqsubseteq	472	(\mathbf{W}_N , τ_N), (\mathbf{W}_{NL} , τ_{NL})	510
$\mathcal{P}(D)$	472	($D_{\mathcal{T}(\tau)}$, $\sqsubseteq_{\mathcal{T}(\tau)}$), $\perp_{\mathcal{T}(\tau)}$	511
$\sqsubseteq_{D_1 \times \dots \times D_n}$	472	(Bool , $\sqsubseteq_{\text{Bool}}$), \perp_{Bool}	512
$(D \rightarrow E, \sqsubseteq_{D \rightarrow E})$	473	($D_{\tau}, \sqsubseteq_{\tau}$), \perp_{τ}	512
$(\text{ch}(\mathcal{D}), \sqsubseteq_{\text{ch}(\mathcal{D})})$	474	($D_{\tau_1 \times \dots \times \tau_n}$, $\sqsubseteq_{\tau_1 \times \dots \times \tau_n}$)	512
$\sqcup E$, $\sqcap E$	476	$D_{\delta(\mathbf{F})}$	512
$(D_{\perp}, \sqsubseteq_{D_{\perp}})$	477	$\text{ran } f$	512
$\cup \mathcal{E}$	478	tt , ff	512
$d \downarrow i$	478	$\Sigma_N = (\mathcal{T}_N, \mathcal{I}_N)$	514
$\mathcal{F}(d)$	479	$\alpha(\mathbf{F})$, $\alpha(\mathbf{X})$	514
$f(g)(d)$	481	$\mathcal{A}_{\mathcal{T}}(V)$	514
$(\text{id}(D), \sqsubseteq_{\text{id}(D)})$	482	$\mathcal{C}\mathcal{A}_{\mathcal{T}}(V)$	515
$\text{pid}(e)$	482	$\bar{\alpha}_{\mathcal{T}}(\mathbf{t})$, $\bar{\alpha}(\mathbf{t})$	515
f_{\perp}	487	$\sqsubseteq_{\mathcal{C}\mathcal{A}_{\mathcal{T}}(V)}$	516
$f(E)$	488	$\Omega_{\mathcal{C}\mathcal{A}_{\mathcal{T}}(V)}$	517
$([D \rightarrow E], \sqsubseteq_{[D \rightarrow E]})$	488	$\alpha_{(d_1 \dots d_n)}$, α_d	520
μf	494	Φ_P^{Σ}	520
$f^n(x)$	495	ADD	524
\times , \rightarrow	505	\mathcal{D}_{Δ_N}	527
VAR_I , VAR_F , VAR	506	($\text{rep}(D_{\mathcal{T}(\tau)})$, $\sqsubseteq_{\text{rep}(\mathcal{T}(\tau))}$)	530
$IV(\mathbf{O})$, $FV(\mathbf{O})$, $V(\mathbf{O})$	506	$\text{rep}(f)$	530
(\mathbf{W}, τ)	506	$\text{rep}(\Sigma) = (\text{rep}(\mathcal{T}), \text{rep}(\mathcal{I}))$	531
$\tau(\mathbf{f})$, $\delta(\mathbf{f})$, $\rho(\mathbf{f})$	506	($D_{r(\tau)}$, $\sqsubseteq_{r(\tau)}$)	531
$\text{ar}(\mathbf{f})$	506	$D_{r(\tau_1) \times \dots \times r(\tau_n)}$	531
$TV(\mathbf{W})$	507	$D_{r(\tau_1) \times \dots \times r(\tau_n) \rightarrow r(\tau)}$	531
$\text{TM}_W^*(V)$, $\text{TM}_W(V)$	507	$D_{r(\delta(\mathbf{F}))}$	531
TM_W^* , TM_W	507	$\text{rep}(\alpha)$	533
\mathbf{N} , \mathbf{NL}	507	\mathcal{D}_{Δ}	533
$\mathbf{n}, s^n(\mathbf{0})$	507	$\Sigma_{NL} = (\mathcal{T}_{NL}, \mathcal{I}_{NL})$	539
$\mathbf{0}, \mathbf{s}, \mathbf{nil}, \mathbf{cons}$	507	Δ_{NL}	540

LIST	540	W_R	554
$\Delta_{\mathcal{S}} = (\mathcal{T}_{\mathcal{S}}, \mathcal{I}_{\mathcal{S}})$	542	$.d_1 d_2 \dots$	554
$\Sigma_{\mathcal{K}(\mathbf{W})} = (\mathcal{T}_{\mathcal{K}(\mathbf{W})}, \mathcal{I}_{\mathcal{K}(\mathbf{W})})$	542	tθ	558
$\Delta_{\mathcal{K}(\mathbf{W})}$	543	u → v	559
W⁺	545	t ⇒ w	559
(TM ^T _{W₊} , ⊑ _{τ₊}), ⊥ _τ	545	$\sigma_{li}^{T,\sigma}, \sigma_{pi}, \sigma_{lo}, \sigma_{po}$	560
$\Sigma_{\mathcal{K}^*(\mathbf{W})} = (\mathcal{T}_{\mathcal{K}^*(\mathbf{W})}, \mathcal{I}_{\mathcal{K}^*(\mathbf{W})})$	546	T_S(P)	561
(D _{τ[*]} , ⊑ _{τ[*]}), ⊥ _{τ[*]}	546	σ_f	564
Σ_N^∞	546	r _P , rr _P	565, 585
n_\perp, ω	547	\mathcal{O}_Δ	566, 585
Δ_N^∞	550	$f_\pi(x_1, \dots, x_n)$	579
$\Sigma_{NL}^\infty, \Delta_{NL}^\infty$	551	# ₂₀ (w), #(t), #(P), #(θ)	580
$\langle d_1, \dots, d_n \rangle_\perp$	552	T _i (P)	584

Index

A

- Algorithms, 3, 68, 79–80, 95, 411
Almost everywhere, 421
Alphabets, 4, 113, 347, 375, 505
 of linear bounded automata, 330
 of pushdown automata, 310
 of Turing machines, 146
Approximation orderings, 470–471, 487, 505,
 see also Complete partial orders; Partial
 orders
Arithmetic hierarchy, 215–230
Arithmetic predicates, 223
Assignments
 complete type, 512, 513, 516, 539
 continuous variable, 514–515, 539
 Herbrand ideal type, 546
 on propositional atoms, 348, 406, 455–456
 type, 511–512
 variable, 514–515
Atoms, 347
Automata theory, 237
Axiomatizable theories, 407–410, 415
 complete, 410
 consistent, 410
 nonrecursive, 415
 ω -consistent, 410
Axiom of choice, 474

B

- Balanced words, 309–310
Bar-Hillel's pumping lemma, 287–290, 299
Base n notation, 116, 328, 580
Binary relations, 472
 restrictions of, 472
Blum axioms, 419–425
Blum, Manuel, 419
Blum speedup theorem, 437–438
Boolean algebra, 350
Bottom elements, 477
Bracket languages, 301–308

C

- Cantor, Georg, 406
Cartesian product, 3
Case, John, 103
Chains, 477, 499
Chang, C. L., 593
Chomsky hierarchy, 327–330
Chomsky normal form grammars, 285–287,
 287–290, 297, 303, 306, 308, 313, 316,
 319
 separators of, 303–308, 313, 319
Chomsky–Schützenberger representation
 theorem, 306

- Church, Alonzo, 411
 Church's thesis, 69, 80, 90, 95, 96, 141, 161,
 162, 197, 407, 445, 447
 Clauses, 355–356
 linked sets of, 366–367, 396, 398, 400
 minimally unsatisfiable sets of, 366–367,
 392, 395
 unit, 361, 362, 363
 Closure properties, 4
 for context-free languages, 291–297
 for context-sensitive languages, 329,
 337–344
 for regular languages, 249–256
 CNF, *see* Conjunctive normal form
 COBOL, 323
 Coding
 of finite functions, 203–204, 580
 of pairs, 59–60, *see also* Pairing functions
 of programs, 65–67
 of sequences, 60–63, *see also* Gödel numbers
 of states of programs, 71
 of strings, 113–115
 of W_N -programs, 579–582
 of W_N -substitutions, 579–582
 of W_N -terms, 579–582
 COF, 227
 Coincidence lemma, 518, 527
 COMP, 48
 Compact elements of a cpo, 486
 Compactness theorem
 for predicate logic, 390, 404–406
 for propositional calculus, 370–373, 390,
 404
 Compilers, 323–326
 Complete partial orders, 475–486, 516, *see*
 also Partial orders; Orderings
 algebraic, 486, 494
 compact elements of, 486
 of continuous variable assignments,
 516–517
 flat, 477, 539
 Complexity
 abstract, 419–438
 measures, 419
 Composition, 39–40, 42, 76, 77, 108, 200, 576
 Computability theory, 17, 18, 31, 70, 98, 113,
 169, 237, 443
 Computable functions, *see* Functions, computable
 Computable real numbers, 554, 587, 589–590
 Computations
 infinite, 469–470, 563
 by linear bounded automata, 339–343
 by nondeterministic Turing machines, 160,
 171, 447–448, 451–456
 by pushdown automata, 311–323
 by \mathcal{S} programs, 27, 28, 29, 70–75, 420,
 468, 557
 for W -terms, 559–575, 584–587, 588
 Configurations
 of linear bounded automata, 330, 331
 of Post-Turing programs, 130
 of pushdown automata, 311
 of Turing machines, 147, 159–160, 444,
 452–456
 Conjunctive normal form, 356–360, 364, 368,
 392, 393, 444, 446, 447–448, 451–456,
 457–458, 461
 co-NP, 450
 Constant symbols, 375, 506
 Context-free grammars, 269–280, 326, 327,
 497–499, *see also* Chomsky normal form
 grammars
 ambiguity of, 300, 303, 307, 319
 branching, 277, 279, 285
 Chomsky normal form, *see* Chomsky nor-
 mal form grammars
 derivations in, 270, 275, 277
 derivation trees in, *see* Derivation trees
 Greibach normal form, 287
 kernels of, 271
 languages generated by, 270
 left-linear, 285
 leftmost derivations in, 275–277
 positive, 269, 271, 273, 277, 327
 productions of, 269
 regular, 280–285, 327
 right-linear, 283, 304
 rightmost derivations in, 275–277
 self-embedding, 284
 unsolvable problems involving, 297–301
 Context-free languages, 269–326, 327–328,
 337, 450, *see also* Context-free gram-
 mars
 closure properties for, 291–297
 and compilers, 323–326
 deterministic, 323
 infinite, 298–299
 inherently ambiguous, 301

nonregular, 270
 and pushdown automata, 308–323
 regular, 280–285
 Context-sensitive grammars, 189–190,
 327–330
 unsolvable problems involving, 339
 Context-sensitive languages, 327–344
 closure properties for, 329, 337–344
 open questions concerning, 343
 Cook–Karp thesis, 445
 Cook’s theorem, 451–456
 Corollaries, 8
 Correctness
 of operational semantics, 557
 of programs, 536
 Course-of-values recursion, 62–63
 cpo, 477, *see also* Complete partial orders
 flat, 477

D

Data structure systems, 531, 536
 infinitary, 544–556, 584–592
 simple, 539–544, 557–575
 Davis, Martin, 593
 Davis–Putnam rules, 360–366, 368, 391, 393,
 396, 444
 Definition by cases, 50–51
 Degree
 of function symbols, 375
 of polynomials, 441
 of relation symbols, 375
 De Morgan identities, 2, 6, 7, 50, 251, 291,
 350, 354, 371, 380
 Derivation trees, 274–279, 298, 498
 pruning and splicing of, 279, 289
 dfa, 243, *see also* Finite automata
 minimal, 266
 Diagonalization, 88–90, 94, 106, 328, 406,
 429
 Diagonal lemma, 492, 502, 518
 Disjunctive normal form, 356–360, 444
 DNF, *see* Disjunctive normal form
 Dovetailing, 80, 81, 408
DTIME, 450
 Duality, general principle of, 350, 356, 358,
 380
 Dyck languages, 303, 306

E

Empty program, 26
 Enderton, Herbert P., 593
 Enumeration principle, 370, 371, 405
 Enumeration theorem, 81
 Equivalence relations, 263
 Euclid, 57
 Exchange lemma, 490, 491, 503
EXPTIME, 450, 451
 Extension lemma, 526–527, 532, 534, 573

F

Fibonacci numbers, 62, 529
 Finite automata, 237–242
 deterministic, 243, 280, 292
 nondeterministic, 242–249, 281, 324
 nonrestarting, 250, 316
 Finiteness theorem, 204
 Finite satisfiability, 370–372
 First-order logic, *see* Quantification theory
 Fixed point induction principle, 499, *see also*
 Induction, fixed point
 Fixed points, 494–503
 as solutions to W-programs, 523
 Fixed point theorem, 101, 501, *see also* Re-
 cursion theorem
 for cpos, 495–496, 497, 499, 501, 524
 Floyd, R. W., 181
 Formulas
 atomic, 376
 ground, 392
 Herbrand instances of, 412–414
 in predicate logic, 376
 propositional, 347, 496–500
 quasi-representing sets, 409
 representing functions, 410
 representing sets, 408, 409
 semantically equivalent, 380
 FORTRAN, 279, 323, 324
 Functions, 3
 binary, 3
 Boolean-valued, 5
 characteristic, 6, 78, *see also* Predicates
 computable, 28–32, 41, 42, 77, 112, 116,
 121, 469, 567
 computed by Post–Turing programs, 134,
 135, 140, 141, 147, 149, 150
 computed by \mathcal{S} programs, 19, 22–25, 30
 computed strictly, 134, 135, 139, 146, 148

- computed by Turing machines, 146–152
 constructible, 428
 continuous, 486–494, 495
 Δ -computable, 574, 575–584
 domain of, 3
 elementary, 59
 exponential, 425, 442, 444, 450
 G -computable, 198
 G -partial recursive, 198
 G -recursive, 198
 higher order, 470, 481, 520
 intuitively computable, *see* Church's thesis
 monotonic, 487–494
 n -ary, 3, 200
 nowhere defined, 3, 31
 one-one, 4, 84
 onto, 4
 partial, 3, 24
 partially computable, 30, 39, 70, 73, 75, 76,
 83, 96, 116, 127, 140, 148, 198, 199, 467,
 501, 578–582
 partially computable in \mathcal{S}_n , 121, 127, 135,
 140, 141
 partially G -computable, 198
 partial recursive, 30, 84
 polynomial time computable, 445
 primitive recursive, 39–63, 84, 105–112,
 117, 188, 262, 576–578
 range of, 3, 82, 83, 84
 recursive, 30, 84, 426, 428, 436, 437
 strict, 487, 539, 564
 strict extensions of, 487
 on strings, 116–117
 total, 3, 5, 30, 40, 42, 90, 198, 199, 200, 201,
 202, 204, 205
 unary, 3
- Function symbols, 375, 506
 arity of, 506
 built-in, 509
 constructor, 509
 degree of, 375
 discriminator, 509
 proper, 506
 selector, 509
- G**
- Γ -trees, 273–274, 278
 paths in, 278
 Gap theorem, 425–428
 Garey, Michael R., 593
- Gödel, Kurt, 60
 Gödel numbers, 60–63, 66, 67, 71, 74–75, 79,
 84, 108, 200, 204, 217–218, 222, 232, 435,
 580
 Gödel's incompleteness theorem, 407–410
 Goldbach's conjecture, 69, 70
 Grammars, 186–191, *see also* Context-free
 grammars; Context-sensitive grammars;
 Regular grammars
 derivations in, 188, 270
 languages generated by, 186, 270, 327–329
 left-linear, 285
 null productions of, 269, 327, 344
 phrase structure, 186, 327
 positive context-free, 269, 327
 productions of, 186, 269
 regular, 280–285, 327
 right-linear, 283, 304
 self-embedding, 284
 start symbol of, 186, 269
 terminals of, 186, 269
 unsolvable problems concerning, 191–192,
 300–301, 322, 339
 variables of, 186, 269
- Graphs, 458–460
 cliques in, 459
 complete, 458–459
 Greatest lower bounds, 476
 Greibach normal form grammars, 287
 G-r.e. sets, 211–215
 Ground clauses, 392
 Ground resolution, 369, *see also* Resolution,
 in propositional calculus
 Ground resolution theorem, 369
 Gunter, Carl A., 593
- H**
- Halmos, Paul R., 593
 Halting problem, 68–70, 78, 82, 89, 97–98,
 99, 197, 467, 488, 523, 554, *see also* Un-
 solvability of, halting problem
 for Post-Turing programs, 144
 for Turing machines, 157–158
 Harrison, Michael, 594
 Herbrand instances of formulas, 412–414
 Herbrand's theorem, 388–398, 404
 Herbrand universes, 388, 389, 390, 392, 393,
 395, 398, 405, 406, 412, 542
 Hilbert, David, 410, 411
 Homomorphisms, 253, 296, 344

Horn clauses, 403
 Horn programs, 404
 Hrbacek, Karel, 594

I

Ideals, 482–483, 546–556
 principal, 482, 552
 Immerman, Neil, 339
 Index sets, 95, 103, 230–231
 Induction, 9–13
 complete, 11–12
 course-of-values, 11–12, 302
 fixed point, 499–500, 536–537
 hypothesis, 10
 loading, 11, 109
 mathematical, 9–13, 52, 86, 206, 215, 245,
 256, 264, 271, 293, 304, 305, 433, 495,
 540, 571, 576
 structural, 500–501, 516, 517, 518, 534,
 535, 558, 567, 571

INF, 225

Initial functions, 42, 76, 108

Instantaneous descriptions, 27, *see also*
 Snapshots of a program

Instructions, 17–18, 26, 65–67, 197
 conditional branch, 18, 35
 decrement, 18
 increment, 18
 labeled, 26
 numbers of, 65–67, 74, 201
 oracle, 197–198, 201
 of Post-Turing programs, 129
 of \mathcal{I}_n programs, 122
 unlabeled, 26
 while, 103

Interpretations

continuous, 513, 539
 Herbrand ideal, 546
 for predicate logic vocabularies, 377, 382,
 411
 for typed vocabularies, 512–513, 539, 546

Interpreters, 70

Intractability, 444, 457, 459

Isomorphisms, 485

Iteration theorem, *see* Parameter theorem**J**

Jech, Thomas, 594
 Johnson, David S., 593
 Jumps of sets, 213

K

Kleene's hierarchy theorem, 216
 Kleene's theorem, 253–260
 Kuroda, S. Y., 331, 339

L

Labels, 18, 25
 Landweber, P. S., 331
 Languages, 4, *see also* Vocabularies
 accepted by deterministic Turing ma-
 chines, 189
 accepted by finite automata, 239
 accepted by linear bounded automata,
 330–337
 accepted by nondeterministic finite au-
 tomata, 243
 accepted by nondeterministic Turing ma-
 chines, 160–161, 186, 189
 accepted by pushdown automata, 311–323
 accepted by Turing machines, 153–157, 189
 context-free, *see* Context-free languages
 context-sensitive, *see* Context-sensitive
 languages
 generated by grammars, 186, 189, 270,
 327–328
 inherently ambiguous, 301
 in NP, *see* NP
 NP-complete, *see* NP-completeness
 NP-hard, 449
 polynomial-time decidable, *see* P
 r.e., *see* r.e. languages
 recursive, 156, 190, 328, 446
 recursively enumerable, *see* r.e. languages
 regular, *see* Regular languages
 spanning sets for, 263
 Lattices, 485, 502
 Lazy functional languages, 553
 Least upper bounds, 476
 Lee, R. C. T., 593
 Lemmas, 8
 Lewis, Harry R., 594
 Lexical analysis, 323
 L'Hospital's rule, 442
 Linear bounded automata, 330–343
 deterministic, 336, 337, 343
 Linears orders, 472, *see also* Partial orders
 Linked conjunct procedures, 400
 LISL, 507

Lists

- finite, 541, 552
- infinite, 553, 588–590
- prefix, 553

Literals, 354, 447–448

Loeckx, Jacques, 594

Logic, *see* Propositional calculus; Quantification theory

Logical consequence, 382–388

Logic programming languages, 403

Loveland, Donald W., 594

M

Machtey, Michael, 427, 594

Macro expansions, 20–24, 33, 35, 121–123

Macros, 20, 32–37, 122–123, 127

- for Post-Turing programs, 136, 137

Many-one reducibility, 90–95, 207–211

Markov, A. A., 178

Mates, 355, 357–358, 366–367, 397–398, 399,
see also Linked conjunct proceduresMathematical induction, *see* Induction, mathematical

m-completeness, 92, 93, 97, 210, 217

Meaning functions

- denotational, 527, 533, 536, 557, 573–574, 586

- operational, 566, 573–574, 585–586

Metavariables, 506

Minimalization, 55–59, 75, 76, 77, 200, 578

- bounded, 55–58

- proper, 77

- unbounded, 57–58

Minimal unsatisfiability, 366–367, 392, 395,
see also Satisfiability, truth functional

Models, 380, 382, 384, 385, 406, 412

- countable, 406

Myhill, John, 232, 263

Myhill-Nerode theorem, 263–264

N

Natural numbers, 1

ndfa, 243, *see also* Finite automata, non-deterministic

Nerode, Anil, 263

Normal form theorem, 75

Normal processes, 192–195

Normal productions, 193

NP, 446–451, 456, 457–463

NP-completeness, 208, 448, 449, 451, 457–463

NP-complete problems

CHROMATIC-NUMBER, 461, 462

COMPLETE-SUBGRAPH, 458–459, 462

EXACT-COVER, 462

HALF-SAT, 456

HAMILTONIAN-CIRCUIT, 460, 462

INTEGER-PROGRAMMING, 460

KNAPSACK, 463

LONGEST-COMMON-SUBSEQUENCE, 462

MAX-CLIQUE, 459

MULTIPROCESSOR-SCHEDULING, 463

PARTITION, 460, 462, 463

QUADRATIC-DIOPHANTINE-EQUATIONS, 461

RECORD-ALLOCATION, 463

SAT, 446–456, 458

SET-COVER, 460

STRAIGHTLINE-PROGRAM-IN-EQUIVALENCE, 461

SUBGRAPH-ISOMORPHISM, 462

SUBSET-SUM, 462

TASK-SEQUENCING, 463

3-DIMENSIONAL-MATCHING, 460

3-SAT, 457–458, 460, 461, 462

TRAVELING-VENDOR, 462

2-COLORABILITY, 461

2-SAT, 461–462

VERTEX-COVER, 459–460, 462

NP-hardness, 449, 451, 457

NPSPACE, 451

NTIME, 450

n-tuples, 2

Numeral systems, 408

O

1-completeness, 209, 226

One-one reducibility, 207–211

Oracles, 197–200

Ordered pairs, 2

Orderings, *see also* Partial orders

- Cartesian product, 472, 479

- continuous function space, 488

- \mathcal{D} -choice function, 474, 481

- function space, 473, 480

- lexicographic, 481

P

- P**, 444–446, 448, 449, 456, 461
 Pairing functions, 59–60, 65, 66, 67, 74–75,
 83, 106, 107, 110, 203, 228, 232, 435
 Pairing function theorem, 60
 Palindromes, 241, 265
 Papadimitriou, Christos H., 594
 Parameter theorem, 85–88, 92, 93, 96, 99,
 209, 225, 226, 229, 231, 428, 435
 relativized, 205–206, 209
 Parsing, 324–326
 Partially computable functions, *see* Functions,
 partially computable
 Partial orders, 472–475, *see also* Complete
 partial orders
 bottom elements of, 471, 477
 bounded-complete, 485, 494
 chains in, 477, 499
 complete, *see* Complete partial orders
 ideal completions of, 483
 ideals of, 482–483, 546–556
 isomorphic, 485, 486, 493, 544
 principal ideals of, 482
 Partial recursive functions, *see* Functions,
 partial recursive
 Pascal, 279, 323, 326, 438
 Pigeon-hole principle, 260, 264, 266, 451
 Π_n , 215–230
 as a collection of sets, 215
 as a property of predicates, 217
 Polynomials, 441, 442
 Polynomial-time computability, 445
 Polynomial-time decidability, 444
 Polynomial-time reducibility, 208, 448–449,
 457
 Post correspondence problem, 181–186,
 299–301, 322
 Post correspondence system, 181, 300
 Post, Emil L., 129, 178, 181
 Post's lemma, 177
 Post's theorem, 217–224, 228
 Post–Turing programs, 129–144, 145, 149,
 150, 161, 200
 Post words, 171–175, 177, 187
 Power sets, 472
 PRC classes, 42–44, 49–56, 62, 63, 79, 84,
 199
 Predecessor function, 46
 Predicate logic, 375–415

- Predicates, 5–7, 10, 34–35, 68, 217–224, 419
 admissible, 499, 502, 537
 arithmetic, 223–224
 computable, 34–35, 50, 68, 85
 primitive recursive, 49–51, 78, 187,
 259–260
 recursive, 419
 Prenex normal form, 384
 Prenex sentences, 384
 Prime numbers, 8, 54, 56–57, 69, 265, 267,
 588
 Primitive recursive functions, *see* Functions,
 primitive recursive
 Productions, *see* Context-free grammars,
 productions of; Semi-Thue productions
 null, 269
 Programming systems, 88
 acceptable, 88, 105, 128, 144, 153
 Programs, 17–28, 65–67, 197, *see also* W-
 programs
 computations of, 27
 correct, 536
 as a data structure system, 542
 functions computed by, 19, 22–24, 28–32,
 198
 G-computations of, 198, 201
 instantaneous descriptions of, 27
 length of, 26
 numbers of, 65–67, 68, 70, 74–76, 86, 89,
 90, 95–97, 97–105, 201, 224, 225
 with oracles, 197–198
 Post-Turing, *see* Post–Turing programs
 snapshots of, 27–28, 29, 70, 74–76,
 197–198, 420
 states of, 26, 197
 straightline, 32
 for string computations, 121–26
 universal, 70
 Program verification, 536
 Projection functions, 42, 478
 Prolog, 403
 Proof by contradiction, 8–9
 Proofs, 8
 Propositional calculus, 347–373
 Propositional connectives, 347
 Propositional variables, 347
PSPACE, 450, 451
 Pumping lemma, 260–262, 264, 267, *see also*
 Bar-Hillel's pumping lemma

Pure literal rule, 362, 363
 Pushdown automata, 308–323
 atomic, 314, 315, 316
 computations by, 311–323
 deterministic, 311, 313, 315
 generalized, 321
 languages accepted by, 311–323
 nondeterministic, 312, 315
 transitions of, 310

Q

Quadruples, 145, 330–333, 338, 454–455
 Quantification theory, 375–415
 Quantifiers, 7–8
 alternating, 223
 bounded, 7, 53–55
 in predicate logic, 376
 Quintuples, 149

R

Rates of growth, 439–443
 Recursion, 40, 41, 44, 45, 76, 77, 100, 200,
 348, 496, 501, 576
 course-of-values, 62, 109
 primitive, 40, 41, 100, 106, 108
 simultaneous, 62
 unnested double, 63
 Recursion equations, 40, 41, 44, 45, 46, 47,
 62, 63, 100, 104, 309, 314, 505, 508, *see also W-recursion equations*
 Recursion theorem, 97–105, 111, 428, 431,
 see also Fixed point theorem
 first, 501, 503
 second, 501
 Recursion theory, 31, *see also Computability theory*
 Recursive functions, *see Functions, recursive*
 Recursive function theory, *see Computability theory*
 Recursive isomorphisms, 231–234
 Recursive languages, *see Languages, recursive*
 Recursively enumerable languages, *see r.e. languages*
 Recursively enumerable sets, *see r.e. sets*
 Recursive operators, 503
 Recursive permutations, 231–234
 Recursive relatedness theorem, 422, 424, 425,
 437

Recursive sets, *see Sets, recursive*
 Redexes, 559
 Reducibilities, 90–95, 96, 207–211, *see also Many-one reducibility; One-one reducibility; Polynomial-time reducibility*
 completeness with respect to, 92–93, 208, 210
 Regular expressions, 256–260
 Regular grammars, 280–285, 327
 Regular languages, 237–267, 280–285, 292,
 304, 306, 327–328, 450, *see also Regular grammars*
 closure properties for, 249–260
 examples of, 247–249
 and finite automata, 237–242
 infinite, 261
 and nondeterministic finite automata, 242–247
 represented by regular expressions, 257
 spanning sets for, 263–267
 r.e. languages, 154–157, 161, 188, 189, 327,
 344, 371, 391–392, 407–408
 Relation symbols, 375
 Relativization, 199
 of computability, 198
 of enumeration theorem, 213
 of parameter theorem, 205–206
 of recursive enumerability, 211–215
 of step-counter theorem, 202–204
 of universality theorem, 201–202
 Remainder, 56, 114
 Representable elements, *see W-structures, representable elements of*
 r.e. sets, 78–85, 88, 90–95, 209–211, 231, 391,
 407, 408, 409, 592
 relative to an oracle, 211–215
 Resolution
 derivations, 367, 402
 in predicate logic, 400–404
 in propositional calculus, 367–370, 393,
 402, 444
 refutations, 367, 402
 Resolvents, 367
 Rewrite rules, 169, 559, *see also W-term rewriting systems*
 Rewriting strategies, 559, 560, 564–565
 Rice–Shapiro theorem, 231
 Rice’s theorem, 95–97, 103, 230–231
 Right quotients, 266, 297
 Robinson’s general resolution theorem, 402

- Rogers, Hartley, 594
Rooted sentences, 412–414
- S**
- SAT, 446–456, 458
Satisfiability
 finite, 370
 in predicate logic, 380, 383, 384, 385,
 389–391, 394, 411
 problem, 358
 truth-functional, 348–352, 352–353,
 355–356, 358, 364, 366–367, 372,
 389–391, 405, 444, 446–456
- Scaling factors, 421
Schmidt, David, 594
Scott domains, 494
Self-embedding grammars, 284
Self-reference, 98, 99
Self-reproducing program, 100
Semantics, *see also* Meaning functions
 denotational, 468, 499, 505–556
 of formulas in predicate logic, 377–382
 operational, 468, 499, 557–592
 of programming languages, 467–468
 of propositional formulas, 348
 of regular expressions, 257
 of \mathcal{L} programs, 26–28
 of W-terms, 511–520
- Semi-decision procedures, 80, 392
Semi-Thue processes, 169–180, 182, 186, 334,
 see also Thue processes
 derivations in, 170
Semi-Thue productions, 169
 inverses of, 175–176, 177, 335
Sentences, 377
 models of, 380
 prenex, 384
 rooted, 412, 413, 414
 satisfiable, *see* Satisfiability, in predicate
 logic
 semantically equivalent, 380
 Skolemized, 384–387, 393, 405
 undecidable, 409
 universal, 385, 388
 valid, 380, 383, 391
- Sentential logic, *see* Propositional calculus
Sequence number theorem, 62
Sets 1, 78
 characteristic functions of, 6, 78, 79
 complement of, 2
- computable, 78
difference of, 2
empty, 1
equality of, 1
finite, 2
G-r.e., *see* G-r.e. sets
G-recursive, 211–214
infinite, 2
intersection of, 2
r.e., *see* r.e. sets
recursive, 78, 80, 88, 91, 210
recursively isomorphic, 231
union of, 2
- Shoenfield, Joseph R., 594
Sieber, Kurt, 594
 Σ_n , 215–230
 as a collection of sets, 215
 as a property of predicates, 217
Simulation, 70–75, 127, 135–144, 163–167,
 171–176, 333–336
- Skolemization, 384–387
 generalized, 386–387, 405
- Skolem–Löwenheim theorem, 406
- s-m-n theorem, *see* Parameter theorem
- Snapshots of a program, 27–28, 70, 74–76,
 197, 420
- initial, 29
successor of, 27
terminal, 27, 75, 76, 197
- Soare, Robert I., 594
Socrates, 393
Solutions to W-programs, 520–530, 534, 536
Spanning sets, 263
Speedup theorem, 428–438
Splitting rule, 361, 363, 364
Stacks, 310–311
Start symbols, 269
Statements, 25, *see also* Instructions
States, 26, 145, 197, 238
 accepting, 238, 310
 dead, 248
 final, 238, 310, 330, 331
 initial, 29, 238, 310, 331
State transition diagrams, 147, 240
Step-counter predicates, 74–75, 78, 81, 82,
 83, 224, 225
 relativized, 201–205, 211, 212
- Step-counter theorem, 74
 relativized, 202–204
- Stoy, Joseph E., 594

Strings, 4, 12, 113–144
 computable functions on, 116, 121
 computations on, 113, 121
 concatenation of, 117–118
 length of, 4, 118
 numerical representation of, 113–117
 partially computable functions on, 116, 121
 primitive recursive functions on, 117
 programming languages for, 121–126

Subsets, 1

proper, 2

Substitution lemma, 558, 570

Substitutions

on alphabets, 252, 296, 344
 answer, 404
 to individual variables, 395, 397–398,
 399–404, *see also W*-substitutions

Subsumption, 366

Symbols, 4, *see also* Constant symbols; Function symbols; Relation symbols

Syntax

of propositional calculus, 347–348
 of quantification theory, 375–377
 of \mathcal{L}_n programs, 121
 of \mathcal{L} programs, 25–26
 of W-programs, 505–511

T

Tautological consequence, 352, 373
 Tautological inference, 352–353
 Tautologies, 348, 352, 355–356
 Terminals, 186, 269
 Terms, 376, *see also* W-terms
 Theorems, 8
 Theoretical computer science, 189, 359, *see also* Automata theory; Complexity; Computability theory; Languages; Semantics, of programming languages

Thue, Axel, 169

Thue processes, 177–181, 411

TOT, 90, 224–229

Tractability, 444

Transition functions, 238

Truth values, 5

Turing, Alan, 129, 153, 411

analysis of computation process, 129, 145

Turing machines, 145–168, 176, 191, 197,
 415, 444, 445, 447, *see also* Configurations, of Turing machines; Linear bounded automata

deterministic, 146, 159, 174, 176, 177, 178,
 189, 415
 functions computed by, 146–152
 languages accepted by, 153–157, 186–187,
 189
 multiple tape, 167, 168
 multiple track, 163–168, 337–343
 nondeterministic, 146, 159–162, 171–176,
 186, 189, 330, 446–456
 one-way infinite, 162–167
 quintuple, 149, 150
 universal, 152–153
 Turing reducibility, 207–211
 Types, 505–506
 domain, 506
 function, 506
 individual, 506
 product, 506
 range, 506

U

Unconditional branch, 19–20

Undecidable sentences, 409

Unification, 399–404, 560

algorithm, 399

Unifiers, 403

Unit clauses, 361, 362, 363

Unit rule, 362

Universal computers, 153

Universality theorem, 70, 209

relativized, 201–202

Universal programs, 70

Universal sentences, 385

Unsolvability of

halting problem, 68, 82, 99, 157–158, 197,
 420

Post's correspondence problem, 181–186

problems involving grammars, 191–192,
 297–301, 339

problems involving programs, *see* Rice's theorem

satisfiability problem in predicate logic,
 392, 410–415

word problems, 176–180, 195, 411

V

Validity, *see* Sentences, valid

Variables, 17, 25, 65

auxiliary function, 508

bound, 376–377
free, 376–377
function, 506
in grammars, 186, 269
individual, 506
initialization of, 18, 121
input, 17, 25
local, 17, 25
occurrences of, 376–377
output, 17, 25
in predicate logic, 375–376
principal function, 508
propositional, 347
of \mathcal{S} , 17, 25
type, 505
Vocabularies, 375, 506
interpretations of, 377
standard, 509, 511, 543, 550
standard constructor, 509, 511
typed, 506

W

W-formulas, 376, *see also* Formulas
Word problems, 176–180, 411
of normal processes, 193–195
Words, 4, *see also* Strings
length of, 4
W-programs, 508, 580
denotational semantics for, 530–538
operational semantics for, 557–592

partially correct, 536
solutions to, 520–530, 534, 536
totally correct, 536
W-recursion equations, 507–508
W-sentences, 377, *see also* Sentences
W-structures, 513
complete, 513, 539
computable elements of, 587
continuous, 513, 539
data structure systems based on, 531
Herbrand ideal, 546
infinitary, 544–545, 548, 587
representable elements of, 530–531, 534,
 539, 551, 554, 587
simple, 539
simple Herbrand, 542
W-substitutions, 558, 580
W-term rewriting systems, 559
for infinitary data structure systems,
 584–585
for simple data structure systems, 561–562
strategies for, 559, 560, 564–565
W-terms, 376, 507, 580
ground, 507
normal, 559, 565
semantics of, 511–520

Y

Young, Paul, 427, 594