A Problem Course in Mathematical Logic

Volume II Computability and Incompleteness

Stefan Bilaniuk

Author address:

Department of Mathematics, Trent University
Peterborough, Ontario, Canada K9J 7B8
E-mail address: sbilaniuk@trentu.ca

1991 *Mathematics Subject Classification.* 03.

A Problem Course in Mathematical Logic
Volume II: Computability and Incompleteness
VERSION 1.2

ABSTRACT. This is the Volume II of a text for a problem-oriented undergraduate course in mathematical logic. It covers the basics of computability, using Turing machines and recursive functions, and Gödel's Incompleteness Theorem, and could be used for a one-semester course on these topics. Volume I, *Propositional and First-Order Logic*, covers the basics of these topics through the Soundness, Completeness, and Compactness Theorems.

Information on availability and the conditions under which this book may be used and reproduced are given in the preface.

This book was typeset using LaTeX, using the AMS-LaTeX and AMSFonts packages of the American Mathematical Society.

Contents

Preface

This book is intended to be the basis for a problem-oriented full-year course in mathematical logic for students with a modicum of mathematical sophistication. Volume II covers the basics of computability, using Turing machines and recursive functions, and incompleteness. It could be used for a one-semester course on these topics. Volume I covers the basics of propositional and first-order logic through the Soundness, Completeness, and Compactness Theorems, plus some material on applications of the Compactness Theorem; it could also be used as for a one-semester course on these topics. However, part of Volume II, particularly the chapters on incompleteness, assume some familiarity with the basics of first-order logic.

In keeping with the modified Moore-method, this book supplies definitions, problems, and statements of results, along with some explanations, examples, and hints. The intent is for the students, individually or in groups, to learn the material by solving the problems and proving the results for themselves. Besides constructive criticism, it will probably be necessary for the instructor to supply further hints or direct the students to other sources from time to time. Just how this text is used will, of course, depend on the instructor and students in question. However, it is probably *not* appropriate for a conventional lecture-based course nor for a really large class.

The material presented here is somewhat stripped-down. Various concepts and topics that are often covered in introductory courses on computability are given very short shrift or omitted entirely, among them models of computation other than Turing machines and recursive functions, formal languages, and computational complexity.[1] Instructors might consider having students do projects on additional material if they wish to to cover it. It might also be expedient, in view of the somewhat repetitive nature of devising Turing machines and recursive

[1]Future versions of both volumes may include more – or less! – material. Feel free to send suggestions, corrections, criticisms, and the like — I'll feel free to ignore them or use them.

functions for various purposes, to be selective about the problems the students are required to do or to divide them up among the students.

Acknowledgements. Various people and institutions deserve the credit for this text: All the people who developed the subject. My teachers and colleagues, especially Gregory H. Moore, whose mathematical logic course convinced me that I wanted to do the stuff. The students at Trent University who suffered, suffer, and will suffer through assorted versions of this text. Trent University and the taxpayers of Ontario, who paid my salary. Ohio University, where I spent my sabbatical in 1995–96. All the people and organizations who developed the software and hardware with which this book was prepared. Anyone else I've missed.

Any blame properly accrues to the author.

Availability. A text-only information file and the LaTeX source files of the latest release of both volumes can be found in the directory

- `ftp://ftp.trentu.ca/pub/pcml/`

Please note that in addition to LaTeX you will need the $\mathcal{AMS}$-LaTeX and $\mathcal{AMS}$Fonts packages to typeset and print either volume.

If you have any problems, feel free to contact the author at the addresses given on the title page, preferably by e-mail, for assistance or even to ask for a paper copy.

Author's Opinion. It's not great, but the price is right!

[2]If you violate these restrictions, I will be flattered, but you will still be in the wrong.

[3]Who knows, it might even find a publisher ...

Introduction

The Entscheidungsproblem. Recall that a logic satisfies the Completeness Theorem if, whenever the truth of a set of sentences Σ implies the truth of a sentence φ, there is a deduction of φ from Σ. Propositional and first-order logics both satisfy the Completeness Theorem, though second- and higher-order logics do not. In the case of propositional logic, the Completeness Theorem leads to a rote procedure for determining whether $\Sigma \vdash \varphi$ or not, so long as Σ is finite: write out a complete truth table for all of Σ together with φ and check whether every assignment that makes every sentence of Σ true also makes φ true. It is natural to ask whether something of the sort can be done for first-order logic. If so, it might be very useful: since most of mathematics can be formalized in first-order logic, such a method would have the obvious use of putting mathematicians out of business ... This question is the general *Entscheidungsproblem*[4] for first-order logic:

[. Entscheidungsproblem] Given a reasonable set Σ of formulas of a first-order language $\mathcal{L}$ and a formula φ of $\mathcal{L}$, is there an effective method for determining whether or not $\Sigma \vdash \varphi$?

Of course, the statement of the problem begs the question of what is a "reasonable" set and an "effective method". In this volume we'll explore two formalizations of the notion of "effective method", namely Turing machines and recursive functions, and then use these to answer the Entscheidungsproblem for first-order logic. These will also give us a clue as to what "reasonable" ought to mean. The answer to the general problem is negative, though decision procedures do exist for some particular first-order languages and sets Σ.

Historically, the Entscheidungsproblem arose in two ways in the course of David Hilbert's scheme to secure the foundations of mathematics by axiomatizing mathematics in first-order logic and showing that the axioms do not give rise to any contradictions. First, given some plausible set of axioms, it is necessary to show that they do not

[4] "Entscheidungsproblem" is German for "decision problem".

lead to any particular contradiction, say $\alpha \wedge (\neg\alpha)$. Second, it is desirable to know whether such a set of axioms is *complete*; *i.e.* given any sentence φ of the language, that the axioms either prove or disprove φ.

In the course of trying to find a suitable formalization of the notion of "effective method", mathematicians developed several different abstract models of computation in the 1930's, including recursive functions, λ-calculus, Turing machines, and grammars. Although these models are very different from each other in spirit and formal definition, it turned out that they were all essentially equivalent in what they could do. This suggested the (empirical!) principle:

[. Church's Thesis] A function is effectively computable in principle in the real world if and only if it is computable by (any) one of the abstract models mentioned above.

Of course, this is not a *mathematical* statement ... We will study Turing machines and recursive functions, and then use this knowledge to formulate and answer a more precise version of the general Entscheidungsproblem for first-order logic.

The development of the theory of computation actually began before the development of electronic digital computers. In fact, the computers and programming languages we use today owe much to the abstract models of computation which preceded them. For two, the standard von Neumann architecture for digital computers was inspired by Turing machines and the programming language LISP borrows much of its structure from λ-calculus.

Approach. This book supplies definitions and statements of results, plus some explanations and a number of problems and examples, but no proofs of the results. The hope is that you, gentle reader, will learn the material presented here by solving the problems and proving the results for yourself. Brief hints are supplied for almost all of the problems and results, but if these do not suffice, you should consult your peers, your instructor, or other texts.

Prerequisites. In principle, little is needed by way of prior mathematical knowledge to define and prove the basic facts about computability. Some knowledge of the natural numbers and a little set theory suffices. The material leading up to the Incompleteness Theorem — the resolution of the general Entscheidungsproblem for first-order logic — does require grounding in first-logic, such as that provided in Volume I, as well as in computability.

What really *is* needed to get anywhere with all of the material developed here is competence in handling abstraction and proofs, including proofs by induction. The experience provided by a rigorous introductory course in algebra, analysis, or discrete mathematics ought to be sufficient. Some problems and examples draw on concepts from other parts of mathematics; students who are not already familiar with these should consult texts in the appropriate subjects for the necessary definitions.

Other Sources and Further Reading. [**3**], [**5**], [**7**], and [**8**] are texts which go over at least some of the material, while [**1**] is a good if terse reference for more advanced material. Entertaining accounts of much of the material may be found in [**6**] and [**9**]; the latter discusses the possibility that Church's Thesis may not be true. Many of the original sources for the material in this volume can be found in the anthology [**4**].

Computability

CHAPTER 10

Turing Machines

Of the various ways to formalize the notion an "effective method", the most commonly used are the simple abstract computers called Turing machines, which were introduced more or less simultaneously by Alan Turing and Emil Post in 1936.[1] Like most real-life digital computers, Turing machines have two main parts, a processing unit and a memory (which doubles as the input/output device), which we will consider separately before seeing how they interact. The memory can be thought of as a tape, without end in one direction, which is divided up into cells like the frames of a movie. The Turing machine proper is the processing unit. It has a scanner or "head" which can read from or write to a single cell of the tape, and which can be moved to the left or right one cell at a time.

Tapes. The first thing we have to do in describing a Turing machine is to specify what symbols it is able to read and write on its tape.

DEFINITION 10.1. An *alphabet* is a non-empty finite set Σ, the elements of which are called *symbols*, such that $0 \notin \Sigma$.

The reason we don't allow Σ to contain 0 is that we will use 0 to mark all the otherwise blank cells on a tape.

DEFINITION 10.2. Given an alphabet Σ, a *tape* (with entries from Σ) is an infinite sequence

$$\mathbf{a} = a_0\, a_1\, a_2\, a_3 \dots$$

such that for each integer i the *cell* $a_i \in \{0\} \cup \Sigma$. The ith cell is said to be *blank* if a_i is 0, and *marked* if a_i is a symbol from Σ.

A blank tape is one in which every cell is 0. It will be shown later on that it is possible to restrict the alphabet to just one non-blank symbol without essentially diminishing what a Turing machine can accomplish, but it is usually convenient to have more symbols about when actually devising a Turing machine for a particular task.

[1]Both papers are reprinted in [**4**]. Post's brief paper gives a particularly lucid informal description.

EXAMPLE 10.1. A blank tape looks like:

$$000000000000000000000000\cdots$$

The 0th cell is the leftmost one, cell 1 is the one immediately to the right, cell 2 is the one immediately to the right of cell 1, and so on.

Letting our alphabet be $\Sigma = \{1, x, y\}$, a slightly more exciting tape is the following:

$$010xx1011x1y01yyx000000000000000\cdots$$

In this case, cell 1 contains a 1, as do cells 5, 8, 9, 11, and 14; cells 4, 5, 10, and 17 contain xs; cells 12, 15, and 17 contain ys; and all the rest contain 0s.

PROBLEM 10.1. *Write down tapes satisfying the the following; you may use any appropriate alphabets.*

1. *Entirely blank except for cells* 3, 12, *and* 20.
2. *Entirely marked except for cells* 0, 2, *and* 3.
3. *Entirely blank except for a block of five consecutive cells just to the right of cell* 0.
4. *Entirely blank except that 1025 is written out in binary just to the left of cell* 2.

To keep track of which cell the Turing machine's scanner is at, plus some other information, we will usually attach additional information to our description of the tape.

DEFINITION 10.3. A *tape position* is a triple $(i, s, \mathbf{a})$, where i and s are natural numbers with $s > 0$, and $\mathbf{a}$ is a tape. Given a tape position $(i, s, \mathbf{a})$, we will refer to cell i as the *scanned cell* and call s the *state*.

The number s mentioned above will be used to keep track of which instruction the Turing machine is to execute next.

Conventions for tapes. Unless stated otherwise, we will assume that all but finitely many cells of any given tape are blank, and that any cells not explicitly described or displayed are blank. We will usually depict as little of a tape as possible and omit the $\cdots$s we used above. Thus

$$010xx1011x1y01yyx$$

represents the tape given in the Example 10.1. In many cases we will also use z^n to abbreviate n consecutive zs, so the same tape could be represented by

$$010x^2101^2x1y01y^2x\,.$$

Similarly, if σ is a finite sequence of elements of $\Sigma \cup \{0\}$, we may write σ^n for the sequence consisting of n copies of σ stuck together end-to-end. For example, $(010)^3$ is 010010010.

In displaying tape positions we will usually underline the scanned cell and write s to the right of the tape. For example, we would display the tape position using the tape from Example 10.1 with cell 5 being scanned and state 2 as follows:

$$010x\underline{x}101^2x1y01y^2x : 2$$

Problem 10.2. *Using the tapes you gave in the corresponding part of Problem 10.1, write down tape positions satisfying the following conditions.*

1. *Cell* 7 *being scanned and state* 4*.*
2. *Cell* 4 *being scanned and state* 1*.*
3. *Cell* 0 *being scanned and state* 3*.*
4. *Cell* 3 *being scanned and state* 413*.*

Turing machines. The "processing unit" of a Turing machine is just a finite list of specifications describing what the machine will do in various situations. (Remember, this is an *abstract* computer ...) The formal definition may not seem to amount to this at first glance.

Definition 10.4. A *Turing machine* (with alphabet Σ) is a function M such that for some natural number n,

$$\mathrm{dom}(M) \subseteq \{1, \dots, n\} \times (\{0\} \cup \Sigma)$$

and

$$\mathrm{ran}(M) \subseteq (\{0\} \cup \Sigma) \times \{-1, 1\} \times \{1, \dots, n\} .$$

Note that M need not be defined for all possible pairs

$$(s, j) \in \{1, \dots, n\} \times (\{0\} \cup \Sigma) .$$

We will sometimes refer to a Turing machine simply as a *machine* or TM . If $n \geq 1$ is least such that M satisfies the definition above, we shall say that M is an *n-state Turing machine* and that $\{1, \dots, n\}$ is the set of *states* of M.

Intuitively, we have a processing unit which has a finite list of basic instructions, the states, which it can execute. Given a combination of current state and the symbol marked in the cell of the tape currently being scanned that it is equipped to handle, the processor specifies

- a symbol to be written in the currently scanned cell, overwriting the symbol being read, then
- a move of the scanner one cell to the left or right, and then
- the next instruction to be executed.

That is, $M(s, a) = (b, d, t)$ means that if our machine is in state s (*i.e.* executing instruction number s) and the scanned cell, say cell i, contains a, then the machine M should

- set $a_i = b$ (*i.e.* write b instead of a in the scanned cell), then
- move the scanner to a_{i+d} (*i.e.* move one cell left if $d = -1$ and one cell right if $d = 1$), and then
- enter state t (*i.e.* go to instruction t).

If our processor isn't equipped to handle input a for instruction s (*i.e.* $M(s, a)$ is undefined), then the computation in progress will simply stop dead.

EXAMPLE 10.2. We will usually present Turing machines in the form of a table, with a row for each state and a column for each possible entry in the scanned cell. Instead of -1 and 1, we will usually use L and R when writing such tables in order to make them more readable. Thus the table

M	0	1
1	$1R2$	$0R1$
2	$0L2$	

defines a Turing machine M with alphabet $\{1\}$ and two states such that $M(1, 0) = (0, 1, 2)$, $M(1, 1) = (0, 1, 1)$, and $M(2, 0) = (0, -1, 2)$, but $M(2, 1)$ is undefined. (So M has domain $\{(1, 0), (1, 1), (2, 0)\}$ and range $\{(1, 1, 2), (0, 1, 1), (0, -1, 2)\}$.) If the machine M were faced with the tape position

$$010\underline{0}1111 : 1 ,$$

it would, being in state 1 and scanning a cell containing 0,

- write a 1 in the scanned cell,
- move the scanner one cell to the right, and
- go to state 2.

This would give the new tape position

$$0101\underline{1}111 : 2 .$$

Since M doesn't know what to do on input 1 in state 2, the prospective computation could go no further.

PROBLEM 10.3. *In each case, give the table of a Turing machine M meeting the given requirement.*

1. *M has alphabet $\{x, y, z\}$ and has three states.*
2. *M has alphabet $\{1\}$ and changes* 0 *to* 1 *and* vice versa *in any cell it scans.*
3. *M is as simple as possible. How many possibilities are there here?*

Computations. Informally, a computation is a sequence of actions of a machine M on a tape according to the rules above, starting with instruction 1 and the scanner at cell 0 on the given tape. A computation ends (or *halts*) when and if the machine encounters a tape position which it does not know what to do in or runs off the left end of the tape. (If it never does either, the computation will never end — not quite like real computers, Turing machines succeed only when they crash!) The formal definition makes all this seem much more formidable.

DEFINITION 10.5. Suppose M is a Turing machine. Then:

- If $p = (i, s, \mathbf{a})$ is a tape position using the same alphabet as M and $M(s, a_i) = (b, d, t)$ is defined, $\mathbf{M}(p) = (i + d, t, \mathbf{a}')$ is the *successor tape position*, where $a'_j = a_j$ if $j \neq i$ and $a'_i = b$.
- A *partial computation* with respect to M is a sequence $p_1 p_2 \dots p_k$ of tape positions such that $p_{\ell+1} = \mathbf{M}(p_\ell)$ for each $\ell < k$.
- A partial computation $p_1 p_2 \dots p_k$ with respect to M is a *computation* (with respect to M) with *input tape* $\mathbf{a}$ if $p_1 = (0, 1, \mathbf{a})$ and $\mathbf{M}(p_k)$ is undefined. The *output tape* of the computation is the tape of the final tape position p_k.

Note that computations must take only finitely many steps.

EXAMPLE 10.3. Let's see the machine M of Example 10.2 perform a computation. Our input tape will be $\mathbf{a} = 1100$, that is, the tape which is entirely blank except that $a_0 = a_1 = 1$. The initial tape position of the computation of M with input tape $\mathbf{a}$ is then

$$\underline{1}100 : 1 .$$

The subsequent steps in the computation are:

$$0\underline{1}00 : 1$$
$$00\underline{0}0 : 1$$
$$001\underline{0} : 2$$
$$00\underline{1}0 : 2$$

We leave it to the reader to check that this is indeed a partial computation with respect to M. Since $M(2, 1)$ is undefined the process terminates at this point and this partial computation is indeed a computation.

PROBLEM 10.4. *Give the (partial) computation of the Turing machine M of Example 10.2 when the input tape is:*

1. $\underline{0}0$.
2. $1\underline{1}0$.

3. *The tape with all cells marked with* 1*s and cell* 5 *being scanned.*

PROBLEM 10.5. *For which possible input tapes does the partial computation of the Turing machine M of Example 10.2 eventually terminate?*

PROBLEM 10.6. *Find a Turing machine that (eventually!) fills a blank input tape with the pattern $xyyz3xyyz3xyyz3\dots$.*

PROBLEM 10.7. *Find a Turing machine with alphabet $\{\$, @\}$that never halts, no matter what is on the tape.*

EXAMPLE 10.4. The Turing machine P given below is intended to produce output $\underline{0}1^m$ on input $\underline{0}1^n01^m$ whenever $n, m > 0$.

P	0	1
1	$0R2$	
2		$1R3$
3	$0R4$	$0R3$
4	$0R4$	$0R5$
5	$0L8$	$1L6$
6	$0L6$	$1R7$
7	$1R4$	
8	$0L8$	$1L9$
9		$1L9$

Trace P's computation on, say, input $\underline{0}1^301^4$ to see how it works.

PROBLEM 10.8. *In each case, find a Turing machine (with the alphabet of your choice) that:*

1. *Halts with output $\underline{0}1^4$ on input $\underline{0}$.*
2. *Halts with output $01^n\underline{0}$ on input $\underline{0}0^n1$.*
3. *Halts with output $\underline{0}1^{2n}$ on input $\underline{0}1^n$.*
4. *Halts with output $\underline{0}(10)^n$ on input $\underline{0}1^n$.*
5. *Halts with output $\underline{0}1^m01^n$ on input $\underline{0}1^m0^k1^n$, if $n, m, k > 0$.*
6. *Halts with output $\underline{0}1^m01^n01^k$ on input $\underline{0}1^n01^k01^m$, if $n, m, k > 0$.*
7. *Halts with output $\underline{0}1^m01^n01^k01^m01^n01^k$ on input $\underline{0}1^m01^n01^k$, if $n, m, k > 0$.*
8. *On input $\underline{0}1^m01^n$, where $m, n > 0$, halts with output $\underline{0}1$ if $m \neq n$ and output $\underline{0}11$ if $m = n$.*

It is quite possible to find such machines with just $\{1\}$ as an alphabet.

NOTE. It doesn't matter what the machine you define in each case does on other inputs, so long as it does the right thing on the given one(s).

CHAPTER 11

Variations and Simulations

The definition of a Turing machine given in Chapter 10 is arbitrary in a number of ways, among them the use of an arbitrary finite alphabet, a single read-write scanner, and a single one-way infinite tape. One could restrict the definition we gave by allowing

- the machine to move the scanner only to one of left or right in each state,
- only $\{1\}$ as an alphabet,

or both, among various possibilities. One could also define apparently more powerful Turing machines by allowing the use of

- two-way infinite tapes,
- multiple tapes,
- two- and higher-dimensional tapes,

or various combinations of these, among many other possibilities. We will construct a number of Turing machines that simulate others with additional features; this will show that none of the modifications mentioned above really change what the machines can compute.

EXAMPLE 11.1. Consider the following Turing machine:

M	0	1
1	$1R2$	$0L1$
2	$0L2$	$1L1$

Note that in state 1, this machine may move the scanner to either the left or the right, depending on the contents of the cell being scanned. We will construct a Turing machine, with alphabet $\{1\}$, that emulates the action of M on any input, but which moves the scanner to only one of left or right in each state. There is no problem with state 2 of M, by the way, because in state 2 M always moves the scanner to the left.

The basic idea is to add some states to M which replace part of the description of state 1.

M'	0	1
1	$1R2$	$0R3$
2	$0L2$	$1L1$
3	$0L4$	$1L4$
4	$0L1$	

This machine is just like M except that in state 1 with input 1, instead of moving the scanner to the left and going to state 1, the machine moves the scanner to the right and goes to the new state 3. States 3 and 4 do nothing between them except move the scanner two cells to the left without changing the tape, thus putting it where M would have put it, and then entering state 1, as M would have.

PROBLEM 11.1. *Compare the computations of the machines M and M' of Example 11.1 on the input tapes*

1. 0
2. 011

and explain why is it not necessary to define M' for state 4 *on input* 1.

PROBLEM 11.2. *Given a Turing machine M with an arbitrary alphabet Σ, explain in detail how to construct a machine M' that simulates what M does on any input, but which moves the scanner only to one of left or right in each state.*

PROBLEM 11.3. *Given a Turing machine M with an arbitrary alphabet Σ, explain in detail how to construct a machine N with alphabet $\{1\}$ that simulates M.*

To define Turing machines with two-way infinite tapes we need only change the definition of the tape: instead of having tapes be sequences $\mathbf{a} = \langle a_0, a_1, a_2, \dots \rangle$ indexed by $\mathbb{N}$, we let them be sequences $\mathbf{b} = \langle \dots, b_{-2}, b_{-1}, b_0, b_1, b_2, \dots \rangle$ indexed by $\mathbb{Z}$. In defining computations for machines with two-way infinite tapes, we adopt the same conventions that we did for machines with one-way infinite tapes, such as the scanner starts off scanning cell 0 on the input tape. The only real difference is that a machine with a two-way infinite tape cannot halt by running off the left end of the tape ...

EXAMPLE 11.2. Consider the following two-way infinite tape Turing machine with alphabet $\{1\}$:

T	0	1
1	$1L1$	$0R2$
2	$0R2$	$1L1$

The biggest problem in trying to emulate T with a one-way infinite tape Turing machine O is representing a two-way infinite tape on a one-way infinite tape. To do this, we choose an alphabet for O with malice aforethought:

$$\left\{ \genfrac{}{}{0pt}{}{0}{\mathrm{S}}, \genfrac{}{}{0pt}{}{1}{\mathrm{S}}, \genfrac{}{}{0pt}{}{0}{0}, \genfrac{}{}{0pt}{}{0}{1}, \genfrac{}{}{0pt}{}{1}{0}, \genfrac{}{}{0pt}{}{1}{1} \right\}$$

We can now represent the tape $\mathbf{a}$,

$$\dots, a_{-2}, a_{-1}, a_0, a_1, a_2, \dots,$$

for T by the tape $\mathbf{a}'$,

$$\genfrac{}{}{0pt}{}{a_0}{\mathrm{S}}, \genfrac{}{}{0pt}{}{a_1}{a_{-1}}, \genfrac{}{}{0pt}{}{a_2}{a_{-2}}, \dots,$$

for O. In effect, this device allows us to split O's tape into two tracks, each of which accomodates half of the tape of T.

The key remaining idea is to split each state of T into a pair of states for O: one for the lower track and one for the upper track. One must take care to keep various details straight: when O changes a "cell" on one track, it should not change the corresponding "cell" on the other track; directions are reversed on the lower track; one has to "turn a corner" moving past cell 0; and so on.

O	0	$\genfrac{}{}{0pt}{}{0}{\mathrm{S}}$	$\genfrac{}{}{0pt}{}{0}{0}$	$\genfrac{}{}{0pt}{}{0}{1}$	$\genfrac{}{}{0pt}{}{1}{\mathrm{S}}$	$\genfrac{}{}{0pt}{}{1}{0}$	$\genfrac{}{}{0pt}{}{1}{1}$
1	$\genfrac{}{}{0pt}{}{1}{0}L1$	$\genfrac{}{}{0pt}{}{1}{\mathrm{S}}R3$	$\genfrac{}{}{0pt}{}{1}{0}L1$	$\genfrac{}{}{0pt}{}{1}{1}L1$	$\genfrac{}{}{0pt}{}{0}{\mathrm{S}}R2$	$\genfrac{}{}{0pt}{}{0}{0}R2$	$\genfrac{}{}{0pt}{}{0}{1}R2$
3	$\genfrac{}{}{0pt}{}{0}{1}R3$	$\genfrac{}{}{0pt}{}{1}{\mathrm{S}}R3$	$\genfrac{}{}{0pt}{}{0}{1}R3$	$\genfrac{}{}{0pt}{}{0}{0}L4$	$\genfrac{}{}{0pt}{}{0}{\mathrm{S}}R2$	$\genfrac{}{}{0pt}{}{1}{1}R3$	$\genfrac{}{}{0pt}{}{1}{0}L4$
2	$\genfrac{}{}{0pt}{}{0}{0}R2$	$\genfrac{}{}{0pt}{}{0}{\mathrm{S}}R2$	$\genfrac{}{}{0pt}{}{0}{0}R2$	$\genfrac{}{}{0pt}{}{0}{1}R2$	$\genfrac{}{}{0pt}{}{1}{\mathrm{S}}R3$	$\genfrac{}{}{0pt}{}{1}{0}L1$	$\genfrac{}{}{0pt}{}{1}{1}L1$
4	$\genfrac{}{}{0pt}{}{0}{0}L4$	$\genfrac{}{}{0pt}{}{0}{\mathrm{S}}R2$	$\genfrac{}{}{0pt}{}{0}{0}L4$	$\genfrac{}{}{0pt}{}{0}{1}R3$	$\genfrac{}{}{0pt}{}{1}{\mathrm{S}}R3$	$\genfrac{}{}{0pt}{}{1}{0}L4$	$\genfrac{}{}{0pt}{}{1}{1}R3$

States 1 and 3 are the upper- and lower-track versions, respectively, of T's state 1; states 2 and 4 are the upper- and lower-track versions, respectively, of T's state 2. We leave it to the reader to check that O actually does simulate T ...

PROBLEM 11.4. *Trace the (partial) computations of T, and their counterparts for O, for each of the following input tapes for T, shown with a bar over cell* 0*:*

1. $\overline{0}$ (i.e. *a blank tape)*
2. $1\overline{0}$
3. $\dots 111\overline{1}111 \dots$ (i.e. *every cell marked with* 1*)*

PROBLEM 11.5. *Explain in detail how, given a Turing machine N with an arbitrary alphabet and a two-way infinite tape, one can construct a Turing machine P with an one-way infinite tape that simulates N.*

PROBLEM 11.6. *Give a precise definition for Turing machines with two tapes. Explain how, given any such machine, one could construct a single-tape machine to simulate it.*

Problem 11.7. *Give a precise definition for Turing machines with two-dimensional tapes. Explain how, given any such machine, one could construct a single-tape machine to simulate it.*

Taken together, these results mean that for the purposes of what investigating what can be computed in principle, we can use any of the above variants on our definition of Turing machines without loss of generality.

CHAPTER 12

Universal Turing Machines and the Halting Problem

In Chapter 11 we devised techniques for constructing, given a particular Turing machine (of some type), a Turing machine (of another type) that would simulate it. We will go further and construct an *universal Turing machine* (sometimes referred to as an *UTM*): a machine U that, when given as input (a suitable description of) some Turing machine M and an input tape $\mathbf{a}$ for M, simulates the computation of M on input $\mathbf{a}$. In effect, an universal Turing machine is a piece of "hardware" that lets us treat Turing machines as "software".

As a bonus, constructing such a machine will give us the tools we will need to answer the following question:

THE HALTING PROBLEM. Given a Turing machine M and an input tape $\mathbf{a}$, is there an effective method to determine whether or not M eventually halts on input $\mathbf{a}$? □

An effective method to determine whether or not a given machine will eventually halt on a given input — short of waiting forever! — would be nice to have. For example, assuming Church's Thesis is true, such a method could let us identify computer programs which have infinite loops before they tie computers up in knots.

An Universal Turing Machine. The first problem in trying to build an universal Turing machine is finding a suitable way to describe the machine which is to be simulated, as well as its input tape, on the input tape of the universal Turing machine. We can simplify our task somewhat by restricting our attention to simulating Turing machines with one-way infinite tapes whose alphabet is just $\{1\}$. We lose no real generality in doing so since, by the results in Chapter 11, such machines can do just as much as any other type. Among the many possible ways of describing such machines as input, the one given below is fairly straightforward but woefully inefficient.[1] Essentially, it consists

[1] For an example of a different method, one could combine the methods developed in Chapter 15 of representing Turing machines and tapes by integers with the represention of integers on a tape used in Chapter 13.

of simply listing the table defining the Turing machine we wish to specify.

DEFINITION 12.1. Suppose M is a Turing machine with alphabet $\{1\}$ and m states. If $1 \leq k \leq m$ and $i \in \{0,1\}$ and $M(k,i) = (j,d,\ell)$, let

$$\llcorner M(k,i) \lrcorner = 1^{1+k} 0^{m-k} 0 1^{1+i} 0^{1-i} 0 1^{1+j} 0^{1-j} 0 1^{2+d} 0^{1-d} 0 1^{1+\ell} 0^{m-\ell};$$

if $M(k,i)$ is not defined, just let $j = d = \ell = 0$ above. The *representation* of M is then

$$\llcorner M \lrcorner = 0^3 1^{1+m} 0^3 \llcorner M(1,0) \lrcorner 0^2 \llcorner M(1,1) \lrcorner 0^2 \llcorner M(2,0) \lrcorner 0^2 \dots 0^2 \llcorner M(m,0) \lrcorner 0^2 \llcorner M(m,1) \lrcorner 0^3 .$$

That is, $\llcorner M(k,i) \lrcorner$ is a string of $2m+13$ 0s and 1s which represents the (k,i)th entry of M's table in five distinct blocks, separated by single 0s:

1. one of length $m+1$ codes k, represented by $k+1$ 1s followed by a padding of 0s;
2. one of length 2 codes i, represented by 11 if $i = 1$ and 10 if $i = 0$;
3. one of length 2 codes j, represented by 11 if $j = 1$ and 10 if $j = 0$;
4. one of length 3 codes d, represented by 111 if $d = 1$, 100 if $d = -1$, and 110 if $d = 0$ (*i.e.* if $M(k,i)$ is undefined);
5. one of length $m+1$ codes ℓ, represented by $\ell + 1$ 1s followed by a padding of 0s.

The representation of the machine M, $\llcorner M \lrcorner$, then consists of an initial string of three 0s, followed by $m+1$ 1s giving the number of states of M, three more 0s, the representations of the entries of the table of M — including the empty ones! — listed in order and separated by pairs of 0s, and a final string of three 0s.

EXAMPLE 12.1. The representation of the Turing machine

E	0	1
1	$0R2$	
2	$1L1$	$0R1$

,

is

$$0^3 1^3 0^3\ 1^2 001001001^3 01^3\ 0^2\ 1^2 001^2 01001^2 0010^2\ 0^2\ 1^3 01001^2 010^2 01^2 0\, 0^2\ 1^3 01^2 01001^3 01^2 0\, 0^3 .$$

PROBLEM 12.1. *Pick a Turing machine different from the one in Example 12.1 and give its representation.*

PROBLEM 12.2. *Give a table for the Turing machine whose representation is*

$$0001100011010011011101100110110100111011000 .$$

PROBLEM 12.3. *How many possible representations are there for a given Turing machine?*

PROBLEM 12.4. *Devise a more efficient way of representing Turing machines than that given in Definition 12.1.*

We now have at least one way of completely describing a given Turing machine M to another Turing machine. Of course, we have not yet considered how this description would actually be used, but we first need to find a way to describe M's input anyway. The most naive way to do this is to simply have the input for M follow the description of M. Unfortunately, this won't do because we will need to keep track which cell is being scanned scanned and what he current state is while simulating M. To solve this problem, we'll go whole hog and describe not just the tape $\mathbf{a}$ that M is reading but a complete tape position $(i, s, \mathbf{a})$. (Recall that i is an integer specifying the currently scanned cell, s is an integer specifying which state M is in, and $\mathbf{a}$ is the tape.)

DEFINITION 12.2. Suppose $(i, s, \mathbf{a})$ is the tape position of an m-state Turing machine with alphabet$\{1\}$. For each $\ell \in \mathbb{N}$, let

$$c_\ell = \begin{cases} 0 & \ell \neq i \text{ (}i.e.\text{ cell } \ell \text{ is not being scanned)} \\ 1 & \ell = i \text{ (}i.e.\text{ cell } \ell \text{ is the scanned cell).} \end{cases}$$

Let n be the least positive integer such that $a_k = 0$ for all $k > n$, if such exists. If so, the *representation* of $(i, s, \mathbf{a})$ is the finite sequence

$$\llcorner(i, s, \mathbf{a})\lrcorner = 0^3 1^{s+1} 0^{m-s} 0^3 1 c_0 a_0 1 c_1 a_1 1 c_2 a_2 \dots 1 c_n a_n 0^3 \,;$$

otherwise, the representation of $(i, s, \mathbf{a})$ is the infinite sequence

$$\llcorner(i, s, \mathbf{a})\lrcorner = 0^3 1^{s+1} 0^{m-s} 0^3 1 c_0 a_0 1 c_1 a_1 1 c_2 a_2 \dots \,.$$

That is, each cell of $\mathbf{a}$ is described by a triple of cells in the representation of $\mathbf{a}$. The first cell of this triple is simply a marker, one indicates whether or not the cell of $\mathbf{a}$ being described is the one being scanned by M, and the last gives the content of the cell being described. The representation of a tape position $(i, s, \mathbf{a})$ then consists of three 0s, a block of length $m + 1$ consisting of $s + 1$ 1s padded out by $m - s$ 0s, another three 0s, the triples coding the cells of $\mathbf{a}$, and, possibly, a final three 0s to mark the end of the code. Since we assume tapes to have all but finitely many cells blank unless stated otherwise, almost all of the representations of tape positions we will encounter will be finite.

EXAMPLE 12.2. Consider the tape position $(2, 3, \mathbf{a})$ for a 5-state Turing machine, where $\mathbf{a} = 01101$. The representation of this tape position is

$$000\,111100\,000\,100\,101\,111\,100\,101\,000\,.$$

Note that the only use of the 1 at the beginning of the representation of each cell is to mark the fact that a cell is being represented: moving a scanner along the representation of $\mathbf{a}$, one can identify the beginning and end (if any) of the representation because one encounters a 0 instead of a 1 marking the representation of another cell.

PROBLEM 12.5. *What are the representations of the following tape positions for a 5-state Turing machine?*

1. $(0, 4, 11111)$
2. $(3, 1, 101011$

PROBLEM 12.6. *What tape positions are represented by the following?*

1. 110000011010010010110110010010110000
2. 111110001011111011011011011011 ...

PROBLEM 12.7. *Devise a more efficient way of representing tape positions than that given in Definition 12.2.*

We can now define our representation for M together with a tape position $(i, s, \mathbf{a})$. Except for inserting some extra 0s, this just amounts to $\ulcorner M \urcorner$ followed by $\ulcorner (i, s, \mathbf{a}) \urcorner$.

DEFINITION 12.3. Suppose M is an m-state Turing machine with alphabet $\{1\}$ and $(i, s, \mathbf{a})$ is a tape position for M (so $1 \leq s \leq m$). Then the representation of the machine M together with the tape position $(i, s, \mathbf{a})$ is the sequence

$$\ulcorner M \urcorner 0^{3m+33} \ulcorner (i, s, \mathbf{a}) \urcorner .$$

The $3m+33$ 0s interpolated in the middle are intended to be used for "scratch space" in the simulation of M by the universal Turing machine we will construct in the following series of problems. Depending on how these are solved, it may be possible to reduce — or necessary to increase! — that $3m + 33$.

NOTE. The statements of Problems 12.8–12.13 below assume the use of the representation schemes given in Definitions 12.1–12.3. If you would rather use the methods you devised in Problems 12.4 and 12.7, by all means adapt the statements of Problems 12.8–12.13 accordingly. You may find it convenient to give the machines you build in these problems alphabets other than $\{1\}$, but one need not do so. (Why doesn't it really matter if one does?)

PROBLEM 12.8. *Suppose M is an m-state Turing machine with alphabet $\{1\}$, $(i, s, \mathbf{a})$ is a tape position for M, and $d = \pm 1$. Find a*

Turing machine **H** *which on input*

$$\underline{0}\llcorner M\lrcorner 0^{3m+30}1^{2+d}0^{1-d}\llcorner(i,s,\mathbf{a})\lrcorner$$

halts with output

$$\underline{0}\llcorner M\lrcorner 0^{3m+30}1^{2+d}0^{1-d}\llcorner(i+d,s,\mathbf{a})\lrcorner\,.$$

H *should extend the representation of* **a** *if necessary.*

PROBLEM 12.9. *Suppose M is an m-state Turing machine with alphabet $\{1\}$, $(i,s,\mathbf{a})$ is a tape position for M. Find a Turing machine* **R** *which on input*

$$\underline{0}\llcorner M\lrcorner 0^{3m+33}\llcorner(i,s,\mathbf{a})\lrcorner$$

halts with output

$$\underline{0}\llcorner M\lrcorner 0^{3m+31}1^{1+c}0^{1-c}\llcorner(i,s,\mathbf{a})\lrcorner\,,$$

where $c=a_i$.

PROBLEM 12.10. *Suppose M is an m-state Turing machine with alphabet $\{1\}$, $(i,s,\mathbf{a})$ is a tape position for M, and $j\in\{0,1\}$. Find a Turing machine* **W** *which on input*

$$\underline{0}\llcorner M\lrcorner 0^{3m+31}1^{1+j}0^{1-j}\llcorner(i,s,\mathbf{a})\lrcorner$$

halts with output

$$\underline{0}\llcorner M\lrcorner 0^{3m+31}1^{1+j}0^{1-j}\llcorner(i,s,\mathbf{a}')\lrcorner\,,$$

where $\mathbf{a}'$ is identical to **a** *except that $a'_i=j$.*

PROBLEM 12.11. *Suppose M is an m-state Turing machine with alphabet $\{1\}$, $(i,s,\mathbf{a})$ is a tape position for M, and $c\in\{0,1\}$. Find a Turing machine* **E** *which on input*

$$\underline{0}\llcorner M\lrcorner 0^{3m+31}1^{1+c}0^{1-c}\llcorner(i,s,\mathbf{a})\lrcorner$$

halts with output

$$\underline{0}\llcorner M\lrcorner\llcorner M(s,c)\lrcorner 0^{m+18}1^{1+c}0^{1-c}\llcorner(i,s,\mathbf{a})\lrcorner\,.$$

PROBLEM 12.12. *Suppose M is an m-state Turing machine with alphabet $\{1\}$ and $(i,s,\mathbf{a})$ is a tape position for M. Find a Turing machine* **S** *which on input*

$$\underline{0}\llcorner M\lrcorner 0^{3m+33}\llcorner(i,s,\mathbf{a})\lrcorner$$

halts with output

$$\underline{0}\llcorner M\lrcorner 0^{3m+33}\llcorner\mathbf{M}(i,s,\mathbf{a})\lrcorner\,.$$

Using these machines, we can finally assemble an universal Turing machine.

THEOREM 12.13. *There is a Turing machine* $\mathbf{U}$ *such that, for any* $m \geq 1$, m*-state Turing machine* M *with alphabet* $\{1\}$, *and tape position* $(i, s, \mathbf{a})$ *for* M, $\mathbf{U}$ *acts on the input position*

$$\underline{0}\llcorner M \lrcorner 0^{3m+33} \llcorner (i, s, \mathbf{a}) \lrcorner$$

as follows:

- *If* M, *starting from position* $(i, s, \mathbf{a})$, *eventually halts in position* $(j, t, \mathbf{b})$, *then* $\mathbf{U}$ *eventually halts in the position*

$$\underline{0}\llcorner M \lrcorner 0^{3m+33} \llcorner (j, t, \mathbf{b}) \lrcorner .$$

- *If* M, *starting from the initial tape position* $(i, s, \mathbf{a})$, *eventually runs off the left end of its tape,* $\mathbf{U}$ *eventually runs off the left end of its own tape.*
- *If* M, *starting from the initial tape position* $(i, s, \mathbf{a})$, *never halts, then* $\mathbf{U}$ *never halts.*

The Halting Problem. Given that we are using Turing machines to formalize the notion of an effective method, one of the difficulties with solving the Halting Problem is representing a given Turing machine and its input tape as input for another machine. As this is exactly what was done above, we can now formulate a precise version of the Halting Problem and solve it.

THE HALTING PROBLEM. Is there a Turing machine $\mathbf{T}$ which, for any $m \geq 1$, m-state Turing machine M with alphabet $\{1\}$, and tape $\mathbf{a}$ for M, halts on input

$$\underline{0}\llcorner M \lrcorner 0^{3m+33} \llcorner (0, 1, \mathbf{a}) \lrcorner$$

with output $\underline{0}11$ if M halts on input $\mathbf{a}$, and with output $\underline{0}1$ if M does not halt on input $\mathbf{a}$? □

Note that this version of the Halting Problem is equivalent to our original one only if Church's Thesis is true.

PROBLEM 12.14. *Find a Turing machine* $\mathbf{C}$ *which, for any Turing machine* M, *on input*

$$\underline{0}\llcorner M \lrcorner$$

eventually halts with output

$$\underline{0}\llcorner (0, 1, \underline{0}\llcorner M \lrcorner) \lrcorner .$$

THEOREM 12.15. *There is no Turing machine* $\mathbf{T}$ *which, for any* $m \geq 1$, m*-state Turing machine* M, *and tape* $\mathbf{a}$ *for* M, *halts on input*

$$\underline{0}\llcorner M \lrcorner 0^{3m+33} \llcorner (0, 1, \mathbf{a}) \lrcorner$$

with output $\underline{0}11$ *if* M *halts on input* $\mathbf{a}$*, and with output* $\underline{0}1$ *if* M *does not halt on input* $\mathbf{a}$*.*

CHAPTER 13

Computable and Non-Computable Functions

So far, the only substantial facts we have about what Turing machines can do is that they can be used to simulate other Turing machines, but cannot solve the Halting Problem. Neither fact is trivial, but neither is really interesting unless Turing machines can also be used to handle more natural computational problems. Arithmetic is a common source of such problems in the real world; indeed, any notion of computation that can't handle it is unlikely to be of great use.

Notation and conventions. To keep things as simple as possible, we will stick to computations involving the *natural numbers*, *i.e.* the non-negative integers, the set of which is usually denoted by $\mathbb{N} = \{0, 1, 2, \dots\}$.. The set of all k-tuples $(n_1, \dots, n_k)$ of natural numbers is denoted by $\mathbb{N}^k$. For all practical purposes, we may take $\mathbb{N}^1$ to be $\mathbb{N}$ by identifying the 1-tuple (n) with the natural number n.

For $k \geq 1$, f is a *k-place function* (from the natural numbers to the natural numbers), often written as $f \colon \mathbb{N}^k \to \mathbb{N}$, if it associates a value, $f(n_1, \dots, n_k)$, to each k-tuple $(n_1, n_2, \dots, n_k) \in \mathbb{N}^k$. Strictly speaking, though we will frequently forget to be explicit about it, we will often be working with k-place *partial functions* which might not be defined for all the k-tuples in $\mathbb{N}^k$. If f is a k-place partial function, the *domain* of f is the set

$$\mathrm{dom}(f) = \{ (n_1, \dots, n_k) \in \mathbb{N}^k \mid f(n_1, \dots, n_k) \text{ is defined} \}.$$

Similarly, the *image* of f is the set

$$\mathrm{im}(f) = \{ f(n_1, \dots, n_k) \mid (n_1, \dots, n_k) \in \mathrm{dom}(f) \}.$$

In subsequent chapters we will also work with relations on the natural numbers. Recall that a *k-place relation* on $\mathbb{N}$ is formally a subset P of $\mathbb{N}^k$; $P(n_1, \dots, n_k)$ is *true* if $(n_1, \dots, n_k) \in P$ and *false* otherwise. In particular, a 1-place relation is really just a subset of $\mathbb{N}$.

Relations and functions are closely realted. A k-place function f could be identified with the $k+1$-place relation P_f given by

$$P_f(n_1, \dots, n_k, n_{k+1}) \iff f(n_1, \dots, n_k) = n_k + 1.$$

In the other direction, a k-place relation P could be identified with its *characteristic function* :

$$\chi_P(n_1, \dots, n_k) = \begin{cases} 1 & \text{if } P(n_1, \dots, n_k) \text{ is true;} \\ 0 & \text{if } P(n_1, \dots, n_k) \text{ is false.} \end{cases}$$

The basic convention for representing natural numbers on the tape of a Turing machine is a slight variation of *unary notation* : n is represented by 1^{n+1}. (Why would using 1^n be a bad idea?) A k-tuple $(n_1, n_2, \dots, n_k) \in \mathbb{N}$ will be represented by $1^{n_1+1}01^{n_2+1}0 \dots 01^{n_k+1}$, *i.e.* with the representations of the individual numbers separated by 0s. This scheme is inefficient in its use of space — compared to binary notation, for example — but it is simple and can be implemented on Turing machines restricted to the alphabet $\{1\}$.

Computable functions. With suitable conventions for representing the input and output of a function on the natural numbers on the tape of a Turing machine in hand, we can define what it means for a function to be computable by a Turing machine.

DEFINITION 13.1. A k-place function f is *Turing computable*, or just *computable*, if there is a Turing machine M such that for any k-tuple $(n_1, \dots, n_k) \in \text{dom}(f)$ the computation of M with input tape $\underline{0}1^{n_1+1}01^{n_2+1} \dots 01^{n_k+1}$ eventually halts with output tape $\underline{0}1^{f(n_1,\dots,n_k)+1}$. Such an M is said to *compute* f.

Note that for a Turing machine M to compute a function f, M need only do the right thing on the right kind of input: what M does in other situations does not matter. In particular, it does not matter what M might do with k-tuple which is not in the domain of f.

EXAMPLE 13.1. The identity function $i_{\mathbb{N}}\colon \mathbb{N} \to \mathbb{N}$, *i.e.* $i_{\mathbb{N}}(n) = n$, is computable. It is computed by $M = \emptyset$, the Turing machine with an empty table that does absolutely nothing on any input.

EXAMPLE 13.2. The projection function $\pi_1^2 : \mathbb{N}^2 \to \mathbb{N}$ given by $\pi_1^2(n, m) = n$ is computed by the Turing machine:

P_1^2	0	1
1	$0R2$	
2	$0R3$	$1R2$
3	$0L4$	$0R3$
4	$0L4$	$1L5$
5		$1L5$

P_1^2 acts as follows: it moves to the right past the first block of 1s without disturbing it, erases the second block of 1s, and then returns to the left of first block and halts.

The projection function $\pi_2^2 : \mathbb{N}^2 \to \mathbb{N}$ given by $\pi_1^2(n, m) = m$ is also computable: the Turing machine P of Example 10.4 does the job.

PROBLEM 13.1. *Find Turing machines that compute the following functions and explain how they work.*

1. $\mathrm{O}(n) = 0$.
2. $\mathrm{S}(n) = n + 1$.
3. $\mathrm{SUM}(n, m) = n + m$.
4. $\mathrm{PRED}(n) = \begin{cases} n-1 & n \geq 1 \\ 0 & n = 0 \end{cases}$.
5. $\mathrm{DIFF}(n, m) = \begin{cases} n-m & n \geq m \\ 0 & n < m \end{cases}$.
6. $\pi_2^3(p, q, r) = q$.

We will see how to build complex functions computable by Turing machines out of simpler ones in the next chapter.

A non-computable function. In the meantime, it is worth asking whether or not every function on the natural numbers is computable. No such luck!

PROBLEM 13.2. *Show that there is some* 1*-place function* f *which is not computable by comparing the number of such functions to the number of Turing machines.*

The argument hinted at above is unsatisfying in that it tells us there is a non-computable function without actually producing an explicit example. We can have some fun on the way to one.

DEFINITION 13.2 (Busy Beaver Competition). A machine M is an *n-state entry in the busy beaver competition* if:

- M has a two-way infinite tape and alphabet $\{1\}$;
- M has $n + 1$ states, but state $n + 1$ is used only for halting (so both $M(n + 1, 0)$ and $M(n + 1, 1)$ are undefined);
- M eventually halts when given a blank input tape.

M's *score* in the competition is the number of 1's on the output tape of its computation from a blank input tape. The greatest possible score of an n-state entry in the competition is denoted by $\Sigma(n)$.

Note that there are only finitely many possible n-state entries in the busy beaver competition because there are only finitely many $n + 1$-state Turing machines with alphabet $\{1\}$. Since there is at least one

n-state entry in the busy beaver competition for every $n \geq 0$, it follows that $\Sigma(n)$ is well-defined for each $n \in \mathbb{N}$.

EXAMPLE 13.3. $M = \emptyset$ is the *only* 0-state entry in the busy beaver competition, so $\Sigma(0) = 0$.

EXAMPLE 13.4. The machine P given by

P	0	1
1	$1R2$	$1L2$
2	$1L1$	$1L3$

is a 2-state entry in the busy beaver competition with a score of 4, so $\Sigma(2) \geq 4$.

The function Σ grows extremely quickly. It is known that $\Sigma(0) = 0$, $\Sigma(1) = 1$, $\Sigma(2) = 4$, $\Sigma(3) = 6$, and $\Sigma(4) = 13$. The value of $\Sigma(5)$ is still unknown, but must be quite large.[1]

PROBLEM 13.3. *Show that:*

1. *The* 2*-state entry given in Example 13.4 actually scores* 4.
2. $\Sigma(1) = 1$.
3. $\Sigma(3) \geq 6$.
4. $\Sigma(n) < \Sigma(n+1)$ *for every* $n \in \mathbb{N}$.

PROBLEM 13.4. *Devise as high-scoring* 4*- and* 5*-state entries in the busy beaver competition as you can.*

The serious point of the busy beaver competition is that Σ is *not* a Turing computable function.

PROPOSITION 13.5. Σ *is not computable by any Turing machine.*

Anyone interested in learning more about the busy beaver competition should start by reading the paper [**10**] in which it was first introduced.

[1]The best score known to the author as of this writing by a 5-state entry in the busy beaver competition is 4098. One of the two machines achieving this score does so in a computation that takes over 40 million steps! The other requires only 11 million or so ...

CHAPTER 14

Primitive Recursive Functions

Starting with a small set of computable functions, and applying computable ways of building functions from simpler ones, we will build up a useful collection of computable functions. This will also go a long way toward giving us a characterization of computable functions which does not mention any particular computing devices.

The initial functions. The set of computable functions that will be the fundamental building blocks for all that follows is infinite only because of the presence of all the projection functions.

DEFINITION 14.1. The following are the *initial functions*:

- O, the 1-place function such that $\mathrm{O}(n) = 0$ for all $n \in \mathbb{N}$;
- S, the 1-place function such that $\mathrm{S}(n) = n+1$ for all $n \in \mathbb{N}$; and,
- for each $k \geq 1$ and $1 \leq i \leq k$, π_i^k, the k-place function such that $\pi_i^k(n_1, \dots, n_k) = n_i$ for all $(n_1, \dots, n_k) \in \mathbb{N}^k$.

O is often referred to as the *zero function*, S is the *successor function*, and the functions π_ℓ^k are called the *projection functions*.

Note that π_1^1 is just the identity function on $\mathbb{N}$. We have already observed that O, S, π_1^1, π_1^2, π_2^2, and π_2^3 are Turing computable in Chapter 13.

PROBLEM 14.1. *Show that all of the initial functions are Turing computable.*

Composition. The first of our methods for assembling computable functions from simpler ones should be thoroughly familiar from many parts of mathematics.

DEFINITION 14.2. Suppose that $m, k \geq 1$, g is an m-place function, and $h_1, \dots, h_m$ are k-place functions. Then the k-place function f is said to be obtained from $g, h_1, \dots, h_m$ by *composition*, written as

$$f = g \circ (h_1, \dots, h_m),$$

if for all $(n_1, \dots, n_k) \in \mathbb{N}^k$,

$$f(n_1, \dots, n_k) = g(h_1(n_1, \dots, n_k), \dots, h_m(n_1, \dots, n_k)).$$

EXAMPLE 14.1. The constant function c_1^1, where $c_1^1(n) = 1$ for all n, can be obtained by composition from the initial functions S and O. For any $n \in \mathbb{N}$,

$$c_1^1(n) = (\mathrm{S} \circ \mathrm{O})(n) = \mathrm{S}(\mathrm{O}(n)) = \mathrm{S}(0) = 0 + 1 = 1\,.$$

PROBLEM 14.2. *Suppose $k \geq 1$ and $a \in \mathbb{N}$. Use composition to define the constant function c_a^k, where $c_a^k(n_1, \dots, n_k) = a$ for all $(n_1, \dots, n_k) \in \mathbb{N}^k$, from the initial functions.*

PROPOSITION 14.3. *Suppose that $1 \leq k$, $1 \leq m$, g is a Turing computable m-place function, and $h_1, \dots, h_m$ are Turing computable k-place functions. Then $g \circ (h_1, \dots, h_m)$ is also Turing computable.*

Unfortunately, one can't do much else of interest using just the initial functions and composition ...

PROPOSITION 14.4. *Suppose f is a 1-place function obtained from the initial functions by finitely many applications of composition. Then there is a constant $c \in \mathbb{N}$ such that $f(n) \leq n + c$ for all $n \in \mathbb{N}$.*

Primitive recursion. Primitive recursion boils down to defining a function inductively, using different functions to tell us what to do at the base and inductive steps. Together with composition, it suffices to build up just about all familiar arithmetic functions from the initial functions.

DEFINITION 14.3. Suppose that $k \geq 1$, g is a k-place function, and h is a $k+2$-place function. Let f be the $k+1$-place function such that

1. $f(n_1, \dots, n_k, 0) = g(n_1, \dots, n_k)$ and
2. $f(n_1, \dots, n_k, m+1) = h(n_1, \dots, n_k, m, f(n_1, \dots, n_k, m)$

for every $(n_1, \dots, n_k) \in \mathbb{N}^k$ and $m \in \mathbb{N}$. Then f is said to be obtained from g and h by *primitive recursion*.

That is, the initial values of f are given by g, and the rest are given by h operating on the given input *and* the preceding value of f.

For a start, primitive recursion and composition let us define addition and multiplication from the initial functions.

EXAMPLE 14.2. $\textsc{Sum}(n, m) = n + m$ is obtained by primitive recursion from the initial function π_1^1 and the composition $\mathrm{S} \circ \pi_3^3$ of initial functions as follows:

- $\textsc{Sum}(n, 0) = \pi_1^1(n)$;
- $\textsc{Sum}(n, m+1) = (\mathrm{S} \circ \pi_3^3)(n, m, \textsc{Sum}(n, m))$.

To see that this works, one can proceed by induction on m:

At the base step, $m = 0$, we have

$$\textsc{Sum}(n, 0) = \pi_1^1(n) = n = n + 0\,.$$

Assume that $m \geq 0$ and $\textsc{Sum}(n, m) = n + m$. Then

$$\begin{aligned}\textsc{Sum}(n, m+1) &= (\mathrm{S} \circ \pi_3^3)(n, m, \textsc{Sum}(n, m)) \\ &= \mathrm{S}(\pi_3^3(n, m, \textsc{Sum}(n, m))) \\ &= \mathrm{S}(\textsc{Sum}(n, m)) \\ &= \textsc{Sum}(n, m) + 1 \\ &= n + m + 1\,,\end{aligned}$$

as desired.

As addition is to the successor function, so multiplication is to addition.

Example 14.3. $\textsc{Mult}(n, m) = nm$ is obtained by primitive recursion from O and $\textsc{Sum} \circ (\pi_3^3, \pi_1^3)$:

- $\textsc{Mult}(n, 0) = \mathrm{O}(n)$;
- $\textsc{Mult}(n, m+1) = (\textsc{Sum} \circ (\pi_3^3, \pi_1^3))(n, m, \textsc{Mult}(n, m))$.

We leave it to the reader to check that this works.

Problem 14.5. *Use composition and primitive recursion to obtain each of the following functions from the initial functions or other functions already obtained from the initial functions.*

1. $\textsc{Exp}(n, m) = n^m$
2. $\textsc{Pred}(n)$ *(defined in Problem 13.1)*
3. $\textsc{Diff}(n, m)$ *(defined in Problem 13.1)*
4. $\textsc{Fact}(n) = n!$

Proposition 14.6. *Suppose $k \geq 1$, g is a Turing computable k-place function, and h is a Turing computable $k+2$-place function. If f is obtained from g and h by primitive recursion, then f is also Turing computable.*

Primitive recursive functions and relations. The collection of functions which can be obtained from the initial functions by (possibly repeatedly) using composition and primitive recursion is useful enough to have a name.

Definition 14.4. A function f is *primitive recursive* if it can be defined from the initial functions by finitely many applications of the operations of composition and primitive recursion.

So we already know that all the initial functions, addition, and multiplication, among others, are primitive recursive.

Problem 14.7. *Show that each of the following functions is primitive recursive.*

1. *For any* $k \geq 0$ *and primitive resursive* $k+1$*-place function* g*, and* $\Pi_{i=0}^{m} g(n_1, \dots, n_k, i)$*, the* $k+1$*-place function* f *given by*
$$\begin{aligned} f(n_1, \dots, n_k, m) &= \Pi_{i=0}^{m} g(n_1, \dots, n_k, i) \\ &= g(n_1, \dots, n_k, 0) \cdot \ldots \cdot g(n_1, \dots, n_k, m) \end{aligned}$$
for any $k+1$*-tuple* $(n_1, \dots, n_k, m)$.
2. *For any constant* $a \in \mathbb{N}$, $\chi_{\{a\}}(n) = \begin{cases} 0 & n \neq a \\ 1 & n = a \,. \end{cases}$
3. $h(n_1, \dots, n_k) = \begin{cases} f(n_1, \dots, n_k) & (n_1, \dots, n_k) \neq (c_1, \dots, c_k) \\ a & (n_1, \dots, n_k) = (c_1, \dots, c_k) \end{cases}$, *if* f *is a primitive recursive* k*-place function and* $a, c_1, \dots, c_k \in \mathbb{N}$ *are constants.*

Theorem 14.8. *Every primitive recursive function is Turing computable.*

Be warned, however, that there are computable functions which are not primitive recursive.

We can extend the idea of "primitive recursive" to relations by using their characteristic functions.

Definition 14.5. Suppose $k \geq 1$. A k-place relation $P \subseteq \mathbb{N}^k$ is *primitive recursive* if its characteristic function

$$\chi_P(n_1, \dots, n_k) = \begin{cases} 1 & (n_1, \dots, n_k) \in P \\ 0 & (n_1, \dots, n_k) \notin P \end{cases}$$

is primitive recursive.

Example 14.4. $P = \{2\} \subset \mathbb{N}$ is primitive recursive since $\chi_{\{2\}}$ is recursive by Problem 14.7.

Problem 14.9. *Show that the following relations and functions are primitive recursive.*

1. $\neg P$, i.e. $\mathbb{N}^k \setminus P$, *if* P *is a primitive recursive* k*-place relation.*
2. $P \vee Q$, i.e. $P \cup Q$, *if* P *and* Q *are primitive recursive* k*-place relations.*
3. $P \wedge Q$, i.e. $P \cap Q$, *if* P *and* Q *are primitive recursive* k*-place relations.*
4. Equal, *where* Equal$(n, m) \iff n = m$.
5. $h(n_1, \dots, n_k, m) = \sum_{i=0}^{m} g(n_1, \dots, n_k, i)$, *for any* $k \geq 0$ *and primitive recursive* $k+1$*-place function* g.

6. Div, *where* $\text{Div}(n,m) \iff n \mid m$.
7. IsPrime, *where* $\text{IsPrime}(n) \iff n$ *is prime.*
8. $\text{Prime}(k) = p_k$, *where* $p_0 = 1$ *and* p_k *is the* k*th prime if* $k \geq 1$.
9. $\text{Power}(n,m) = k$, *where* $k \geq 0$ *is maximal such that* $n^k \mid m$.
10. $\text{Length}(n) = \ell$, *where* ℓ *is maximal such that* $p_\ell \mid n$.
11. $\text{Element}(n,i) = n_i$, *if* $n = p_1^{n_1} \dots p_k^{n_k}$ *(and* $n_i = 0$ *if* $i > k$*).*
12. $\text{Subseq}(n,i,j) = \begin{cases} p_i^{n_i} p_{i+1}^{n_{i+1}} \dots p_j^{n_j} & \text{if } 1 \leq i \leq j \leq k \\ 0 & \text{otherwise} \end{cases}$, *whenever* $n = p_1^{n_1} \dots p_k^{n_k}$.
13. $\text{Concat}(n,m) = p_1^{n_1} \dots p_k^{n_k} p_{k+1}^{m_1} \dots p_{k+\ell}^{m_l}$, *if* $n = p_1^{n_1} \dots p_k^{n_k}$ *and* $m = p_1^{m_1} \dots p_\ell^{m_\ell}$.

Parts of Problem 14.9 give us tools for representing finite sequences of integers by single integers, as well as some tools for manipulating these representations. This lets us reduce, in principle, all problems involving primitive recursive functions and relations to problems involving only 1-place primitive recursive functions and relations.

Theorem 14.10. *A* k*-place* g *is primitive recursive if and only if the* 1*-place function* h *given by* $h(n) = g(n_1, \dots, n_k)$ *if* $n = p_1^{n_1} \dots p_k^{n_k}$ *is primitive recursive.*

Note. It doesn't matter what the function h does on ns which do not represent a sequence of length k.

Corollary 14.11. *A* k*-place relation* P *is primitive recursive if and only if the* 1*-place relation* P' *is primitive recursive, where*

$$(n_1, \dots, n_k) \in P \iff p_1^{n_1} \dots p_k^{n_k} \in P'.$$

Computable non-primitive recursive functions. While primitive recursion and composition do not suffice to build up all Turing computable functions from the initial functions, they are powerful enough that specific counterexamples are not all that easy to find.

Example 14.5 (Ackerman's Function). Define the 2-place function A from as follows:

- $\mathrm{A}(0,\ell) = \mathrm{S}(\ell)$
- $\mathrm{A}(\mathrm{S}(k),0) = \mathrm{A}(k,1)$
- $\mathrm{A}(\mathrm{S}(k),\mathrm{S}(\ell)) = \mathrm{A}(k,\mathrm{A}(\mathrm{S}(k),\ell))$

Given A, define the 1-place function α by $\alpha(n) = \mathrm{A}(n,n)$.

It isn't too hard to show that A, and hence also α, are Turing computable. However, though it takes considerable effort to prove it, α grows faster with n than any primitive recursive function. (Try working out the first few values of α …)

PROBLEM 14.12. *Show that the functions* A *and* α *defined in Example 14.5 are Turing computable.*

If you are very ambitious, you can try to prove the following theorem.

THEOREM 14.13. *Suppose* α *is the function defined in Example 14.5 and* f *is any primitive recursive function. Then there is an* $n \in \mathbb{N}$ *such that for all* $k > n$, $\alpha(k) > f(k)$.

COROLLARY 14.14. *The function* α *defined in Example 14.5 is not primitive recursive.*

... but if you aren't, you can still try the following exercise.

PROBLEM 14.15. *Informally, define a computable function which must be different from every primitive recursive function.*

CHAPTER 15

Recursive Functions

We add one more computable method of building functions, unbounded minimalization, to our repertoire. The functions which can be defined from the initial functions using unbounded minimalization, as well as composition and primitive recursion, turn out to be precisely the Turing computable functions.

Unbounded minimalization. Unbounded minimalization is the counterpart for functions of "brute force" algorithms that try every possibility until they succeed. (Which, of course, they might not ...)

DEFINITION 15.1. Suppose $k \geq 1$ and g is a $k+1$-place function. Then the *unbounded minimalization* of g is the k-place function f defined by

$$f(n_1, \dots, n_k) = m \text{ where } m \text{ is least so that } g(n_1, \dots, n_k, m) = 0.$$

This is often written as $f(n_1, \dots, n_k) = \mu m[g(n_1, \dots, n_k, m) = 0]$.

NOTE. If there is no m such that $g(n_1, \dots, n_k, m) = 0$, then the unbounded minimalization of g is not defined on $(n_1, \dots, n_k)$. This is one reason we will occasionally need to deal with partial functions.

If the unbounded minimalization of a computable function is to be computable, we have a problem even if we ask for some default output (0, say) to ensure that it is defined for all k-tuples. The obvious procedure which tests successive values of g to find the needed m will run forever if there is no such m, and the incomputability of the Halting Problem suggests that other procedure's won't necessarily succeed either. It follows that it is desirable to be careful, so far as possible, which functions unbounded minimalization is applied to.

DEFINITION 15.2. A $k+1$-place function g is said to be *regular* if for every $(n_1, \dots, n_k) \in \mathbb{N}^k$, there is at least one $m \in \mathbb{N}$ so that $g(n_1, \dots, n_k, m) = 0$.

That is, g is regular precisely if the obvious strategy of computing $g(n_1, \dots, n_k, m)$ for $m = 0, 1, \dots$ in succession until an m is found with $g(n_1, \dots, n_k, m) = 0$ always succeeds.

PROPOSITION 15.1. *If g is a Turing computable regular $k+1$-place function, then the unbounded minimalization of g is also Turing computable.*

While unbounded minimalization adds something essentially new to our repertoire, it is worth noticing that *bounded minimalization* does not.

PROBLEM 15.2. *Suppose g is a $k+1$-place primitive recursive regular function such that for some primitive recursive k-place function h,*

$$\mu m[g(n_1, \dots, n_k, m) = 0] \leq h(n_1, \dots, n_k)$$

for all $(n_1, \dots, n_k) \in \mathbb{N}$. Show that $\mu m[g(n_1, \dots, n_k, m) = 0]$ is also primitive recursive.

Recursive functions and relations. We can finally define an equivalent notion of computability for functions on the natural numbers which makes no mention of any computational device.

DEFINITION 15.3. A k-place function f is *recursive* if it can be defined from the initial functions by finitely many applications of composition, primitive recursion, and the unbounded minimalization of regular functions.

Similarly, k-place partial function is *recursive* if it can be defined from the initial functions by finitely many applications of composition, primitive recursion, and the unbounded minimalization of (possibly non-regular) functions.

In particular, every primitive recursive function is a recursive function.

THEOREM 15.3. *Every recursive function is Turing computable.*

We shall show that every Turing computable function is recursive later on. Similarly to primitive recursive relations we have the following.

DEFINITION 15.4. A k-place relation P is said to be *recursive* (*Turing computable*) if its characteristic function χ_P is recursive (Turing computable).

Since every recursive function is Turing computable, and *vice versa*, "recursive" is just a synonym of "Turing computable", for functions and relations alike.

Also, similarly to Theorem 14.10 and Corollary 14.11 we have the following.

THEOREM 15.4. *A k-place g is recursive if and only if the* 1*-place function h given by $h(n) = g(n_1, \ldots, n_k)$ if $n = p_1^{n_1} \ldots p_k^{n_k}$ is recursive.*

As before, it doesn't really matter what the function h does on an n which does not represent a sequence of length k.

COROLLARY 15.5. *A k-place relation P is recursive if and only if the* 1*-place relation P' is recursive, where*

$$(n_1, \ldots, n_k) \in P \iff p_1^{n_1} \ldots p_k^{n_k} \in P' .$$

Turing computable functions are recursive. By putting some of the ideas in Chapters 12 and 14 together, we can use recursive functions to simulate Turing machines. This will show that Turing computable functions are recursive and, as a bonus, give us another way of constructing an universal Turing machine. Since recursive functions operate on integers, we first need to specify some way to code the tapes of Turing machines by integers. We'll try keep close to the representation schemes given in Definitions 12.1 and 12.2 in the process. As we did in those definitions, we shall stick to Turing machines with alphabet $\{1\}$ for simplicity.

DEFINITION 15.5. Suppose $(i, s, \mathbf{a})$ is a tape position such that all but finitely many cells of $\mathbf{a}$ are blank. Let n be any positive integer such that $a_k = 0$ for all $k \in \mathbb{N}$ with $k > n$. Then the *code* of $(i, s, \mathbf{a})$ is

$$\ulcorner (i, s, \mathbf{a}) \urcorner = 2^i 3^s 5^{a_0} 7^{a_1} 11^{a_2} \ldots p_{n+3}^{a_n} .$$

EXAMPLE 15.1. Consider the tape position $(1, 2, 1001)$. Then

$$\ulcorner (1, 2, 1001) \urcorner = 2^1 3^2 5^1 7^0 11^0 13^1 = 1170 .$$

PROBLEM 15.6. *Find the codes of the following tape positions.*

1. $(0, 1, \mathbf{a})$*, where* $\mathbf{a}$ *is entirely blank.*
2. $(3, 4, \mathbf{a})$*, where* $\mathbf{a}$ *is* 1011100101*.*

PROBLEM 15.7. *What is the tape position whose code is* 10314720*?*

We'll also need to code sequences of tape positions when we deal with computations.

DEFINITION 15.6. Suppose $t_1 t_2 \ldots t_n$ is a sequence of tape positions. Then the *code* of this sequence is

$$\ulcorner t_1 t_2 \ldots t_2 \urcorner = 2^{\ulcorner t_1 \urcorner} 3^{\ulcorner t_2 \urcorner} \ldots p_n^{\ulcorner t_n \urcorner} .$$

NOTE. Both tape positions and sequences of tape positions also have unique codes.

PROBLEM 15.8. *Pick some (short!) sequence of tape positions and find its code.*

Having defined how to represent tape positions as integers, we now need to manipulate these representations using recursive functions. The recursive functions and relations in Problem 14.9 provide the necessary tools.

PROBLEM 15.9. *Show that each of the following is primitive recursive.*

1. *The* 4*-place function* ENTRY, *where*

$$\text{ENTRY}(j,w,t,n) = \begin{cases} \ulcorner(i+w-1,t,\mathbf{a}')\urcorner & \text{if } n = \ulcorner(i,s,\mathbf{a})\urcorner,\ j \in \{0,1\}, \\ & w \in \{0,2\},\ \text{and } i+w-1 \geq 0;\ \text{where} \\ & a'_k = a_k \text{ for } k \neq i \text{ and } a'_i = j; \\ 0 & \text{otherwise.} \end{cases}$$

2. *For any Turing machine M with alphabet $\{1\}$, the* 1*-place function* TM_M*, such that*

$$\text{TM}_M(n) = \begin{cases} \ulcorner\mathbf{M}(i,s,\mathbf{a})\urcorner & \text{if } n = \ulcorner(i,s,\mathbf{a})\urcorner \\ & \text{and } \mathbf{M}(i,s,\mathbf{a}) \text{ is defined;} \\ 0 & \text{otherwise.} \end{cases}$$

3. *For any Turing machine M with alphabet $\{1\}$, the* 1*-place relation* COMP_M*, where*

$$\text{COMP}_M(n) \iff n \text{ is the code of a computation of } M.$$

The functions and relations above may be primitive recursive, but the last step in showing that Turing computable functions are recursive requires unbounded minimalization.

THEOREM 15.10. *Any k-place Turing computable function is recursive.*

One can push the techniques used above just a little farther to get a recursive function that simulates any Turing machine. Since any recursive function can be computed by some Turing machine, this effectively gives us another universal Turing machine.

PROBLEM 15.11. *Devise a suitable definition for the code $\ulcorner M\urcorner$ of a Turing machine M with alphabet $\{1\}$.*

PROBLEM 15.12. *Show, using your definition of $\ulcorner M\urcorner$ from Problem 15.11, that the following are primitive recursive.*

1. *The* 2*-place function* TM*, where*

$$\mathrm{TM}(m,n) = \begin{cases} \ulcorner \mathbf{M}(i,s,\mathbf{a}) \urcorner & \text{if } m = \ulcorner M \urcorner \text{ for some machine } M, \\ & n = \ulcorner (i,s,\mathbf{a}) \urcorner, \\ & \text{and } \mathbf{M}(i,s,\mathbf{a}) \text{ is defined;} \\ 0 & \text{otherwise.} \end{cases}$$

2. *The* 2*-place relation* COMP*, where*

$$\textsc{Comp}(m,n) \iff m = \ulcorner M \urcorner$$

for some Turing machine M and n is the code of a computation of M.

PROBLEM 15.13. *Show that the* 2*-place partial function* SIM *is recursive, where, for any Turing machine M with alphabet $\{1\}$ and input tape $\mathbf{a}$ for M,*

$$\textsc{Sim}(\ulcorner M \urcorner, \ulcorner (0,1,\mathbf{a}) \urcorner) = \ulcorner (0,1,\mathbf{b}) \urcorner$$

if *M halts with output $\mathbf{b}$ on input $\mathbf{a}$.*

Note that $\textsc{Sim}(m,n)$ may be undefined on other inputs.

Recursively enumerable sets. The following notion is of particular interest in the advanced study of computability.

DEFINITION 15.7. A subset (*i.e.* a 1-place relation) P of $\mathbb{N}$ is *recursively enumerable*, often abbreviated as *r.e.*, if there is a 1-place recursive function f such that $P = \mathrm{im}(f) = \{\, f(n) \mid n \in \mathbb{N} \,\}$.

Since the image of any recursive 1-place function is recursively enumerable by definition, we do not lack for examples. For one, the set E of even natural numbers is recursively enumerable, since it is the image of $f(n) = \textsc{Mult}(\mathrm{S}(\mathrm{S}(\mathrm{O}(n))), n)$.

PROPOSITION 15.14. *If P is a* 1*-place recursive relation, then P is recursively enumerable.*

This proposition is not reversible, but it does come close.

PROPOSITION 15.15. *$P \subseteq \mathbb{N}$ is recursive if and only if both P and $\mathbb{N} \setminus P$ are recursively enumerable.*

PROBLEM 15.16. *Find an example of a recursively enumerable set which is not recursive.*

PROBLEM 15.17. *Is $P \subseteq \mathbb{N}$ primitive recursive if and only if both P and $\mathbb{N} \setminus P$ are enumerable by primitive recursive functions?*

Incompleteness

CHAPTER 16

Preliminaries

It was mentioned in the Introduction that one of the motivations for the development of notions of computability was the following question.

Entscheidungsproblem. Given a reasonable set Σ of formulas of a first-order language $\mathcal{L}$ and a formula φ of $\mathcal{L}$, is there an effective method for determining whether or not $\Sigma \vdash \varphi$? □

Armed with knowledge of first-order logic on the one hand and of computability on the other, we are in a position to formulate this question precisely and then solve it. To cut to the chase, the answer is "no" in general. Gödel's Incompleteness Theorem asserts, roughly, that for any computable set of axioms in a first-order language powerful enough to prove certain facts about arithmetic, it is possible to formulate statements in the language whose truth is not decided by the axioms. In particular, it turns out that no consistent set of axioms can hope to prove its own consistency.

We will tackle the Incompleteness Theorem in three stages. First, we will code the formulas and proofs of a first-order language as numbers and show that the functions and relations involved are recursive. This will, in particular, make it possible for us to define "computable set of axioms" precisely. Second, we will show that all recursive functions and relations can be defined by first-order formulas in the presence of a fairly minimal set of axioms about elementary number theory. Finally, by putting recursive functions talking about first-order formulas together with first-order formulas defining recursive functions, we will manufacture a self-referential sentence which asserts its own unprovability.

A language for first-order number theory. To keep things as concrete as possible we will work with and in the following language for first-order number theory, used as an example in Chapter 5.

Definition 16.1. $\mathcal{L}_N$ is the first-order language with the following symbols:

1. Parentheses: (and)
2. Connectives: $\neg$ and $\rightarrow$

3. Quantifier: $\forall$
4. Equality: $=$
5. Variable symbols: v_0, v_2, v_3, ...
6. Constant symbol: 0
7. 1-place function symbol: S
8. 2-place function symbols: $+$, $\cdot$, and E.

The non-logical symbols of $\mathcal{L}_N$, 0, S, $+$, $\cdot$, and E, are intended to name, respectively, the number zero, and the successor, addition, multiplication, and exponentiation functions on the natural numbers. That is, the (standard!) structure this language is intended to discuss is $\mathfrak{N} = (\mathbb{N}, 0, \mathrm{S}, +, \cdot, \mathrm{E})$.

NOTE. Notation for and the definitions of terms, formulas, sentences, structures, interpretations, logical axioms, deductions, and so on, of first-order languages, plus various conventions involving these, are given in Chapters 5-8 of Volume I. Look them up as (and if) you need to.

Completeness. The notion of completeness mentioned in the Incompleteness Theorem is different from the one mentioned in the Completeness Theorem.[1] "Completeness" in the latter sense is a property of a logic: it asserts that whenever $\Gamma \models \sigma$ (*i.e.* the truth of the sentence σ follows from that of the set of sentences Γ), $\Gamma \vdash \sigma$ (*i.e.* there is a deduction of σ from Γ). The sense of "completeness" in the Incompleteness Theorem, defined below, is a property of a set of sentences.

DEFINITION 16.2. A set of sentences Σ of a first-order language $\mathcal{L}$ is said to be *complete* if for every sentence τ either $\Sigma \vdash \tau$ or $\Sigma \vdash \neg\tau$.

That is, a set of sentences, or non-logical axioms, is complete if it suffices to prove or disprove every sentence of the langage in in question.

PROBLEM 16.1. *Show that a consistent set Σ of sentences of a first-order language $\mathcal{L}$ is complete if and only if the theory of Σ,*

$$\mathrm{Th}(\Sigma) = \{\, \tau \mid \tau \textit{ is a sentence of } \mathcal{L} \textit{ and } \Sigma \vdash \tau \,\},$$

is maximally consistent.

[1]Which, to confuse the issue, was also first proved by Gödel.

CHAPTER 17

Coding First-Order Logic

We will encode the symbols, formulas, and deductions of $\mathcal{L}_N$ as natural numbers in such a way that the operations necessary to manipulate these codes are recursive. Although we will do so just for $\mathcal{L}_N$, any countable first-order language can be coded in a similar way.

DEFINITION 17.1. To each symbol s of $\mathcal{L}_N$ we assign an unique positive integer $\ulcorner s \urcorner$, the *Gödel code* of s, as follows:

1. $\ulcorner (\urcorner = 1$ and $\ulcorner) \urcorner = 2$
2. $\ulcorner \neg \urcorner = 3$ and $\ulcorner \rightarrow \urcorner = 4$
3. $\ulcorner \forall \urcorner = 5$
4. $\ulcorner = \urcorner = 6$.
5. $\ulcorner v_k \urcorner = k + 12$
6. $\ulcorner 0 \urcorner = 7$
7. $\ulcorner S \urcorner = 8$
8. $\ulcorner + \urcorner = 9$, $\ulcorner \cdot \urcorner = 10$, and $\ulcorner E \urcorner = 11$

Note that each positive integer is the Gödel code of one and only one symbol of $\mathcal{L}_N$. We will also need to code sequences of the symbols of $\mathcal{L}_N$, such as terms and formulas, as numbers, not to mention sequences of sequences of symbols of $\mathcal{L}_N$, such as deductions.

DEFINITION 17.2. Suppose $s_1 s_2 \dots s_k$ is a sequence of symbols of $\mathcal{L}_N$. Then the *Gödel code* of this sequence is

$$\ulcorner s_1 \dots s_k \urcorner = p_1^{\ulcorner s_1 \urcorner} \dots p_k^{\ulcorner s_k \urcorner},$$

where p_n is the nth primes number.

Similarly, if $\sigma_1 \sigma_2 \dots \sigma_\ell$ is a sequence of sequences of symbols of $\mathcal{L}_N$, then the *Gödel code* of this sequence is

$$\ulcorner \sigma_1 \dots \sigma_\ell \urcorner = p_1^{\ulcorner \sigma_1 \urcorner} \dots p_k^{\ulcorner \sigma_\ell \urcorner}.$$

EXAMPLE 17.1. $\ulcorner \forall v_1 = \cdot v_1 S 0 v_1 \urcorner = 2^5 3^{13} 5^6 7^{10} 11^{13} 13^8 17^7 19^{13}$.

A particular integer n may simultaneously be the Gödel code of a symbol, a sequence of symbols, and a sequence of sequences of symbols of $\mathcal{L}_N$. We shall rely on context to avoid confusion, but, with some more work, one could set things up so that no integer was the code of more than one kind of thing.

We need to show that various relations and functions for recognizing and manipulating Gödel codes are recursive.

PROBLEM 17.1. *Show that each of the following relations is primitive recursive.*

1. $\text{TERM}(n) \iff n = \ulcorner t \urcorner$ *for some term* t *of* $\mathcal{L}_N$.
2. $\text{FORMULA}(n) \iff n = \ulcorner \varphi \urcorner$ *for some formula* φ *of* $\mathcal{L}_N$.
3. $\text{SENTENCE}(n) \iff n = \ulcorner \sigma \urcorner$ *for some sentence* σ *of* $\mathcal{L}_N$.
4. $\text{LOGICAL}(n) \iff n = \ulcorner \gamma \urcorner$ *for some logical axiom* γ *of* $\mathcal{L}_N$.

Using these relations as building blocks, we will develop relations and functions to handle deductions of $\mathcal{L}_N$. First, though, we need to make "a computable set of formulas" precise.

DEFINITION 17.3. A set Δ of formulas of $\mathcal{L}_N$ is said to be *recursive* if the set of Gödel codes of formulas of Δ,

$$\ulcorner \Delta \urcorner = \{ \ulcorner \delta \urcorner \mid \delta \in \Delta \},$$

is recursive. Similarly, Δ is said to be *recursively enumerable* if $\ulcorner \Delta \urcorner$ is recursively enumerable.

PROBLEM 17.2. *Suppose* Δ *is a recursive set of sentences of* $\mathcal{L}_N$. *Show that each of the following relations is recursive.*

1. $\text{PREMISS}_\Delta(n) \iff n = \ulcorner \beta \urcorner$ *for some formula* β *of* $\mathcal{L}_N$ *which is either a logical axiom or in* Δ.
2. $\text{FORMULAS}(n) \iff n = \ulcorner \varphi_1 \dots \varphi_k \urcorner$ *for some sequence* $\varphi_1 \dots \varphi_k$ *of formulas of* $\mathcal{L}_N$.
3. $\text{INFERENCE}(n, i, j) \iff n = \ulcorner \varphi_1 \dots \varphi_k \urcorner$ *for some sequence* $\varphi_1 \dots \varphi_k$ *of formulas of* $\mathcal{L}_N$, $1 \le i, j \le k$, *and* φ_k *follows from* φ_i *and* φ_j *by Modus Ponens.*
4. $\text{DEDUCTION}_\Delta(n) \iff n = \ulcorner \varphi_1 \dots \varphi_k \urcorner$ *for a deduction* $\varphi_1 \dots \varphi_k$ *from* Δ *in* $\mathcal{L}_N$.
5. $\text{CONCLUSION}_\Delta(n, m) \iff n = \ulcorner \varphi_1 \dots \varphi_k \urcorner$ *for a deduction* $\varphi_1 \dots \varphi_k$ *from* Δ *in* $\mathcal{L}_N$ *and* $m = \ulcorner \varphi_k \urcorner$.

If $\ulcorner \Delta \urcorner$ *is primitive recursive, which of these are primitive recursive?*

It is at this point that the connection between computability and completeness begins to appear.

THEOREM 17.3. *Suppose* Δ *is a recursive set of sentences of* $\mathcal{L}_N$. *Then* $\ulcorner \text{Th}(\Delta) \urcorner$ *is*

1. *recursively enumerable, and*
2. *recursive if and only if* Δ *is complete.*

NOTE. It follows that $\ulcorner \mathrm{Th}(\Delta) \urcorner$ is an example of a recursively enumerable but not recursive set if Δ is not complete.

CHAPTER 18

Defining Recursive Functions In Arithmetic

We will also need definitions and results complementing those obtained in Chapter 17: a set of non-logical axioms in $\mathcal{L}_N$ which prove enough to let us define all the recursive functions by suitable formulas of $\mathcal{L}_N$. The non-logical axioms in question essentially just ensure that basic arithmetic works properly.

DEFINITION 18.1. Let $\mathcal{A}$ be the following set of sentences of $\mathcal{L}_N$, written out in official form.

1. $\forall v_0 \, (\neg = Sv_0 0)$
2. $\forall v_0 \, ((\neg = v_0 0) \to (\neg \forall v_1 \, (\neg = Sv_1 v_0)))$
3. $\forall v_0 \forall v_1 \, (= Sv_0 Sv_1 \to = v_0 v_1)$
4. $\forall v_0 \, = +v_0 0 v_0$
5. $\forall v_0 \forall v_1 \, = +v_0 Sv_1 S + v_0 v_1$
6. $\forall v_0 \, = \cdot v_0 0 0$
7. $\forall v_0 \forall v_1 \, = \cdot v_0 Sv_1 + \cdot v_0 v_1 v_0$
8. $\forall v_0 \, = Ev_0 0 S0$
9. $\forall v_0 \forall v_1 \, = Ev_0 Sv_1 \cdot Ev_0 v_1 v_0$

Translated from the official forms, $\mathcal{A}$ consists of the following axioms about the natural numbers:

1. $\forall x \, x + 1 \neq 0$
2. $\forall x \, x \neq 0 \to \exists y \, y + 1 = x$
3. $\forall x \forall y \, x + 1 = y + 1 \to x = y$
4. $\forall x \, x + 0 = x$
5. $\forall x \forall y \, x + y + 1 = (x + y) + 1$
6. $\forall x \, x \cdot 0 = 0$
7. $\forall x \forall y \, x \cdot (y + 1) = (x \cdot y) + x$
8. $\forall x \, x^0 = 1$
9. $\forall x \forall y \, x^{y+1} = (x^y) \cdot x$

Each of the axioms in $\mathcal{A}$ is true in $\mathfrak{N} = (\mathbb{N}, 0, \mathrm{S}, +, \cdot, \mathrm{E})$. However, they are a long way from being able to prove all the sentences of first-order arithmetic true in $\mathfrak{N}$. For example, though we won't prove it, it turns out that $\mathcal{A}$ is not enough to ensure that induction works: that for every formula φ with at most the variable x free, if φ^x_0 and

$\forall y\,(\varphi^x_y \to \varphi^x_{Sy})$ hold, then so does $\forall x\,\varphi$. On the other hand, neither $\mathcal{L}_N$ nor $\mathcal{A}$ are quite as minimal as they might be. For example, one could do without E and define it from $\cdot$ and $+$.

It will be convenient to use a couple of conventions in what follows. First, we will often abbreviate the term of $\mathcal{L}_N$ consisting of m Ss followed by 0 by S^m0. For example, S^30 abbreviates $SSS0$. We will use S^m0 to represent the natural number m in $\mathcal{L}_N$. (The interpretation of S^m0 in $\mathfrak{N}$ will, in fact, be the mth successor of 0, namely m.) Second, if φ is a formula of $\mathcal{L}_N$ with all of its free variables among $v_1, \dots, v_n$, and $m_0, m_1, \dots, m_n$ are natural numbers, we will write $\varphi(S^{m_1}0, \dots, S^{m_k}0)$ for the sentence $\varphi^{v_1 \dots v_n}_{S^{m_1}0, \dots, S^{m_k}0}$, *i.e.* φ with every free occurrence of v_i replaced by $S^{m_i}0$. Note that since the term $S^{m_i}0$ involves no variables, it must be substitutable for v_i in φ.

Definition 18.2. Suppose Σ is a set of sentences of $\mathcal{L}_N$. A k-place function f is said to be *representable* in $\mathrm{Th}(\Sigma) = \{\, \tau \mid \Sigma \vdash \tau \,\}$ if there is a formula φ of $\mathcal{L}_N$ with at most $v_1, \dots, v_k$, and v_{k+1} as free variables such that

$$\begin{aligned} f(n_1, \dots, n_k) = m &\iff \varphi(S^{n_1}0, \dots, S^{n_k}0, S^m0) \in \mathrm{Th}(\Sigma) \\ &\iff \Sigma \vdash \varphi(S^{n_1}0, \dots, S^{n_k}0, S^m0) \end{aligned}$$

for all $n_1, \dots, n_k$ in $\mathbb{N}$. Such a formula φ is said to *represent* f in $\mathrm{Th}(\Sigma)$.

Similarly, a k-place relation $P \subseteq \mathbb{N}^k$ is said to be *representable* in $\mathrm{Th}(\Sigma)$ if there is a formula ψ of $\mathcal{L}_N$ with at most $v_1, \dots, v_k$ as free variables such that

$$\begin{aligned} P(n_1, \dots, n_k) &\iff \psi(S^{n_1}0, \dots, S^{n_k}0) \in \mathrm{Th}(\Sigma) \\ &\iff \Sigma \vdash \psi(S^{n_1}0, \dots, S^{n_k}0) \end{aligned}$$

for all $n_1, \dots, n_k$ in $\mathbb{N}$. Such a formula ψ is said to *represent* P in $\mathrm{Th}(\Sigma)$.

We will use this definition mainly with $\Sigma = \mathcal{A}$.

Example 18.1. The constant function c_3 given by $c_3(n) = 3$ is representable in $\mathrm{Th}(\mathcal{A})$; $v_2 = SSS0$ is a formula representing it. Note that that this formula has no free variable for the input of the 1-place function in question, but then the input is irrelevant ...

Almost the same formula, $v_1 = SSS0$, serves to represent the set — *i.e.* 1-place relation — $\{3\}$ in $\mathrm{Th}(\mathcal{A})$.

Example 18.2. The set of all even numbers is a 1-place relation is representable in $\mathrm{Th}(\mathcal{A})$; $\exists v_0\, v_1 = S0 \cdot v_1$ is a formula representing it.

EXAMPLE 18.3. The projection function π_2^3 can be represented in $\mathrm{Th}(\mathcal{A})$. $v_2 = v_4$ is one formula which represents π_2^3; another is $\exists v_7\,(v_2 = v_7 \land v_7 = v_4)$.

PROBLEM 18.1. *Suppose Σ and Γ are sets of sentences of $\mathcal{L}_N$ and $\Sigma \vdash \Gamma$, i.e. $\Sigma \vdash \gamma$ for every $\gamma \in \Gamma$. Then every function and relation which is representable in $\mathrm{Th}(\Gamma)$ is representable in $\mathrm{Th}(\Sigma)$.*

PROBLEM 18.2. *Suppose Σ is a set of sentences of $\mathcal{L}_N$ and f is a k-place function which is representable in $\mathrm{Th}(\Sigma)$. Then Σ must be consistent.*

It turns out that all recursive functions and relations are representable in $\mathrm{Th}(\mathcal{A})$.

PROBLEM 18.3. *Show that the following functions are representable in $\mathrm{Th}(\mathcal{A})$:*

1. *The zero function $\mathrm{O}(n) = 0$.*
2. *The successor function $\mathrm{S}(n) = n + 1$.*
3. *For every positive k and $i \leq k$, the projection function π_i^k.*

PROPOSITION 18.4. *A k-place function f is representable in $\mathrm{Th}(\mathcal{A})$ if and only if the $k+1$-place relation P_f defined by*

$$P_f(n_1, \dots, n_k, n_{k+1}) \iff f(n_1, \dots, n_k) = n_{k+1}$$

is representable in $\mathrm{Th}(\mathcal{A})$.

Also, a relation $P \subseteq \mathbb{N}^k$ is representable in $\mathrm{Th}(\mathcal{A})$ if and only if its characteristic function χ_P is representable in $\mathrm{Th}(\mathcal{A})$.

PROPOSITION 18.5. *Suppose $g_1, \dots, g_m$ are k-place functions and h is an m-place function, all of them representable in $\mathrm{Th}(\mathcal{A})$. Then $f = h \circ (g_1, \dots, g_m)$ is a k-place function representable in $\mathrm{Th}(\mathcal{A})$.*

PROPOSITION 18.6. *Suppose g is a $k + 1$-place regular function which is representable in $\mathrm{Th}(\mathcal{A})$. Then the unbounded minimalization of g is a k-place function representable in $\mathrm{Th}(\mathcal{A})$.*

Between them, the above results supply most of the ingredients needed to conclude that all recursive functions and relations on the natural numbers are representable. The exception is showing that functions defined by primitive recursion from representable functions are also representable, which requires some additional effort. The basic problem is that it is not obvious how a formula defining a function can get at previous values of the function. To accomplish this, we will borrow a trick from Chapter 14.

PROBLEM 18.7. *Show that each of the following relations and functions (first defined in Problem 14.9) is representable in* $\mathrm{Th}(\mathcal{A})$.

1. $\textsc{Div}(n,m) \iff n \mid m$
2. $\textsc{IsPrime}(n) \iff n$ *is prime*
3. $\textsc{Prime}(k) = p_k$, *where* $p_0 = 1$ *and* p_k *is the* k*th prime if* $k \geq 1$.
4. $\textsc{Power}(n,m) = k$, *where* $k \geq 0$ *is maximal such that* $n^k \mid m$.
5. $\textsc{Length}(n) = \ell$, *where* ℓ *is maximal such that* $p_\ell \mid n$.
6. $\textsc{Element}(n,i) = n_i$, *where* $n = p_1^{n_1} \dots p_k^{n_k}$ *(and* $n_i = 0$ *if* $i > k$*).*

Using the representable functions and relations given above, we can represent the "history" function of any representable function ...

PROBLEM 18.8. *Suppose* f *is a* k*-place function representable in* $\mathrm{Th}(\mathcal{A})$. *Show that*

$$\begin{aligned} F(n_1,\dots,n_k,m) &= p_1^{f(n_1,\dots,n_k,0)} \dots p_{m+1}^{f(n_1,\dots,n_k,m)} \\ &= \prod_{i=0}^{m} p_i^{f(n_1,\dots,n_k,i)} \end{aligned}$$

is also representable in $\mathrm{Th}(\mathcal{A})$.

... and use it!

PROPOSITION 18.9. *Suppose* g *is a* k*-place function and* h *is a* $k+2$*-place function, both representable in* $\mathrm{Th}(\mathcal{A})$. *Then the* $k+1$*-place function* f *defined by primitive recursion from* g *and* h *is also representable in* $\mathrm{Th}(\mathcal{A})$.

THEOREM 18.10. *Recursive functions are representable in* $\mathrm{Th}(\mathcal{A})$.

In particular, it follows that there are formulas of $\mathcal{L}_N$ representing each of the functions from Chapter 17 for manipulating the codes of formulas. This will permit us to construct formulas which encode assertions about terms, formulas, and deductions; we will ultimately prove the Incompleteness Theorem by showing there is a formula which codes its own unprovability.

CHAPTER 19

The Incompleteness Theorem

By pulling the material in Chapters 16–18 together, we can finally state and prove the Incompleteness Theorem.

PROBLEM 19.1. *Show that $\mathcal{A}$ is a recursive set of sentences of $\mathcal{L}_N$.*

PROBLEM 19.2. *Show that the function*

$$\text{SUB}(n,k) = \begin{cases} \ulcorner\varphi(S^k 0)\urcorner & \textit{if } n = \ulcorner\varphi\urcorner \textit{ for a formula } \varphi \textit{ of } \mathcal{L}_N \\ & \textit{with at most } v_1 \textit{ free} \\ 0 & \textit{otherwise} \end{cases}$$

is recursive.

The key result needed to prove the Incompleteness Theorem is the following lemma.

LEMMA 19.3 (Fixed-Point Lemma). *Suppose φ is a formula of $\mathcal{L}_N$ with only v_1 as a free variable. Then there is a sentence σ of $\mathcal{L}_N$ such that*

$$\mathcal{A} \vdash \sigma \leftrightarrow \varphi(S^{\ulcorner\sigma\urcorner}0).$$

Note that σ must be different from the sentence $\varphi(S^{\ulcorner\sigma\urcorner}0)$: there is no way to find a formula φ with one free variable and an integer k such that $\ulcorner\varphi(S^k 0)\urcorner = k$. (Think about how Gödel codes are defined ...)

With the Fixed-Point Lemma in hand, Gödel's Incompleteness Theorem can be put away in fairly short order.

THEOREM 19.4 (Gödel's Incompleteness Theorem). *Suppose Σ is a consistent recursive set of sentences of $\mathcal{L}_N$ such that $\Sigma \vdash \mathcal{A}$. Then Σ is not complete.*

That is, any consistent set of sentences which proves at least as much about the natural numbers as $\mathcal{A}$ does can't be both complete and recursive. The Incompleteness Theorem has many variations and corollaries; [**11**] is a good place to learn about many of them.

PROBLEM 19.5. *Prove each of the following.*

1. *Let Γ be a complete set of sentences of $\mathcal{L}_N$ such that $\Gamma \cup \mathcal{A}$ is consistent. Then Γ is not recursive.*

2. *Let* Δ *be a recursive set of sentences such that* $\Delta \cup \mathcal{A}$ *is consistent. Then* Δ *is not complete.*
3. *The theory of* $\mathfrak{N}$*,*

$$\mathrm{Th}(\mathfrak{N}) = \{\, \sigma \mid \sigma \text{ is a sentence of } \mathcal{L}_N \text{ and } \mathfrak{N} \models \sigma \,\},$$

is not recursive.

There is nothing all that special about working in $\mathcal{L}_N$. The proof of Gödel's Incompleteness Theorem can be executed for any first order language and recursive set of axioms which allow one to code and prove enough facts about arithmetic. In particular, it can be done whenever the language and axioms are powerful enough — as in Zermelo-Fraenkel set theory, for example — to define the natural numbers and prove some modest facts about them.

Gödel also proved a strengthened version of the Incompleteness Theorem which asserts that a consistent recursive set of sentences Σ of $\mathcal{L}_N$ cannot prove its own consistency. To get at it, we need to express the statement "Σ is consistent" in $\mathcal{L}_N$.

PROBLEM 19.6. *Suppose* Σ *is a recursive set of sentences of* $\mathcal{L}_N$*. Find a sentence of* $\mathcal{L}_N$*, which we'll denote by* $\mathrm{Con}(\Sigma)$*, such that* Σ *is consistent if and only if* $\mathcal{A} \vdash \mathrm{Con}(\Sigma)$*.*

THEOREM 19.7 (Gödel's Second Incompleteness Theorem). *Let* Σ *be a consistent recursive set of sentences of* $\mathcal{L}_N$ *such that* $\Sigma \vdash \mathcal{A}$*. Then* $\Sigma \nvdash \mathrm{Con}(\Sigma)$*.*

As with the (First) Incompleteness Theorem, the Second Incompleteness Theorem holds for any recursive set of sentences in a first-order language which allow one to code and prove enough facts about arithmetic. The perverse consequence of the Second Incompleteness Theorem is that only an inconsistent set of axioms can prove its own consistency ...

The implications. Gödel's Incompleteness Theorems have profound implications.

Since almost all of mathematics can be formalized in first-order logic, the First Incompleteness Theorem implies that there is no effective procedure that will find and prove all theorems. This might be considered as job security for research mathematicians ...

The Second Incompleteness Theorem, on the other hand, implies that we can never be completely sure that any reasonable set of axioms is actually consistent unless we take a more powerful set of axioms on faith. It follows that one can never be completely sure — faith aside —

that the theorems proved in mathematics are really true. This might be considered as job security for philosophers of mathematics ...

Truth and definability. A close relative of the Incompleteness Theorem is the assertion that truth in $\mathfrak{N} = (\mathbb{N}, \mathrm{S}, +, \cdot, \mathrm{E}, 0)$ is not definable in $\mathfrak{N}$. To make sense of this, of course, we first need to define what "definable in $\mathfrak{N}$ means.

DEFINITION 19.1. A k-place relation is *definable* in $\mathfrak{N}$ if there is a formula φ of $\mathcal{L}_N$ with at most $v_1, \dots, v_k$ as free variables such that

$$P(n_1, \dots, n_k) \iff \mathfrak{N} \models \varphi[s(v_1|n_1)\dots(v_k|n_k)]$$

for every assignment s of $\mathfrak{N}$. Such a formula φ is said to *define* P in $\mathfrak{N}$.

A definition of "function definable in $\mathfrak{N}$" could be made in a similar way, of course. Definability is a close relative of representability:

PROPOSITION 19.8. *Suppose P is a k-place relation which is representable in* $\mathrm{Th}(\mathcal{A})$. *Then P is definable in $\mathfrak{N}$.*

PROBLEM 19.9. *Is the converse to Proposition 19.6 true?*

A counterpart for definability of the Entscheidungsproblem is the question of whether the truth in $\mathfrak{N}$ is a definable relation in $\mathfrak{N}$, *i.e.* whether the set of Gödel codes of sentences of $\mathcal{L}_N$ true in $\mathfrak{N}$,

$$\ulcorner \mathrm{Th}(\mathfrak{N}) \urcorner = \{ \ulcorner \sigma \urcorner \mid \sigma \text{ is a sentence of } \mathcal{L}_N \text{ and } \mathfrak{N} \models \sigma \},$$

is definable in $\mathfrak{N}$.

THEOREM 19.10 (Tarski's Undefinability Theorem). $\ulcorner \mathrm{Th}(\mathfrak{N}) \urcorner$ *is not definable in $\mathfrak{N}$.*

Hints

CHAPTER 10

Hints

10.1. This should be easy ...

10.2. Ditto.

10.3. 1. Any machine with the given alphabet and a table with three non-empty rows will do.
2. Every entry in the table in the 0 column must write a 1 in the scanned cell; similarly, every entry in the 1 column must write a 0 in the scanned cell.
3. What's the simplest possible table for a given alphabet?

10.4. Unwind the definitions step by step in each case. Not all of these are computations ...

10.5. Examine your solutions to the previous problem and, if necessary, take the computations a little farther.

10.6. Have the machine run on forever to the right, writing down the desired pattern as it goes no matter what may be on the tape already.

10.7. Consider your solution to Problem 10.6 for one possible approach. It should be easy to find simpler solutions, though.

10.8. 1. Use four states to write the 1s, one for each.
2. The input has a convenient marker.
3. Run back and forth to move one marker n cells *from* the block of 1's while moving another *through* the block, and then fill in.
4. Modify the previous machine by having it delete every other 1 after writing out 1^{2n}.
5. Run back and forth to move the right block of 1s cell by cell to the desired position.
6. Run back and forth to move the left block of 1s cell by cell past the other two, and then apply a minor modification of the previous machine.
7. Run back and forth between the blocks, moving a marker through each. After the race between the markers to the ends of their

respective blocks has been decided, erase everything and write down the desired output.

CHAPTER 11

Hints

11.1. This ought to be easy.

11.2. Generalize the technique of Example 11.1, adding two new states to help with each old state that may cause a move in different directions. Be careful not to make a machine that would run off the end of the tape when the original wouldn't.

11.3. Note that the simulation must operate with coded versions of Ms tape, unless $\Sigma = \{1\}$. The key idea is to use the tape of the simulator in blocks of some fixed size, with the patterns of 0s and 1s in each block corresponding to elements of Σ.

11.4. This should be straightforward, if somewhat tedious. You do need to be careful in coming up with the appropriate input tapes for O.

11.5. Generalize the technique of Example 11.2, splitting up the tape of the simulator into upper and lower tracks and splitting each state of N into two states in P. You will need to be quite careful in describing just how the latter is to be done.

11.6. If you're in doubt, go with one read/write scanner for each tape, and have each entry in the table of a two-tape machine take both scanners into account. Simulating such a machine is really just a variation on the techniques used in Example 11.2.

11.7. Such a machine should be able to move its scanner to cells up and down from the current one, as well to the side. (Diagonally too, if you want to!) Simulating such a machine on a single tape machine is a challenge. You might find it easier to first describe how to simulate it on a suitable multiple-tape machine.

CHAPTER 12

Hints

12.1. Pick a *simple* Turing machine ...

12.2. Unwind the representation using Definition 12.1.

12.3. Trick question!

12.4. One could omit the representations of any empty entries in the table of a Turing machine, for one thing.

12.5. Just use Definition 12.2 in each case.

12.6. Unwind each representation using Definition 12.2.

12.7. For one thing, is the leading 1 in the representation of each cell really necessary?

12.8. **H** needs to make exactly two changes to the representation of the tape position. Note that $i = 0$ is a special case.

12.9. **R** needs to check the representation of a particular cell.

12.10. **W** needs to make just one change to the representation of the tape position.

12.11. **E** must first find and then copy the representation of particular entry in the table of M. Some of the machines in Problem 10.8 may come in handy here.

12.12. Put the machines developed in Problems 12.8–12.11 together.

12.13. Use the machine **S** of Problem 12.12 to do most of the work.

12.14. Essentially, all **C** does is unwind Definition 12.2.

12.15. Assume, by way of contradiction, that there was a Turing machine **T** which solved the Halting Problem. Modify **T** to get a machine **Y** which, for any Turing machine M, halts on input $\underline{0}\ulcorner M \urcorner$ if and only if M does *not* halt on input $\underline{0}\ulcorner M \urcorner$. Now give **Y** input $\underline{0}\ulcorner \mathbf{Y} \urcorner$ ...

CHAPTER 13

Hints

13.1. 1. Delete most of the input.
2. Add a little to the input.
3. Add a little to the input, and delete a little more elsewhere.
4. Delete a little from the input *most* of the time.
5. Run back and forth between the two blocks in the input, deleting until one side disappears. Clean up appropriately!
6. Delete two of blocks and move the remaining one.

13.2. There are just as many functions $\mathbb{N} \to \mathbb{N}$ as there are real numbers, of which there are many more than there are natural numbers.

13.3. 1. Trace the computation through step-by-step.
2. Consider the scores of each of the 1-state entries in the busy beaver competition.
3. Find a 3-state entry in the busy beaver competition which scores six.
4. Show how to turn an n-state entry in the busy beaver competition into an $n+1$-state entry that scores just a little better.

13.4. You could start by looking at modifications of the 3-state entry you devised in Problem 13.3.

13.5. Suppose Σ was computable by a Turing machine M. Modify M to get an n-state entry in the busy beaver competition for some n which achieves a score greater than $\Sigma(n)$.

CHAPTER 14

Hints

14.1. You only need to take care of the projection functions, and these can be computed by Turing machines very similar to one another.

14.2. Generalize Example 14.1.

14.3. Use machines computing g, h_1, ..., h_m as sub-machines of the machine computing the composition. You might also find sub-machines that copy the original input and various stages of the output useful. It is important that each sub-machine get all the data it needs and does not damage the data needed by other sun-machines.

14.4. Proceed by induction on the number of applications of composition used to define f from the initial functions.

14.5. 1. Exponentiation is to multiplication as multiplication is to addition.
2. This is straightforward except for taking care of $\text{Pred}(0) = \text{Pred}(1) = 0$.
3. Diff is to Pred as S is to Sum.
4. This is straightforward if you let $0! = 1$.

14.6. Machines used to compute g and h are the principal parts of the machine computing f, along with parts to copy, move, and/or delete data on the tape between stages in the recursive process.

14.7. 1. f is to g as Fact is to the identity function.
2. Use Diff and a suitable constant function as the basic building blocks.
3. This is a slight generalization of the preceding part.

14.8. Proceed by induction on the number of applications of primitive recursion and composition.

14.9. 1. Use a composition including Diff, χ_P, and a suitable constant function.
2. A suitable composition will do the job; it's a little harder than it looks.

3. A suitable composition will do the job; it's much more straightforward than the previous part.
4. Note tht $n = m$ exactly when $n - m = 0 = m - n$.
5. Compare this with Problem 14.5.
6. First devise a characteristic function for the relation

$$\text{Divides}(n, k, m) \iff nk = m\,,$$

and then sum up.
7. Use Div and sum up.
8. Use IsPrime and some ingenuity.
9. Use Exp and Div and some more ingenuity.
10. A suitable combination of Prime with other things will do.
11. A suitable combination of Prime and Power will do.
12. Throw the kitchen sink at this one ...
13. Ditto.

14.10. For the hard direction, do an induction on how g was built up from the initial functions.

14.11. A straightforward application of Theorem 14.10.

14.12. This is not unlike, though a little more complicated than, showing that primitive recursion preserves computability.

14.13. It's not easy! Look it up ...

14.14. This is a very easy consequence of Theorem 14.13.

14.15. Listing the definitions of all possible primitive recursive functions is a computable task. Borrow a trick from Cantor's proof that the real numbers are uncountable. (A formal argument to this effect could be made using techniques similar to those used to show that all Turing computable functions are recursive in the next chapter.)

CHAPTER 15

Hints

15.1. The strategy is obvious ... Make sure that at each stage you preserve a copy of the original input for use at later stages.

15.2. The primitive recursive function you define only needs to check values of $g(n_1, \ldots, n_k, m)$ for m such that $0 \leq m \leq h(n_1, \ldots, n_k)$, but it still needs to pick the least m such that $g(n_1, \ldots, n_k, m) = 0$.

15.3. This is very similar to Theorem 14.8.

15.4. This is virtually identical to Theorem 14.10.

15.5. This is virtually identical to Corollary 14.11.

15.6. In both cases, emulate Example 15.1.

15.7. Unwind Definition 15.5; you will need to do some factoring.

15.8. Find the codes of each of the positions in the sequence you chose and then apply Definition 15.6.

15.9.
1. It will probably be convenient to first devise a function which recognizes whether the input is of the correct form or not. If it is, make the necessary changes in n using the tools in Problem 14.9.
2. Piece TM_M together by cases using the function ENTRY in each case.
3. It will probably be convenient to first devise a function which recognizes whether the input is a sequence of tape positions or not. Use the function TM_M in checking that a sequence is a computation.

15.10. The last part of Problem 15.9 and some unbounded minimalization are the key ingredients, though you will need to compose with some auxiliary functions to get exactly what you want.

15.11. Take some creative inspiration from Definitions 15.5 and 15.6.

15.12. Much of what you need for both parts is just what was needed for Problem 15.9. The additional ingredients mainly have to do with using $m = \ulcorner M \urcorner$ properly.

15.13. Essentially, this is to Problem 15.12 as proving Theorem 15.10 is to Problem 15.9.

15.14. Use χ_P to help define a function f such that $\mathrm{im}(f) = P$.

15.15. One direction is an easy application of Proposition 15.14. For the other, given an $n \in \mathbb{N}$, run the functions enumerating P and $\mathbb{N} \setminus P$ concurrently until one or the other outputs n.

15.16. Consider the set of natural numbers coding (according to some scheme you must devise) Turing machines and an input tapes on which they halt.

15.17. See how far you can adapt your argument for Proposition 15.15.

CHAPTER 16

Hints

16.1. Compare Definition 16.2 with the definition of maximal consistency.

CHAPTER 17

Hints

17.1. In each case, use Definitions 17.1 and 17.2, together with the appropriate definitions from first-order logic and the tools developed in Problem 14.9.

17.2. In each case, use Definitions 17.1 and 17.2, together with the appropriate definitions from first-order logic and the tools developed in Problems 14.9 and 17.1. (They're all primitive recursive if $\ulcorner\Delta\urcorner$ is, by the way.)

17.3.
1. Use unbounded minimalization and the relations in Problem 17.2 to define a function which, given n, returns the nth integer which codes an element of $\mathrm{Th}(\Delta)$.
2. If Δ is complete, then for any sentence σ, either $\lceil\sigma\rceil$ or $\lceil\neg\sigma$ must eventually turn up in an enumeration of $\ulcorner\mathrm{Th}(\Delta)\urcorner$. The other direction is really just a matter of unwinding the definitions involved.

CHAPTER 18

Hints

18.1. Every deduction from Γ can be replaced by a dedudction of Σ with the same conclusion.

18.2. If Σ were insconsistent it would prove entirely too much ...

18.3. 1. Adapt Example 18.1.
2. Use the 1-place function symbol S of $\mathcal{L}_N$.
3. There is much less to this part than meets the eye ...

18.4. In each case, you need to use the given representing formula to define the one you need.

18.5. String together the formulas representing $g_1, \ldots, g_m$, and h with $\wedge$s and put some existential quantifiers in front.

18.6. First show that that $<$ is representable in $\mathrm{Th}(\mathcal{A})$ and then exploit this fact.

18.7. 1. $n \mid m$ if and only if there is some k such that $n \cdot k = m$.
2. n is prime if and only if there is no ℓ such that $\ell \mid n$ and $1 < \ell < n$.
3. p_k is the first prime with exactly $k - 1$ primes less than it.
4. Note that k must be minimal such that $n^{k+1} \nmid m$.
5. You'll need a couple of the previous parts.
6. Ditto.

18.8. Problem 18.7 has most of the necessary ingredients needed here.

18.9. Problems 18.7 and 18.8 have most of the necessary ingredients between them.

18.10. Proceed by induction on the numbers of applications of composition, primitive recursion, and unbounded minimalization in the recursive definition f, using the previous results in Chapter 18 at the basis and induction steps.

CHAPTER 19

Hints

19.1. $\mathcal{A}$ is a *finite* set of sentences.

19.2. First show that recognizing that a formula has at most v_1 as a free variable is recursive. The rest boils down to checking that substituting a term for a free variable is also recursive, which has already had to be done in the solutions to Problem 17.1.

19.3. Let ψ be the formula (with at most v_1, v_2, and v_3 free) which represents the function f of Problem 19.2 in $\text{Th}(\mathcal{A})$. Then the formula $\forall v_3\,(\psi^{v_2}v_1 \to \varphi^{v_1}_{v_3})$ has only one variable free, namely v_1, and is very close to being the sentence σ needed. To obtain σ you need to substitute S^kO for a suitable k for v_1.

19.4. Try to prove this by contradiction. Observe first that if Σ is recursive, then $\ulcorner\text{Th}(\Sigma)\urcorner$ is representable in $\text{Th}(\mathcal{A})$.

19.5. 1. If Γ were recursive, you could get a contradiction to the Incompleteness Theorem.
2. If Δ were complete, it couldn't also be recursive.
3. Note that $\mathcal{A} \subset \text{Th}(\mathfrak{N})$.

19.6. Modify the formula representing the function $\textsc{Conclusion}_\Sigma$ (defined in Problem 17.2) to get $\text{Con}(\Sigma)$.

19.7. Try to do a proof by contradiction in three stages. First, find a formula φ (with just v_1 free) that represents "n is the code of a sentence which cannot be proven from Σ" and use the Fixed-Point Lemma to find a sentence τ such that $\Sigma \vdash \tau \leftrightarrow \varphi(S^{\ulcorner\tau\urcorner})$. Second, show that if Σ is consistent, then $\Sigma \nvdash \tau$. Third — the *hard* part — show that $\Sigma \vdash \text{Con}(\Sigma) \to \varphi(S^{\ulcorner\tau\urcorner})$. This leads directly to a contradiction.

19.8. Note that $\mathfrak{N} \models \mathcal{A}$.

19.9. If the converse was true, $\mathcal{A}$ would run afoul of the (First) Incompleteness Theorem.

19.10. Suppose, by way of contradiction, that $\ulcorner\text{Th}(\mathfrak{N})\urcorner$ was definable in $\mathfrak{N}$. Now follow the proof of the (First) Incompleteness Theorem as closely as you can.

Bibliography

[1] Jon Barwise (ed.), *Handbook of Mathematical Logic*, North Holland, Amsterdam, 1977, ISBN 0-7204-2285-X.

[2] C.C. Chang and H.J. Keisler, *Model Theory*, third ed., North Holland, Amsterdam, 1990.

[3] Martin Davis, *Computability and Unsolvability*, McGraw-Hill, New York, 1958; Dover, New York, 1982, ISBN 0-486-61471-9.

[4] Martin Davis (ed.), *The Undecidable; Basic Papers On Undecidable Propositions, Unsolvable Problems And Computable Functions*, Raven Press, New York, 1965.

[5] Herbert B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[6] Douglas R. Hofstadter, *Gödel, Escher, Bach*, Random House, New York, 1979, ISBN 0-394-74502-7.

[7] Jerome Malitz, *Introduction to Mathematical Logic*, Springer-Verlag, New York, 1979, ISBN 0-387-90346-1.

[8] Yu.I. Manin, *A Course in Mathematical Logic*, Graduate Texts in Mathematics 53, Springer-Verlag, New York, 1977, ISBN 0-387-90243-0.

[9] Roger Penrose, *The Emperor's New Mind*, Oxford University Press, Oxford, 1989.

[10] T. Rado, *On non-computable functions*, Bell System Tech. J. **41** (1962), 877–884.

[11] Raymond M. Smullyan, *Gödel's Incompleteness Theorems*, Oxford University Press, Oxford, 1992, ISBN 0-19-504672-2.

Index