Introduction to the Theory of Computation
Computability, Complexity,
And the Lambda Calculus
Some Notes for CIS511

Jean Gallier and Jocelyn Quaintance
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
e-mail: jean@seas.upenn.edu

April 9, 2025

Contents

Chapter 1

RAM Programs, Turing Machines, and the Partial Computable Functions

In this chapter we address the fundamental question

What is a computable function?

Nowadays computers are so pervasive that such a question may seem trivial. Isn't the answer that a function is computable if we can write a program computing it!

This is basically the answer so what more can be said that will shed more light on the question?

The first issue is that we should be more careful about the kind of functions that we are considering. Are we restricting ourselves to total functions or are we allowing partial functions that may not be defined for some of their inputs? It turns out that if we consider functions computed by programs, then partial functions must be considered. In fact, we will see that "deciding" whether a program terminates for all inputs is impossible. But what does deciding mean?

To be mathematically precise requires a fair amount of work. One of the key technical points is the ability to design a program U that takes other programs P as input, and then executes P on any input x. In particular, U should be able to take U itself as input!

Of course a compiler does exactly the above task. But fully describing a compiler for a "real" programming language such as JAVA, PYTHON, C++, *etc.* is a complicated and lengthy task. So a simpler (still quite complicated) way to proceed is to develop a toy programming language and a toy computation model (some kind of machine) capable of executing programs written in our toy language. Then we show how programs in this toy language can be coded so that they can be given as input to other programs. Having done this we need to demonstrate that our language has *universal computing power*. This means that we need to show that a "real" program, say written in JAVA, could be translated into a possibly much longer program written in our toy language. This step is typically an act of faith, in the sense that the details that such a translation can be performed are usually not provided.

A way to be precise regarding universal computing power is to define mathematically a family of functions that should be regarded as "obviously computable," and then to show that the functions computed by the programs written either in our toy programming language or in any modern progamming language are members of this mathematically defined family of computable functions. This step is usually technically very involved, because one needs to show that executing the instructions of a program can be mimicked by functions in our family of computable functions. Conversely, we should prove that every computable function in this family is indeed computable by a program written in our toy programming language or in any modern progamming language. Then we will be have the assurance that we have captured the notion of universal computing power.

Remarkably, Herbrand, Gödel, and Kleene defined such a family of functions in 1934-1935. This is a family of numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$ containing a subset of very simple functions called base functions, and this family is the smallest family containing the base functions closed under three operations:

1. Composition

2. Primitive recursion

3. Minimization.

Historically, the first two models of computation are the *λ-calculus* of Church (1935) and the *Turing machine* (1936) of Turing. Kleene proved that the λ-definable functions are exactly the (total) computable functions in the sense of Herbrand–Gödel–Kleene in 1936, and Turing proved that the functions computed by Turing machines are exactly the computable functions in the sense of Herbrand–Gödel–Kleene in 1937. Therefore, the λ-calculus and Turing machines have the same "computing power," and both compute exactly the class of computable functions in the sense of Herbrand–Gödel–Kleene. In those days these results were considered quite surprising because the formalism of the λ-calculus has basically nothing to do with the formalism of Turing machines.

Once again we should be more precise about the kinds of functions that we are dealing with. Until Turing (1936), only numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$ were considered. In order to compute numerical functions in the λ-calculus, Church had to encode the natural numbers as certain λ-terms, which can be viewed as iterators.

Turing assumes that what he calls his *a-machines* (for automatic machines) make use of the symbols 0 and 1 for the purpose of input and output, and if the machine stops, then the output is a *string* of 0s and 1s. Thus a Turing machine can be viewed as computing a function $f\colon (\{0,1\}^*)^m \to \{0,1\}^*$ *on strings*. By allowing a more general alphabet Σ, we see that a Turing machine computes a function $f\colon (\Sigma^*)^m \to \Sigma^*$ on strings over Σ.

At first glance it appears that Turing machines compute a larger class of functions, but this is not so because there exist mutually invertible computable coding functions $C\colon \Sigma^* \to \mathbb{N}$ and decoding functions $D\colon \mathbb{N} \to \Sigma^*$. Using these coding and decoding functions, it suffices to consider numerical functions.

However, Turing machines can also very naturally be viewed as devices for defining computable languages in terms of acceptance and rejection; some kinds of generalized DFA's or NFA's. In this role, it would be very awkward to limit ourselves to sets of natural numbers, although this is possible in theory.

We should also point out that the notion of computable language can be handled in terms of a computation model for functions by considering the characteristic functions of languages. Indeed, a language A is computable (we say decidable) iff its characteristic function χ_A is computable.

The above considerations motivate the definition of the computable functions in the sense of Herbrand–Gödel–Kleene to functions $f\colon (\Sigma^*)^m \to \Sigma^*$ operating *on strings*. However, it is technically simpler to work out all the undecidability results for numerical functions or for subsets of $\mathbb{N}$. Since there is no loss of generally in doing so in view of the computable bijections $C\colon \Sigma^* \to \mathbb{N}$ and $D\colon \mathbb{N} \to \Sigma^*$, we will do so.

Nevertherless, in order to deal with languages, it is important to develop a fair amount of computability theory about functions computing on strings, so we will present another computation model, the *RAM program model*, which computes functions defined on strings. This model was introduced around 1963 (although it was introduced earlier by Post in a different format). It has the advantage of being closer to actual computer architecture, because the RAM model consists of programs operating on a fixed set of registers. This model is equivalent to the Turing machine model, and the translations, although tedious, are not that bad.

The RAM program model also has the technical advantage that coding up a RAM program as a natural number is not that complicated.

The λ-calculus is a very elegant model but it is more abstract than the RAM program model and the Turing machine model so we postpone discussing it until Chapter 3.

Another very interesting computation model particularly well suited to deal with decidable sets of natural numbers is *Diophantine definability*. This model, arising from the work involved in proving that Hilbert's tenth problem is undecidable, will be discussed in Chapter 7.

In the following sections we will define the RAM program model, the Turing machine model, and then argue without proofs (relegated to Chapter 2) that there are algorithms to convert RAM programs into Turing machines, and conversely. Then we define the class of computable functions in the sense of Herbrand–Gödel–Kleene, both for numerical functions (defined on $\mathbb{N}$) and functions defined on strings. This will require explaining what is primitive recursion, which is a restricted form of recursion which guarantees that if it is applied to total functions, then the resulting function is total. Intuitively, primitive recursion corresponds to writing programs that only use for loops (loops where the number of iterations is known ahead of time and fixed).

1.1 Partial Functions and RAM Programs

In this section we define an abstract machine model for computing functions

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_k\}$ is some input alphabet.

Numerical functions $f\colon \mathbb{N}^n \to \mathbb{N}$ can be viewed as functions defined over the one-letter alphabet $\{a_1\}$, using the bijection $m \mapsto a_1^m$.

Since programs are not guaranteed to terminate for all inputs, we are forced to deal with partial functions, so we recall their definition.

Definition 1.1. A binary relation $R \subseteq A \times B$ between two sets A and B is *functional* iff, for all $x \in A$ and $y, z \in B$,

$$(x, y) \in R \quad \text{and} \quad (x, z) \in R \quad \text{implies that} \quad y = z.$$

A *partial function* is a triple $f = \langle A, G, B\rangle$, where A and B are arbitrary sets (possibly empty) and G is a functional relation (possibly empty) between A and B, called the *graph* of f.

Hence, a partial function is a functional relation such that every argument has *at most one image* under f.

The graph of a function f is denoted as $graph(f)$. When no confusion can arise, a function f and its graph are usually identified.

A partial function $f = \langle A, G, B\rangle$ is often denoted as $f\colon A \to B$.

The *domain $dom(f)$* of a partial function $f = \langle A, G, B\rangle$ is the set

$$dom(f) = \{x \in A \mid \exists y \in B,\ (x, y) \in G\}.$$

For every element $x \in dom(f)$, the unique element $y \in B$ such that $(x, y) \in graph(f)$ is denoted as $f(x)$. We say that *$f(x)$ is defined*, also denoted as $f(x)\downarrow$.

If $x \in A$ and $x \notin dom(f)$, we say that *$f(x)$ is undefined*, also denoted as $f(x)\uparrow$.

Intuitively, if a function is partial, it does not return any output for any input not in its domain. This corresponds to an *infinite computation*.

It is important to define precisely the notion of equality of partial functions.

Definition 1.2. Two partial functions $f\colon A \to B$ and $f'\colon A' \to B'$ are equal iff $A = A'$, $B = B'$, and $graph(f) = graph(f')$, which means that for all $a \in A$, either both $f(a)$ and $f'(a)$ are defined and $f(a) = f'(a)$, or both $f(a)$ and $f'(a)$ are undefined.

This definition implies that when we write $f(a) = f'(a)$ for some $a \in A$, we mean that either both $f(a)$ and $f'(a)$ are defined and $f(a) = f'(a)$, or f and f' are both undefined at a (equivalently, $a \notin dom(f) = dom(f')$). There is a slight abuse of notation since $f(a)$ (and $f'(a)$) may not be defined, but this is the customary notation.

Definition 1.3. A partial function $f\colon A \to B$ is a *total function* iff $dom(f) = A$. *It is customary to call a total function simply a function.*

We now define a model of computation know as the *RAM programs* or *Post machines*.

RAM programs are written in a sort of assembly language involving simple instructions manipulating strings stored into registers.

Every RAM program uses a fixed and finite number of *registers* denoted as $R1, \ldots, Rp$, with no limitation on the size of strings held in the registers.

RAM programs can be defined either in flowchart form or in linear form. Since the linear form is more convenient for the purpose of encoding programs as numbers (a process known as Gödel numbering), we focus primarily on RAM programs in linear form. However, the flowchart form tends to be more intuitive and is useful to describe certain constructions (such as primitive recursion and minimization) so we will also describe it.

A RAM program P (in linear form) consists of a finite sequence of *instructions* using a finite number of registers $R1, \ldots, Rp$.

Instructions may optionally be labeled with line numbers denoted as $N1, \ldots, Nq$.

It is neither mandatory to label all instructions, nor to use distinct line numbers! Thus the same line number can be used in more than one line. As we will see later on, this makes it easier to concatenate two different programs without performing a renumbering of line numbers.

Every instruction has *four fields*, not necessarily all used. The main field is the **op-code**.

Definition 1.4. *RAM programs* are constructed from seven types of *instructions* shown below.

(1_j)	N		$\mathtt{add}_j$	Y
(2)	N		$\mathtt{tail}$	Y
(3)	N		$\mathtt{clr}$	Y
(4)	N	Y	$\leftarrow$	X
$(5a)$	N		$\mathtt{jmp}$	$N1a$
$(5b)$	N		$\mathtt{jmp}$	$N1b$
$(6_j a)$	N	Y	$\mathtt{jmp}_j$	$N1a$
$(6_j b)$	N	Y	$\mathtt{jmp}_j$	$N1b$
(7)	N		$\mathtt{continue}$	

1. An instruction of type (1_j) concatenates the letter a_j to the right of the string held by register Y $(1 \leq j \leq k)$. The effect is the assignment
$$Y := Y a_j.$$

2. An instruction of type (2) deletes the leftmost letter of the string held by the register Y. This corresponds to the function *tail*, defined such that
$$\begin{aligned} \mathit{tail}(\epsilon) &= \epsilon, \\ \mathit{tail}(a_j u) &= u \end{aligned}$$
for all $u \in \Sigma^*$. The effect is the assignment
$$Y := \mathit{tail}(Y).$$

3. An instruction of type (3) clears register Y, i.e., sets its value to the empty string ϵ. The effect is the assignment
$$Y := \epsilon.$$

4. An instruction of type (4) assigns the value of register X to register Y. The effect is the assignment
$$Y := X.$$

5. An instruction of type (5a) or (5b) is an unconditional jump.

   The effect of (5a) is to jump to the closest line number $N1$ occurring above the instruction being executed, and the effect of (5b) is to jump to the closest line number $N1$ occurring below the instruction being executed.

6. An instruction of type $(6_j a)$ or $(6_j b)$ is a conditional jump. Let *head* be the function defined as follows:
$$\begin{aligned} \mathit{head}(\epsilon) &= \epsilon, \\ \mathit{head}(a_j u) &= a_j \end{aligned}$$
for all $u \in \Sigma^*$. The effect of $(6_j a)$ is to jump to the closest line number $N1$ occurring above the instruction being executed iff $\mathit{head}(Y) = a_j$, else to execute the next instruction (the one immediately following the instruction being executed).

   The effect of $(6_j b)$ is to jump to the closest line number $N1$ occurring below the instruction being executed iff $\mathit{head}(Y) = a_j$, else to execute the next instruction.

   When computing over $\mathbb{N}$, instructions of type $(6_j a)$ or $(6_j b)$ jump to the closest $N1$ above or below iff Y is nonnull.

7. An instruction of type (7) is a no-op, i.e., the registers are unaffected. If there is a next instruction, then it is executed, else the program stops.

When computing over $\mathbb{N}$, which corresponds to the case where $\Sigma = \{a_1\}$, an instruction of type (1) computes the successor function S (or **Succ**) given by $S(n) = n+1$, an instruction of type (2) computes the predecessor function *pred* given by $pred(n+1) = n$ and $pred(0) = 0$, and an instruction of type (3) computes the zero function Z given by $Z(n) = 0$.

Obviously, a program is syntactically correct only if certain conditions hold.

Definition 1.5. A *RAM program* P is a finite sequence of instructions as in Definition 1.4, and satisfying the following conditions:

(1) For every jump instruction (conditional or not), the line number to be jumped to must exist in P.

(2) The last instruction of a RAM program is a `continue`.

The reason for allowing multiple occurences of line numbers is to make it easier to concatenate programs without having to perform a renaming of line numbers.

The technical choice of jumping to the closest address $N1$ above or below comes from the fact that it is easy to search up or down using primitive recursion, as we will see later on.

For the purpose of computing a function $f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$ using a RAM program P, we assume that P has at least n registers called *input registers*, and that these registers $R1, \ldots, Rn$ are initialized with the input values of the function f. We also assume that *the output is returned in register $R1$*.

Example 1.1. The following RAM program concatenates two strings x_1 and x_2 held in registers $R1$ and $R2$. Since $\Sigma = \{a, b\}$, for more clarity, we wrote $\texttt{jmp}_a$ instead of $\texttt{jmp}_1$, $\texttt{jmp}_b$ instead of $\texttt{jmp}_2$, $\texttt{add}_a$ instead of $\texttt{add}_1$, and $\texttt{add}_b$ instead of $\texttt{add}_2$.

```
        R3    ←       R1
        R4    ←       R2
N0      R4    jmp_a   N1b
        R4    jmp_b   N2b
              jmp     N3b
N1            add_a   R3
              tail    R4
              jmp     N0a
N2            add_b   R3
              tail    R4
              jmp     N0a
N3      R1    ←       R3
           continue
```

The instructions of a RAM program in flowchart form are shown in Figure 1.1.

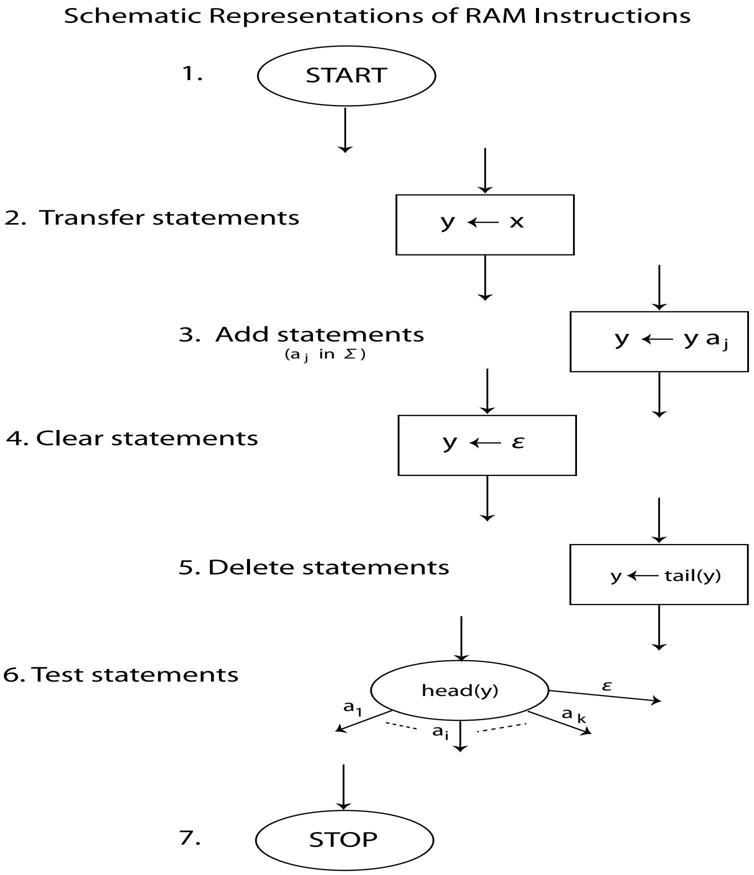


Figure 1.1: RAM instructions in flowchart form.

They are all self-explanatory except perhaps the test statements which behave as follows. If the leftmost symbol $head(y)$ is the letter a_i, then follow the arrow labeled a_i (to the instruction to be executed next). Otherwise $y = \epsilon$ and then follow the arrow labeled ϵ.

Remark: The instructions of a RAM program in flowchart form are very similar to the instructions of the Post machines discussed in Manna [30]. However, Post machines use a single register. Nevertheless, it can be shown that the two models are equivalent.

Definition 1.6. A *RAM flowchart program* is a directed graph obtained by interconnecting statements in such a way that:

(1) There is a single START.

(2) There is a single STOP.

(3) Every entry point of a statement is connected to an exit point of some statement and every exit point of a statement is connected to the entry point of some statement.

As in the case of a RAM program in linear form, a RAM program in flowchart form is assumed to have prescribed input variables. A flowchart form representation of the RAM program of Example 1.1 is shown in Figure 1.2.

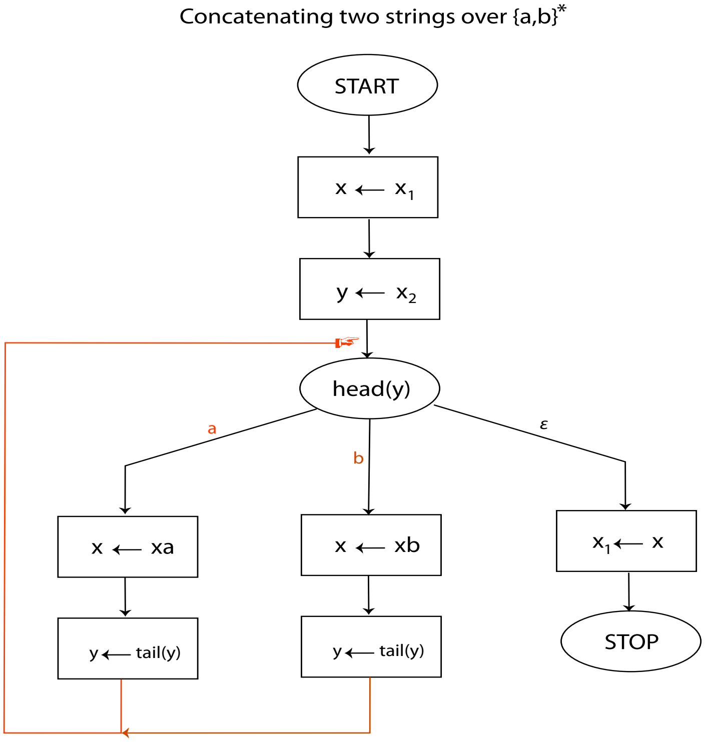


Figure 1.2: A RAM program in flowchart form for computing concatenation.

Remark: The reader may have noticed that the definition of a RAM program, either in flowchart form or linear form, does not exclude undesirable programs such as disconnected programs consisting of several connected components. We could fix the definitions to avoid such pathological cases, but they are exceptional and we will not go into such trouble now. The reader is invited to think about pathological cases that should be ruled out and ways of fixing the definitions to avoid them.

Definition 1.7. A RAM program P *computes the partial function* $\varphi\colon (\Sigma^*)^n \to \Sigma^*$ if the following conditions hold: for every input $(x_1, \ldots, x_n) \in (\Sigma^*)^n$, having initialized the input registers $R1, \ldots, Rn$ with $x_1, \ldots, x_n$, the program eventually halts iff $\varphi(x_1, \ldots, x_n)$ is defined, and if and when P halts, the value of $R1$ is equal to $\varphi(x_1, \ldots, x_n)$. A partial function φ is *RAM-computable* iff it is computed by some RAM program.

Example 1.2. The following program computes the *erase function E* defined such that

$$E(u) = \epsilon$$

for all $u \in \Sigma^*$:

```
clr         $R1$
continue
```

Example 1.3. The following program computes the *jth successor function S_j* defined such that

$$S_j(u) = ua_j$$

for all $u \in \Sigma^*$:

```
add$_j$     $R1$
continue
```

Example 1.4. The following program (with n input variables) computes the *projection function P_i^n* defined such that

$$P_i^n(u_1, \ldots, u_n) = u_i,$$

where $n \geq 1$, and $1 \leq i \leq n$:

```
$R1$  $\leftarrow$       $Ri$
      continue
```

Note that P_1^1 is the identity function.

The equivalence of the flowchart form and the linear form of RAM programs is straightforward. Translating a program in linear form to the flowchart form is almost immediate and is left as an exercise. In the other direction, first we assign distinct labels to all the statements in the flowchart except START. The only translation which is not immediately obvious is the case of a test statement. If the target labels of the arrows labeled $a_1, \ldots, a_k, \epsilon$ are $N1, \ldots, Nk, N(k+1)$, we create the following piece of code

```
$Y$    jmp$_1$   $N1c$
       ⋮
$Y$    jmp$_k$   $Nkc$
$Y$    jmp     $N(k+1)c$,
```

where c is a or b depending on the location of Ni in the linear RAM program. Extra unconditional jumps may also be needed to mimic the flow of control of the program in flowchart form. The details are left as an exercise.

Having a programming language, we would like to know how powerful it is, that is, we would like to know what kind of functions are RAM-computable. At first glance, it seems that RAM programs don't do much, but this is not so. Indeed, we will see shortly that the class of RAM-computable functions is quite extensive.

One way of getting new programs from previous ones is via composition. Another one is by primitive recursion. We will investigate these constructions after introducing another model of computation, *Turing machines*.

Remarkably, the classes of (partial) functions computed by RAM programs and by Turing machines are identical. This is the class of *partial computable functions* in the sense of Herbrand–Gödel–Kleene, also called *partial recursive functions*, a term which is now considered old-fashion. We will present the definition of the so-called *μ-recursive functions* (due to Kleene).

The following proposition will be needed to simplify the encoding of RAM programs as numbers.

Proposition 1.1. *Every RAM program can be converted to an equivalent program only using the following type of instructions.*

$$\begin{array}{lllll}
(1_j) & N & & \mathtt{add}_j & Y \\
(2) & N & & \mathtt{tail} & Y \\
(6_j a) & N & Y & \mathtt{jmp}_j & N1a \\
(6_j b) & N & Y & \mathtt{jmp}_j & N1b \\
(7) & N & & \mathtt{continue} &
\end{array}$$

The proof is fairly simple. For example, instructions of the form

$$Ri \leftarrow Rj$$

can be eliminated by transferring the contents of Rj into an auxiliary register Rk, and then by transferring the contents of Rk into Ri and Rj.

1.2 Definition of a Turing Machine

We define a Turing machine model for computing functions

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_k\}$ is some input alphabet. In this section, since we are primarily interested in computing functions we only consider deterministic Turing machines.

There are many variants of the Turing machine model. The main decision that needs to be made has to do with the kind of tape used by the machine. We opt for a single *finite tape* that is both an input and a storage mechanism. This tape can be viewed as a string over *tape alphabet* Γ such that $\Sigma \subseteq \Gamma$. There is a read/write head pointing to some symbol on the tape, symbols on the tape can be overwritten, and the read/write head can move one symbol to the left or one symbol to the right, also causing a state transition. When the write/read head attempts to move past the rightmost or the leftmost symbol on the tape, the tape is allowed to grow. To accomodate such a move, the tape alphabet contains some special symbol $B \notin \Sigma$, the *blank*, and this symbol is added to the tape as the new leftmost or rightmost symbol on the tape.

A common variant uses a tape which is infinite at both ends, but only has finitely many symbols not equal to B, so effectively it is equivalent to a finite tape allowed to grow at either ends. Another variant uses a semi-infinite tape infinite to the right, but with a left end. We find this model cumbersome because it requires shifting right the entire tape when a left move is attempted from the left end of the tape.

Another decision that needs to be made is the format of the instructions. Does an instruction cause *both* a state transition *and* a symbol overwrite, or do we have separate instructions for a state transition and a symbol overwrite. In the first case, an instruction can be specified as a quintuple, and in the second case by a quadruple. We opt for quintuples. Here is our definition.

Definition 1.8. A (deterministic) *Turing machine* (or *TM*) M is a sextuple $M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$, where

- K is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *tape alphabet*, s.t. $\Sigma \subseteq \Gamma$, $K \cap \Gamma = \emptyset$, and with blank $B \notin \Sigma$;
- $q_0 \in K$ is the *start state* (or *initial state*);
- δ is the *transition function*, a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

  such that for all $(p, a) \in K \times \Gamma$, there is at most one triple $(b, m, q) \in \Gamma \times \{L, R\} \times K$ such that $(p, a, b, m, q) \in \delta$.

A quintuple $(p, a, b, m, q) \in \delta$ is called an *instruction*. It is also denoted as

$$p, a \to b, m, q.$$

The effect of an instruction is to switch from state p to state q, overwrite the symbol currently scanned a with b, and move the read/write head either left or right, according to m.

Example 1.5. Here is an example of a Turing machine specified by

$K = \{q_0, q_1, q_2, q_3\}$; $\Sigma = \{a, b\}$; $\Gamma = \{a, b, B\}$.

The instructions in δ are

$$\begin{aligned}
q_0, B &\rightarrow B, R, q_3,\\
q_0, a &\rightarrow b, R, q_1,\\
q_0, b &\rightarrow a, R, q_1,\\
q_1, a &\rightarrow b, R, q_1,\\
q_1, b &\rightarrow a, R, q_1,\\
q_1, B &\rightarrow B, L, q_2,\\
q_2, a &\rightarrow a, L, q_2,\\
q_2, b &\rightarrow b, L, q_2,\\
q_2, B &\rightarrow B, R, q_3.
\end{aligned}$$

1.3 Computations of Turing Machines

To explain how a Turing machine works, we describe its action on *instantaneous descriptions*. We take advantage of the fact that $K \cap \Gamma = \emptyset$ to define instantaneous descriptions.

Definition 1.9. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

an *instantaneous description* (for short an *ID*) is a (nonempty) string in $\Gamma^* K \Gamma^+$, that is, a string of the form

$$upav,$$

where $u, v \in \Gamma^*$, $p \in K$, and $a \in \Gamma$.

The intuition is that an ID $upav$ describes a snapshot of a TM in the current state p, whose tape contains the string uav, and with the read/write head pointing to the symbol a. Thus, in $upav$, the state p is just to the left of the symbol presently scanned by the read/write head.

We explain how a TM works by showing how it acts on ID's.

Definition 1.10. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

the *yield relation (or compute relation)* $\vdash$ is a binary relation defined on the set of ID's as follows. For any two ID's ID_1 and ID_2, we have $ID_1 \vdash ID_2$ iff either

(1) $(p, a, b, R, q) \in \delta$, and either

(a) $ID_1 = upacv$, $c \in \Gamma$, and $ID_2 = ubqcv$, or

(b) $ID_1 = upa$ and $ID_2 = ubqB$;

or

(2) $(p, a, b, L, q) \in \delta$, and either

(a) $ID_1 = ucpav$, $c \in \Gamma$, and $ID_2 = uqcbv$, or

(b) $ID_1 = pav$ and $ID_2 = qBbv$.

See Figure 1.3.

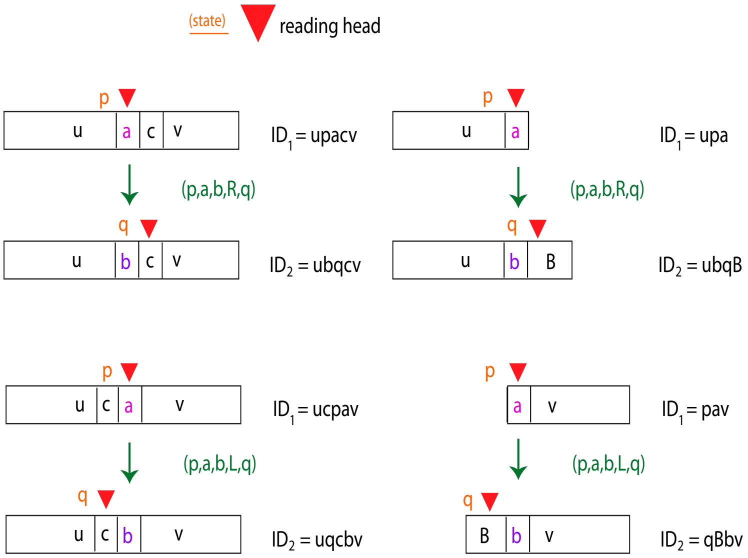


Figure 1.3: Moves of a Turing machine.

Note how the tape is extended by one blank after the rightmost symbol in Case (1)(b), and by one blank before the leftmost symbol in Case (2)(b).

As usual, we let $\vdash^+$ denote the transitive closure of $\vdash$, and we let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$. We can now explain how a Turing machine computes a partial function

$$f \colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*.$$

Since we allow functions taking $n \geq 1$ input strings, we assume that Γ contains the special delimiter **,** not in Σ, used to separate the various input strings.

It is convenient to assume that a Turing machine "cleans up" its tape when it halts before returning its output. What this means is that when the Turing machine halts, the output should be clearly identifiable, so all symbols not in $\Sigma \cup \{B\}$ that may have been used during the computation must be erased. Thus when the TM stops the tape must consist of a string $w \in \Sigma^*$ possibly surrounded by blanks (the symbol B). Actually, if the output is ϵ, the tape must contain a nonempty string of blanks. To achieve this technically, we define proper ID's.

Definition 1.11. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

where Γ contains some delimiter **,** not in Σ in addition to the blank B, a *starting ID* is of the form

$$q_0 w_1 , w_2 , \ldots , w_n$$

where $w_1, \ldots, w_n \in \Sigma^*$ and $n \geq 2$, or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$.

A *blocking (or halting) ID* is an ID $upav$ such that there are no instructions $(p, a, b, m, q) \in \delta$ for any $(b, m, q) \in \Gamma \times \{L, R\} \times K$.

A *proper ID* is a halting ID of the form

$$B^h p w B^l,$$

where $w \in \Sigma^*$, and $h, l \geq 0$ (with $l \geq 1$ when $w = \epsilon$).

Computation sequences are defined as follows.

Definition 1.12. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

a *computation sequence (or computation)* is a finite or infinite sequence of ID's

$$ID_0, ID_1, \ldots, ID_i, ID_{i+1}, \ldots,$$

such that $ID_i \vdash ID_{i+1}$ for all $i \geq 0$.

A computation sequence *halts* iff it is a finite sequence of ID's, so that

$$ID_0 \vdash^* ID_n,$$

and ID_n is a halting ID.

A computation sequence *diverges* if it is an infinite sequence of ID's.

We now explain how a Turing machine computes a partial function.

Definition 1.13. A Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$$

computes the partial function

$$f \colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$$

iff the following conditions hold:

(1) For every $w_1, \ldots, w_n \in \Sigma^*$, given the starting ID

$$ID_0 = q_0 w_1{,}w_2, \ldots {,}w_n$$

or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$, the computation sequence of M from ID_0 halts in a proper ID iff $f(w_1, \ldots, w_n)$ is defined.

(2) If $f(w_1, \ldots, w_n)$ is defined, then M halts in a proper ID of the form

$$ID_n = B^h p f(w_1, \ldots, w_n) B^l,$$

which means that it computes the right value.

A function f (over Σ^*) is *Turing computable* iff it is computed by some Turing machine M.

Note that by (1), the TM M may halt in an improper ID, in which case $f(w_1, \ldots, w_n)$ must be undefined. This corresponds to the fact that *we only accept to retrieve the output of a computation if the TM has cleaned up its tape*, *i.e.*, produced a proper ID. In particular, intermediate calculations have to be erased before halting.

Example 1.6. Consider the Turing machine of Example 1.5 specified by $K = \{q_0, q_1, q_2, q_3\}$; $\Sigma = \{a, b\}$; $\Gamma = \{a, b, B\}$.

The instructions in δ are

$$\begin{aligned}
q_0, B &\to B, R, q_3,\\
q_0, a &\to b, R, q_1,\\
q_0, b &\to a, R, q_1,\\
q_1, a &\to b, R, q_1,\\
q_1, b &\to a, R, q_1,\\
q_1, B &\to B, L, q_2,\\
q_2, a &\to a, L, q_2,\\
q_2, b &\to b, L, q_2,\\
q_2, B &\to B, R, q_3.
\end{aligned}$$

The reader can easily verify that this machine exchanges the a's and b's in a string. For example, on input $w = aaababb$, the output is $bbbabaa$. The computation is given by the following sequence of ID's.

$$\begin{aligned}
&q_0\, aaababb \vdash b\, q_1\, aababb \vdash bb\, q_1\, ababb \vdash bbb\, q_1\, babb \vdash bbba\, q_1\, abb \vdash bbbab\, q_1\, bb\\
&\vdash bbbaba\, q_1\, b \vdash bbbabaa\, q_1\, B \vdash bbbaba\, q_2\, aB \vdash bbbab\, q_2\, aaB \vdash bbba\, q_2\, baaB\\
&\vdash bbb\, q_2\, abaaB \vdash bb\, q_2\, babaaB \vdash b\, q_2\, bbabaaB \vdash q_2\, bbbabaaB \vdash q_2\, BbbbabaaB\\
&\vdash B\, q_3\, bbbabaaB.
\end{aligned}$$

The last ID $B\, q_3\, bbbabaaB$ is a proper ID and the output is $bbbabaa$.

1.4 Equivalence of RAM Programs And Turing Machines

Turing machines can simulate RAM programs, and as a result, we have the following theorem.

Theorem 1.2. *Every RAM-computable function is Turing-computable. Furthermore, given a RAM program P, we can effectively construct a Turing machine M computing the same function.*

The idea of the proof is to represent the contents of the registers $R1, \ldots Rp$ on the Turing machine tape by the string

$$\#r1\#r2\# \cdots \#rp\#,$$

where # is a special marker and ri represents the string held by Ri. We also use Proposition 1.1 to reduce the number of instructions to be dealt with.

The Turing machine M is built of blocks, each block simulating the effect of some instruction of the program P. The details are a bit tedious, and can be found in Section 2.1 or in Machtey and Young [29].

RAM programs can also simulate Turing machines.

Theorem 1.3. *Every Turing-computable function is RAM-computable. Furthermore, given a Turing machine M, one can effectively construct a RAM program P computing the same function.*

The idea of the proof is to design a RAM program containing an encoding of the current ID of the Turing machine M in register $R1$, and to use other registers $R2, R3$ to simulate the effect of executing an instruction of M by updating the ID of M in $R1$.

The details are tedious and can be found in Section 2.2.

Another proof can be obtained by proving that the class of Turing computable functions coincides with the class of *partial computable functions* (formerly called *partial recursive functions*), to be defined shortly. Indeed, it turns out that both RAM programs and Turing machines compute precisely the class of partial recursive functions. For this, we will need to define the *primitive recursive functions*.

Informally, a primitive recursive function is a *total* recursive function that can be computed using only **for** loops, that is, loops in which the number of iterations is fixed (unlike a **while** loop). A formal definition of the primitive functions is given in Section 1.7. For the time being we make the following provisional definition.

Definition 1.14. Let $\Sigma = \{a_1, \ldots, a_k\}$. The class of *partial computable functions*, also called *partial recursive functions*, is the class of partial functions (over Σ^*) that can be computed by RAM programs (or equivalently by Turing machines).

The class of *computable functions*, also called *recursive functions*, is the subset of the class of partial computable functions consisting of functions defined for every input (i.e., total functions).

Turing machines can also be used as acceptors to define languages so we introduce the basic relevant definitions. A more detailed study of these languages will be provided in Chapter 5.

1.5 Listable Languages and Computable Languages

We define the computably enumerable languages, also called listable languages, and the computable languages. The old-fashion terminology for listable languages is recursively enumerable languages, and for computable languages is recursive languages.

When operating as an acceptor, a Turing machine takes a single string as input and either goes on forever or halts with the answer "accept" or "reject." One way to deal with acceptance or rejection is to assume that the TM has a set of final states. Another way more consistent with our view that machines compute functions is to assume that the TM's under consideration have a tape alphabet containing the special symbols 0 and 1. Then acceptance is signaled by the output 1, and rejection is signaled by the output 0.

Note that with our convention that in order to produce an output a TM must halt in a proper ID, the TM must erase the tape before outputing 0 or 1.

Definition 1.15. Let $\Sigma = \{a_1, \ldots, a_k\}$. A language $L \subseteq \Sigma^*$ is *(Turing) listable* or *(Turing) computably enumerable (for short, a c.e. set)* (or *recursively enumerable (for short, a r.e. set)*) iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, either M halts in a proper ID with the output 0 or it runs forever.

A language $L \subseteq \Sigma^*$ is *(Turing) computable* (or *recursive*) iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, M halts in a proper ID with the output 0.

Thus, given a computably enumerable language L, for some $w \notin L$, it is possible that a TM accepting L runs forever on input w. On the other hand, for a computable (recursive) language L, a TM accepting L always halts in a proper ID.

When dealing with languages, it is often useful to consider *nondeterministic Turing machines*. Such machines are defined just like deterministic Turing machines, except that their transition function δ is just a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

with no particular extra condition.

It can be shown that every nondeterministic Turing machine can be simulated by a deterministic Turing machine, and thus, nondeterministic Turing machines also accept the class of c.e. sets. This is a very tedious simulation, and very few books actually provide all the details!

It can be shown that a computably enumerable language is the *range* of some computable (recursive) function; see Section 5.4. It can also be shown that a language L is computable (recursive) iff both L and its complement are computably enumerable; see Section 5.4. There are computably enumerable languages that are not computable (recursive); see Section 5.4.

1.6 A Simple Function Not Known to be Computable

The "$3n+1$ problem" proposed by Collatz around 1937 is the following:

Given any positive integer $n \geq 1$, construct the sequence $c_i(n)$ as follows starting with $i = 1$:

$$\begin{aligned} c_1(n) &= n \\ c_{i+1}(n) &= \begin{cases} c_i(n)/2, & \text{if } c_i(n) \text{ is even} \\ 3c_i(n)+1, & \text{if } c_i(n) \text{ is odd.} \end{cases} \end{aligned}$$

Observe that for $n = 1$, we get the infinite periodic sequence

$$1 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1 \Longrightarrow \cdots,$$

so we may assume that we stop the first time that the sequence $c_i(n)$ reaches the value 1 (if it actually does). Such an index i is called the *stopping time* of the sequence. And this is the problem:

Conjecture (Collatz):

For any starting integer value $n \geq 1$, the sequence $(c_i(n))$ always reaches 1.

Starting with $n = 3$, we get the sequence

$$3 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 5$, we get the sequence

$$5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 6$, we get the sequence

$$6 \Longrightarrow 3 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 7$, we get the sequence

$$\begin{aligned}7 &\Longrightarrow 22 \Longrightarrow 11 \Longrightarrow 34 \Longrightarrow 17 \Longrightarrow 52 \Longrightarrow 26 \Longrightarrow 13 \Longrightarrow 40\\ &\Longrightarrow 20 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.\end{aligned}$$

One might be surprised to find that for $n = 27$, it takes 111 steps to reach 1, and for $n = 97$, it takes 118 steps. We computed the stopping times for n up to 10^7 and found that the largest stopping time, 686 (685 steps) is obtained for $n = 8400511$. The terms of this sequence reach values over 1.5×10^{11}. The graph of the sequence $c(8400511)$ is shown in Figure 1.4.

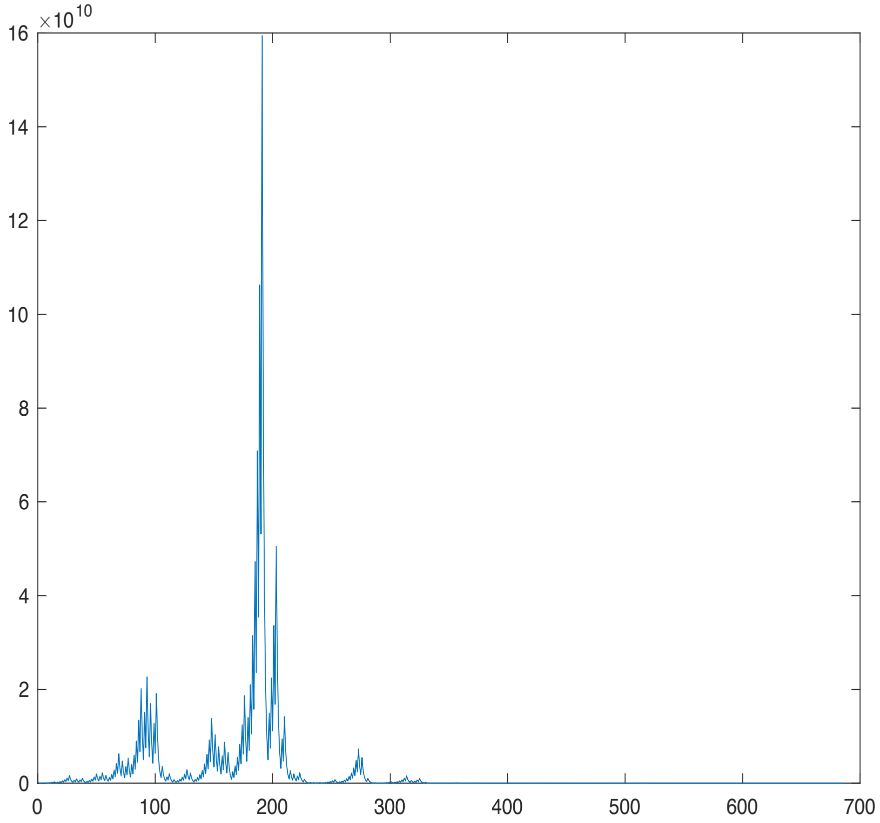


Figure 1.4: Graph of the sequence for $n = 8400511$.

We can define the partial computable function C (with positive integer inputs) defined by

$$C(n) = \text{the smallest } i \text{ such that } c_i(n) = 1 \text{ if it exists.}$$

Then the Collatz conjecture is equivalent to asserting that the function C is (total) computable. The graph of the function C for $1 \leq n \leq 10^7$ is shown in Figure 1.5.

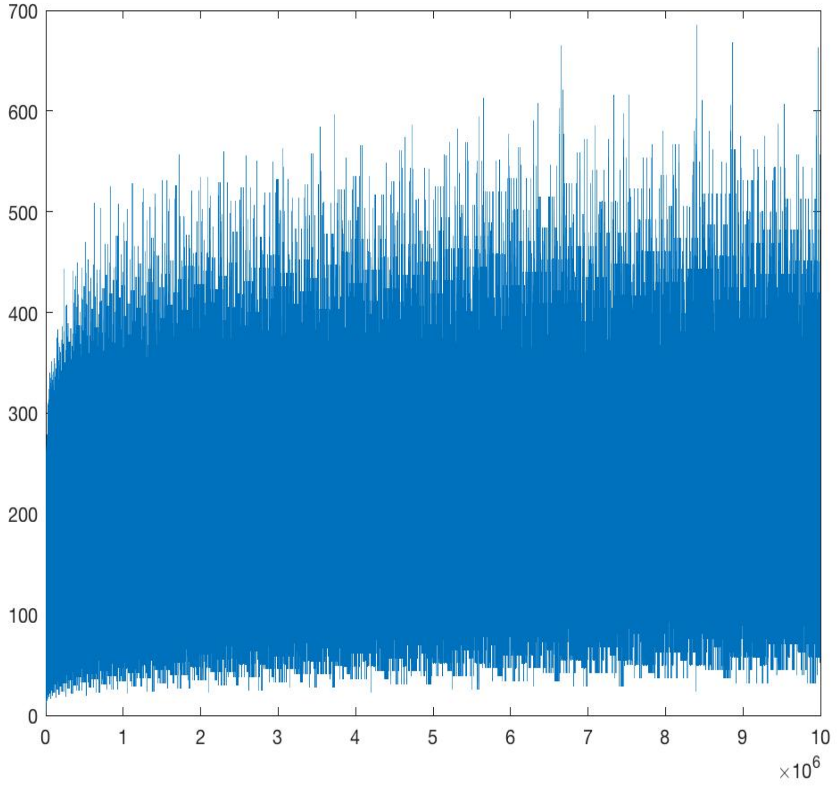


Figure 1.5: Graph of the function C for $1 \leq n \leq 10^7$.

So far, the conjecture remains open. It has been checked by computer for all integers less than or equal to 87×2^{60}.

We now return to the computability of functions. Our goal is to define the partial computable functions in the sense of Herbrand–Gödel–Kleene. This class of functions is defined from some base functions in terms of three closure operations:

1. Composition
2. Primitive recursion
3. Minimization.

The first two operations preserve the property of a function to be total, and this subclass of total computable functions called *primitive recursive functions* plays an important technical role.

1.7 The Primitive Recursive Functions

Historically the primitive recursive functions were defined for numerical functions (computing on the natural numbers). Since one of our goals is to show that the RAM-computable functions are partial recursive, we define the primitive recursive functions as functions $f\colon (\Sigma^*)^m \to \Sigma^*$, where $\Sigma = \{a_1, \ldots, a_k\}$ is a finite alphabet. As usual, by assuming that $\Sigma = \{a_1\}$, we can deal with numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$.

The class of primitive recursive functions is defined in terms of base functions and two closure operations.

Definition 1.16. Let $\Sigma = \{a_1, \ldots, a_k\}$. The *base functions* over Σ are the following functions:

(1) The *erase function E*, defined such that $E(w) = \epsilon$, for all $w \in \Sigma^*$;

(2) For every j, $1 \leq j \leq k$, the *j-successor function S_j*, defined such that $S_j(w) = wa_j$, for all $w \in \Sigma^*$;

(3) The *projection functions P_i^n*, defined such that

$$P_i^n(w_1, \ldots, w_n) = w_i,$$

for every $n \geq 1$, every i, $1 \leq i \leq n$, and for all $w_1, \ldots, w_n \in \Sigma^*$.

Note that P_1^1 is the identity function on Σ^*. Projection functions can be used to permute, duplicate, or drop the arguments of another function.

Definition 1.17. In the special case where we are only considering numerical functions ($\Sigma = \{a_1\}$), the function $E\colon \mathbb{N} \to \mathbb{N}$ is the *zero function* given by $E(n) = 0$ for all $n \in \mathbb{N}$, and it is often denoted by Z. There is a single successor function $S_{a_1}\colon \mathbb{N} \to \mathbb{N}$ usually denoted S (or **Succ**) given by $S(n) = n + 1$ for all $n \in \mathbb{N}$.

Even though in this section we are primarily interested in total functions, later on, the same closure operations will be applied to partial functions so we state the definition of the closure operations in the more general case of partial functions. The first closure operation is (extended) composition.

Definition 1.18. Let $\Sigma = \{a_1, \ldots, a_k\}$. For any partial or total function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

and any $m \geq 1$ partial or total functions

$$h_i\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*, \quad n \geq 1,$$

the *composition of g and the h_i* is the partial function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

denoted as $g \circ (h_1, \ldots, h_m)$, such that

$$f(w_1, \ldots, w_n) = g(h_1(w_1, \ldots, w_n), \ldots, h_m(w_1, \ldots, w_n))$$

for all $w_1, \ldots, w_n \in \Sigma^*$. If g and all the h_i are total functions, then $g \circ (h_1, \ldots, h_m)$ is obviously a total function. But if g or any of the h_i is a partial function, then the value *$(g\circ(h_1, \ldots, h_m))(w_1, \ldots, w_n)$ is defined if and only if all the values $h_i(w_1, \ldots, w_n)$ are defined for $i = 1, \ldots, m$, and $g(h_1(w_1, \ldots, w_n), \ldots, h_m(w_1, \ldots, w_n))$ is defined.*

Thus even if g "ignores" some of its inputs, in computing $g(h_1(w_1, \ldots, w_n), \ldots, h_m(w_1, \ldots, w_n))$, all arguments $h_i(w_1, \ldots, w_n)$ *must* be evaluated.

As an example of a composition, $f = g \circ (P_2^2, P_1^2)$ is such that

$$f(w_1, w_2) = g(P_2^2(w_1, w_2), P_1^2(w_1, w_2)) = g(w_2, w_1).$$

The second closure operation is *primitive recursion*. First we define primitive recursion for numerical functions because it is simpler.

Definition 1.19. Given any two partial or total functions $g\colon \mathbb{N}^{m-1} \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+1} \to \mathbb{N}$ $(m \geq 2)$, the partial or total function $f\colon \mathbb{N}^m \to \mathbb{N}$ is defined by *primitive recursion from g and h* if f is given by

$$\begin{aligned} f(0, x_2, \ldots, x_m) &= g(x_2, \ldots, x_m) \\ f(n+1, x_2, \ldots, x_m) &= h(n, f(n, x_2, \ldots, x_m), x_2, \ldots, x_m), \end{aligned}$$

for all $n, x_2, \ldots, x_m \in \mathbb{N}$. When $m = 1$, we have

$$\begin{aligned} f(0) &= b \\ f(n+1) &= h(n, f(n)), \quad \text{for all } n \in \mathbb{N}, \end{aligned}$$

for some fixed natural number $b \in \mathbb{N}$.

If g and h are total functions, it is easy to show that f is also a total function. If g or h is partial, obviously $f(0, x_2, \ldots, x_m)$ is defined iff $g(x_2, \ldots, x_m)$ is defined, and $f(n+1, x_2, \ldots, x_m)$ is defined iff $f(n, x_2, \ldots, x_m)$ is defined and $h(n, f(n, x_2, \ldots, x_m), x_2, \ldots, x_m)$ is defined.

Definition 1.19 is quite a straightjacket in the sense that $n+1$ must be the first argument of f, and the definition only applies if h has $m + 1$ arguments, but in practice a "natural" definition often ignores the argument n and some of the arguments $x_2, \ldots, x_m$. This is where the projection functions come into play to drop, duplicate, or permute arguments.

Example 1.7. For example, a “natural” definition of the predecessor function *pred* is

$$\begin{aligned} pred(0) &= 0 \\ pred(m+1) &= m, \end{aligned}$$

but this is not a legal primitive recursive definition. To make it a legal primitive recursive definition we need the function $h = P_1^2$, and a legal primitive recursive definition for *pred* is

$$\begin{aligned} pred(0) &= 0 \\ pred(m+1) &= P_1^2(m, pred(m)). \end{aligned}$$

Example 1.8. Addition, multiplication, exponentiation, and super-exponentiation, can be defined by primitive recursion as follows (being a bit loose, for supexp we should use some projections ...):

$$\begin{aligned} add(0,n) &= P_1^1(n) = n \\ add(m+1,n) &= S \circ P_2^3(m, add(m,n), n) \\ &= S(add(m,n)) \\ mult(0,n) &= E(n) = 0 \\ mult(m+1,n) &= add \circ (P_2^3, P_3^3)(m, mult(m,n), n) \\ &= add(mult(m,n), n) \\ rexp(0,n) &= S \circ E(n) = 1 \\ rexp(m+1,n) &= mult \circ (P_2^3, P_3^3)(m, rexp(m,n), n) \\ exp(m,n) &= rexp \circ (P_2^2, P_1^2)(m,n) \\ supexp(0,n) &= 1 \\ supexp(m+1,n) &= exp(n, supexp(m,n)). \end{aligned}$$

We usually write $m+n$ for $add(m,n)$, $m*n$ or even mn for $mult(m,n)$, and m^n for $exp(m,n)$.

Example 1.9. The recursive definition of m^n is $m^{(n+1)} = m^n * m$, which corresponds to

$$exp(m, n+1) = mult(exp(m,n), m).$$

Unfortunately, the recursion is on the second argument n, so we have to create the auxiliary function *rexp* given by

$$rexp(m,n) = n^m,$$

write the primitive recusive definition of *rexp* in m, and then

$$exp(m,n) = rexp(n,m) = rexp \circ (P_2^2, P_1^2)(m,n).$$

Example 1.10. There is a minus operation on $\mathbb{N}$ named *monus*. This operation denoted by $\dot{-}$ is defined by

$$m \mathbin{\dot{-}} n = \begin{cases} m - n, & \text{if } m \geq n \\ 0, & \text{if } m < n. \end{cases}$$

Then *monus* is defined by

$$\begin{aligned} m \mathbin{\dot{-}} 0 &= m \\ m \mathbin{\dot{-}} (n+1) &= \mathit{pred}(m \mathbin{\dot{-}} n), \end{aligned}$$

except that the above is not a legal primitive recursion. For one thing, recursion should be performed on m, not n. We can define *rmonus* as

$$\mathit{rmonus}(n, m) = m \mathbin{\dot{-}} n,$$

and then $m \mathbin{\dot{-}} n = (\mathit{rmonus} \circ (P_2^2, P_1^2))(m, n)$, and

$$\begin{aligned} \mathit{rmonus}(0 \mathbin{\dot{-}} m) &= P_1^1(m) \\ \mathit{rmonus}(n+1, m) &= \mathit{pred} \circ P_2^2(n, \mathit{rmonus}(n, m)). \end{aligned}$$

Example 1.11. The following functions are also primitive recursive:

$$sg(n) = \begin{cases} 1, & \text{if } n > 0 \\ 0, & \text{if } n = 0, \end{cases}$$

$$\overline{sg}(n) = \begin{cases} 0, & \text{if } n > 0 \\ 1, & \text{if } n = 0, \end{cases}$$

as well as

$$\mathit{abs}(m, n) = |m - n| = m \mathbin{\dot{-}} n + n \mathbin{\dot{-}} m,$$

and

$$\mathit{eq}(m, n) = \begin{cases} 1, & \text{if } m = n \\ 0, & \text{if } m \neq n. \end{cases}$$

Indeed

$$\begin{aligned} sg(0) &= 0 \\ sg(n+1) &= S \circ E \circ P_1^2(n, sg(n)) \\ \overline{sg}(n) &= S(E(n)) \mathbin{\dot{-}} sg(n) = 1 \mathbin{\dot{-}} sg(n), \end{aligned}$$

and

$$\mathit{eq}(m, n) = \overline{sg}(|m - n|).$$

Example 1.12. Finally, the function

$$cond(m,n,p,q) = \begin{cases} p, & \text{if } m = n \\ q, & \text{if } m \neq n, \end{cases}$$

is primitive recursive, since

$$cond(m,n,p,q) = eq(m,n) * p + \overline{sg}(eq(m,n)) * q.$$

Example 1.13. We can also design more general version of *cond*. For example, define $compare_{\leq}$ as

$$compare_{\leq}(m,n) = \begin{cases} 1, & \text{if } m \leq n \\ 0, & \text{if } m > n, \end{cases}$$

which is given by

$$compare_{\leq}(m,n) = 1 \dot{-} sg(m \dot{-} n).$$

Then we can define

$$cond_{\leq}(m,n,p,q) = \begin{cases} p, & \text{if } m \leq n \\ q, & \text{if } m > n, \end{cases}$$

with

$$cond_{\leq}(m,n,n,p) = compare_{\leq}(m,n) * p + \overline{sg}(compare_{\leq}(m,n)) * q.$$

The above allows to define functions by cases.

We now generalize primitive recursion to functions defined on strings (in Σ^*). The new twist is that instead of the argument $n+1$ of f, we need to consider the k arguments ua_i of f for $i = 1, \ldots, k$ (with $u \in \Sigma^*$), so instead of a single function h, we need k functions h_i to define primitive recursively what $f(ua_i, w_2, \ldots, w_m)$ is.

Definition 1.20. Let $\Sigma = \{a_1, \ldots, a_k\}$. For any partial or total function

$$g \colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m-1} \to \Sigma^*,$$

where $m \geq 2$, and any k partial or total functions

$$h_i \colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

the partial function

$$f \colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*$$

is defined by *primitive recursion from g and $h_1, \ldots, h_k$*, if

$$\begin{aligned} f(\epsilon, w_2, \ldots, w_m) &= g(w_2, \ldots, w_m) \\ f(ua_1, w_2, \ldots, w_m) &= h_1(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m) \\ \cdots &= \cdots \\ f(ua_k, w_2, \ldots, w_m) &= h_k(u, f(u, w_2, \ldots, w_m), w_2, \ldots, w_m), \end{aligned}$$

for all $u, w_2, \ldots, w_m \in \Sigma^*$.

When $m = 1$, for some fixed $w \in \Sigma^*$, we have

$$\begin{aligned} f(\epsilon) &= w \\ f(ua_1) &= h_1(u, f(u)) \\ \cdots &= \cdots \\ f(ua_k) &= h_k(u, f(u)), \end{aligned}$$

for all $u \in \Sigma^*$.

Again, if g and the h_i are total, it is easy to see that f is total.

Example 1.14. As an example over $\{a, b\}^*$, the following function $g\colon \Sigma^* \times \Sigma^* \to \Sigma^*$, is defined by primitive recursion:

$$\begin{aligned} g(\epsilon, v) &= P_1^1(v) \\ g(ua_i, v) &= S_i \circ P_2^3(u, g(u, v), v), \end{aligned}$$

where $1 \leq i \leq k$. It is easily verified that $g(u, v) = vu$. Then,

$$con = g \circ (P_2^2, P_1^2)$$

computes the concatenation function, *i.e.*, $con(u, v) = uv$. The extended concatenation con_{n+1} $(n \geq 1)$ defined by

$$con_{n+1}(x_1, \ldots, x_{n+1}) = x_1 \cdots x_{n+1}$$

is primitive recursive because $con_2 = con$ and

$$\begin{gathered} con_{n+1}(x_1, \ldots, x_{n+1}) = con(con_n(P_1^{n+1}(x_1, \ldots, x_{n+1}), \ldots P_n^{n+1}(x_1, \ldots, x_{n+1})), \\ P_{n+1}^{n+1}(x_1, \ldots, x_{n+1})). \end{gathered}$$

Here are some primitive recursive functions that often appear as building blocks for other primitive recursive functions.

Example 1.15. The *delete last* function *dell* given by

$$\begin{aligned} dell(\epsilon) &= \epsilon \\ dell(ua_i) &= u, \quad 1 \leq i \leq k,\ u \in \Sigma^* \end{aligned}$$

is defined primitive recursively by

$$\begin{aligned} dell(\epsilon) &= \epsilon \\ dell(ua_i) &= P_1^2(u, dell(u)), \quad 1 \leq i \leq k,\ u \in \Sigma^*. \end{aligned}$$

Example 1.16. For every string $w \in \Sigma^*$, the constant function c_w given by

$$c_w(u) = w, \quad \text{for all } u \in \Sigma^*$$

is defined primitive recursively by induction on the length of w by

$$\begin{aligned} c_\epsilon &= E \\ c_{va_i} &= S_i \circ c_v, \quad 1 \leq i \leq k. \end{aligned}$$

Example 1.17. The *sign function sg* given by

$$sg(x) = \begin{cases} \epsilon, & \text{if } x = \epsilon \\ a_1, & \text{if } x \neq \epsilon \end{cases}$$

is defined primitive recursively by

$$\begin{aligned} sg(\epsilon) &= \epsilon \\ sg(ua_i) &= (c_{a_1} \circ P_1^2)(u, sg(u)). \end{aligned}$$

Example 1.18. The *anti-sign function* $\overline{sg}$ given by

$$\overline{sg}(x) = \begin{cases} a_1, & \text{if } x = \epsilon \\ \epsilon, & \text{if } x \neq \epsilon \end{cases}$$

is primitive recursive. The proof is left an an exercise.

Example 1.19. The function end_j $(1 \leq j \leq k)$ given by

$$end_j(x) = \begin{cases} a_1, & \text{if } x \text{ ends with } a_j \\ \epsilon, & \text{otherwise} \end{cases}$$

is primitive recursive. The proof is left an an exercise.

Example 1.20. The reverse function $rev\colon \Sigma^* \to \Sigma^*$ given by $rev(u) = u^R$ is primitive recursive because

$$\begin{aligned} rev(\epsilon) &= \epsilon \\ rev(ua_i) &= (con \circ (c_{a_i} \circ P^2_1, P^2_2))(u, rev(u)), \quad 1 \leq i \leq k. \end{aligned}$$

Example 1.21. The *tail function tail* given by

$$\begin{aligned} tail(\epsilon) &= \epsilon \\ tail(a_i u) &= u \end{aligned}$$

is primitive recursive because

$$tail = rev \circ dell \circ rev.$$

Example 1.22. The *last function last* given by

$$\begin{aligned} last(\epsilon) &= \epsilon \\ last(ua_i) &= a_i \end{aligned}$$

is primitive recursive because

$$\begin{aligned} last(\epsilon) &= \epsilon \\ last(ua_i) &= c_{a_i} \circ P^2_1(u, last(u)). \end{aligned}$$

Example 1.23. The *head function head* given by

$$\begin{aligned} head(\epsilon) &= \epsilon \\ head(a_i u) &= a_i \end{aligned}$$

is primitive recursive because

$$head = last \circ rev.$$

We are now ready to define the class of primitive recursive functions.

Definition 1.21. Let $\Sigma = \{a_1, \ldots, a_k\}$. The class of *primitive recursive functions* is the smallest class of (total) functions (over Σ^*) which contains the base functions and is closed under composition and primitive recursion.

In the special where $k = 1$, we obtain the class of *numerical primitive recursive functions*.

Example 1.24. The function f given by $f(x_1, x_2) = x_1^{|x_2|}$ is defined by primitive recursion as follows. First we introduce g given by $g(x_1, x_2) = x_2^{|x_1|}$, with

$$\begin{aligned} g(\epsilon, x_2) &= \epsilon \\ g(x_1 a_i, x_2) &= con(g(x_1, x_2), x_2). \end{aligned}$$

Then $f(x_1, x_2) = g(x_2, x_1)$. A RAM program in flowchart form computing f is shown in Figure 1.6. Observe how this program makes use of the program for computing concatenation.

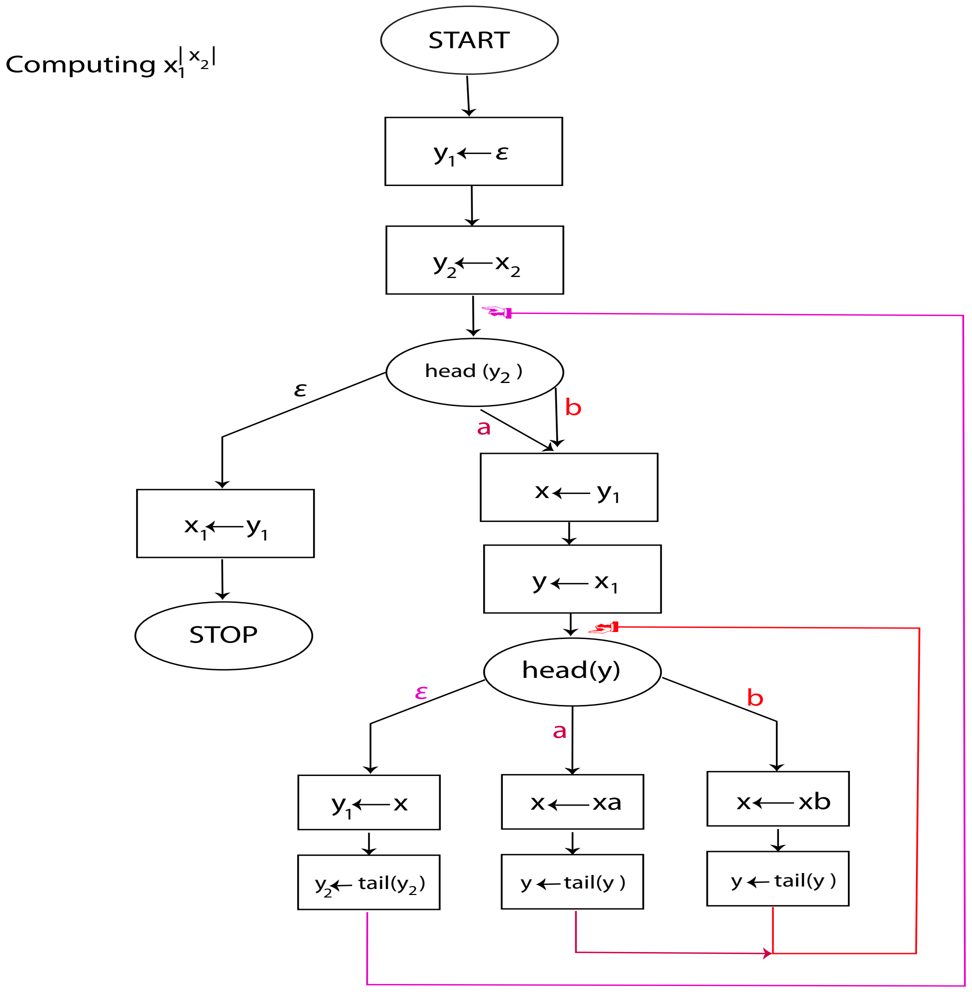


Figure 1.6: Computing $f(x_1, x_2) = x_1^{|x_2|}$ by primitive recursion.

1.8 Primitive Recursive Functions are RAM Computable

The class of primitive recursive functions may not seem very big, but it contains all the total functions that we would ever want to compute. Although it is rather tedious to prove, the following theorem can be shown.

Theorem 1.4. *For any alphabet $\Sigma = \{a_1, \ldots, a_k\}$, every primitive recursive function is RAM computable, and thus Turing computable.*

Proof. We showed just after Definition 1.7 that the base functions are RAM-computable.

Let us first show closure of the class of RAM programs under composition. Let $R, P_1, \ldots,$

P_m be RAM programs computing $g, h_1, \ldots, h_m$, and assume that $h_1, \ldots, h_m$ are functions of n variables. The idea is to use $P_1, \ldots, P_m$ as subroutines to R. Let q be least integer greater than m and n and such that no register of index past q is used in $R, P_1, \ldots, P_m$. The program computing $g \circ (h_1, \ldots, h_m)$ is designed as follows. First, we save the contents of the input registers.

$$\begin{array}{rcl} R(q+1) & \leftarrow & R1 \\ & \vdots & \\ R(q+n) & \leftarrow & Rn \end{array}$$

Next we initialize the noninput registers and compute $h_1(x_1, \ldots, x_n)$ by "calling" $P1$ as a subroutine. The output is stored in $R(q+n+1)$.

$$\begin{array}{rcl} & \texttt{clr} & R(n+1) \\ & \vdots & \\ & \texttt{clr} & Rq \\ & P_1 & \\ R(q+n+1) & \leftarrow & R1 \end{array}$$

We have similar pieces of RAM code to execute $P_2, \ldots, P_m$, the mth piece of code being

$$\begin{array}{lcl} R1 & \leftarrow & R(q+1) \\ & \vdots & \\ Rn & \leftarrow & R(q+n) \\ & \texttt{clr} & R(n+1) \\ & \vdots & \\ & \texttt{clr} & Rq \\ & P_m & \\ R(q+n+m) & \leftarrow & R1 \end{array}$$

At this stage, the values $h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)$ have been computed and are stored in the registers $R(q+n+1), \ldots, R(q+n+m)$, or one of the P_i diverged. We finally call the subroutine R to compute $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$.

$$\begin{array}{rcl} R1 & \leftarrow & R(q+n+1) \\ & \vdots & \\ Rn & \leftarrow & R(q+n+m) \\ & \texttt{clr} & R(m+1) \\ & \vdots & \\ & \texttt{clr} & Rq \\ & R & \end{array}$$

The output is in register $R1$ (or the program diverged). Now the reader should understand why we are using relative addresses in the jumps–this allows us to simply plug in the programs acting as subroutines in the right places. The other instructions simply make sure that these programs are correctly initialized.

Next we show closure of the class of RAM programs under primitive recursion.

Suppose $g, h_1, \ldots, h_k$ are some total functions computable by some RAM programs, with $g\colon (\Sigma^*)^{m-1} \to \Sigma^*$, and $h_i\colon (\Sigma^*)^{m+1} \to \Sigma^*$, for $i = 1, \ldots, k$. If we write $\overline{x}$ for $(x_2, \ldots, x_m)$, for any $y \in \Sigma^*$, where $y = a_{i_1} \cdots a_{i_n}$ (with $a_{i_j} \in \Sigma$), let f be defined by primitive recursion from g and the h_i's, that is,

$$\begin{aligned}
f(\epsilon, \overline{x}) &= g(\overline{x}) \\
f(ya_1, \overline{x}) &= h_1(y, f(y, \overline{x}), \overline{x}) \\
&\;\;\vdots \\
f(ya_i, \overline{x}) &= h_i(y, f(y, \overline{x}), \overline{x}) \\
&\;\;\vdots \\
f(ya_k, \overline{x}) &= h_k(y, f(y, \overline{x}), \overline{x}),
\end{aligned}$$

for all $y \in \Sigma^*$ and all $\overline{x} \in (\Sigma^*)^{m-1}$. Define the following sequences, u_j and v_j, for $j = 0, \ldots, n+1$:

$$\begin{aligned}
u_0 &= \epsilon \\
u_1 &= u_0 a_{i_1} \\
&\;\;\vdots \\
u_j &= u_{j-1} a_{i_j} \\
&\;\;\vdots \\
u_n &= u_{n-1} a_{i_n} \\
u_{n+1} &= u_n a_i,
\end{aligned}$$

and

$$\begin{aligned}
v_0 &= g(\overline{x}) \\
v_1 &= h_{i_1}(u_0, v_0, \overline{x}) \\
&\;\;\vdots \\
v_j &= h_{i_j}(u_{j-1}, v_{j-1}, \overline{x}) \\
&\;\;\vdots \\
v_n &= h_{i_n}(u_{n-1}, v_{n-1}, \overline{x}) \\
v_{n+1} &= h_i(y, v_n, \overline{x}).
\end{aligned}$$

We leave it as an exercise to prove by induction that

$$v_j = f(u_j, \overline{x})$$

for $j = 0, \ldots, n+1$. It follows that

$$f(u_n a_i, \overline{x}) = h_i(u_n, f(u_n, \overline{x}), \overline{x}),$$

so $f(u_n a_i, \overline{x})$ is defined and the function f is total. The RAM program in flowchart form shown in Figure 1.7 implements the computation of the v_j.

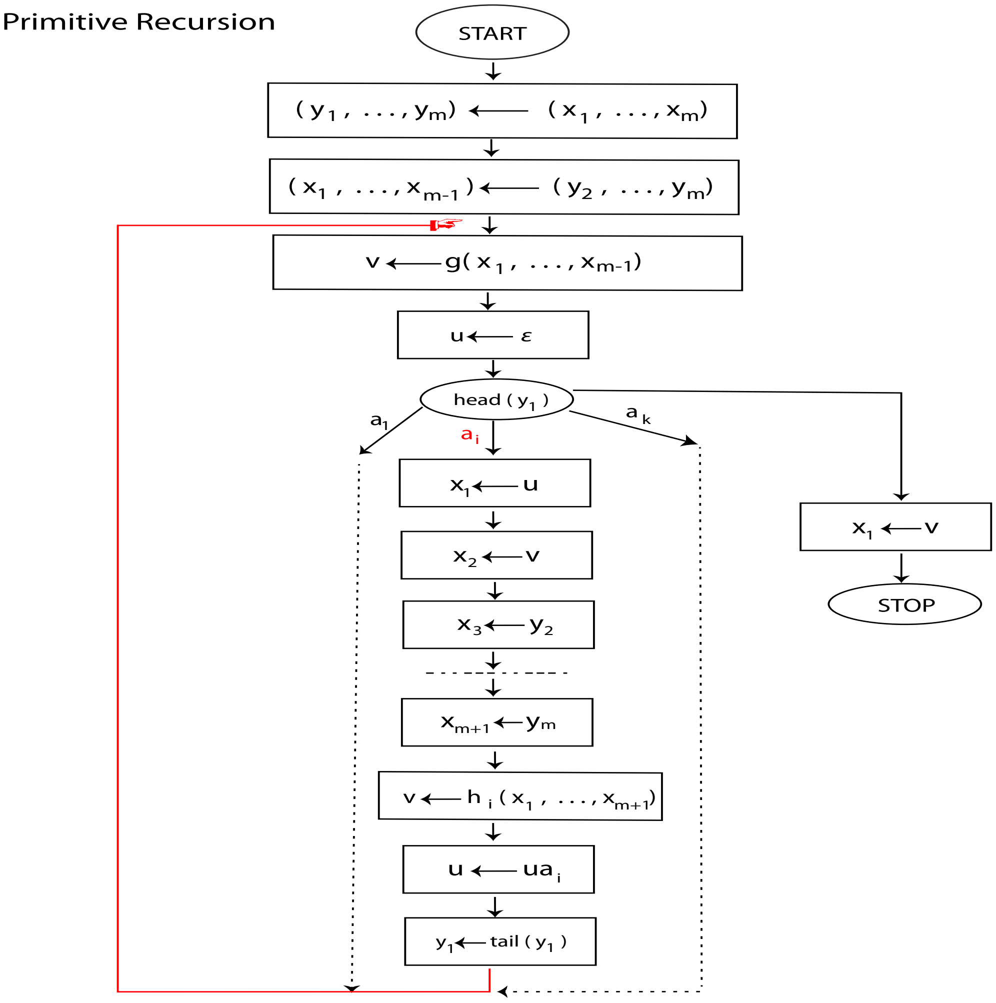


Figure 1.7: Closure under primitive recursion.

A statement such as

$$v \leftarrow g(x_1, \ldots, x_{m-1})$$

is an abbreviation for a RAM program R computing g, in which it is assumed that the variables used by R, except the variables $x_1, \ldots, x_{m-1}$, are not used elsewhere in the program implementing primitive recursion. The same convention applies to the statement

$$v \leftarrow h_i(x_1, \ldots, x_{m+1}).$$ □

In order to define new functions it is also useful to use predicates.

1.9 Primitive Recursive Predicates

Primitive recursive predicates will be used in Section 4.3.

Definition 1.22. An *n-ary predicate* P over $\mathbb{N}$ is any subset of $\mathbb{N}^n$. We write that a tuple $(x_1, \ldots, x_n)$ *satisfies* P as $(x_1, \ldots, x_n) \in P$ or as $P(x_1, \ldots, x_n)$. The *characteristic function* of a predicate P is the function $C_P \colon \mathbb{N}^n \to \{0, 1\}$ defined by

$$C_p(x_1, \ldots, x_n) = \begin{cases} 1, & \text{iff } P(x_1, \ldots, x_n) \text{ holds} \\ 0, & \text{iff } \textbf{not } P(x_1, \ldots, x_n). \end{cases}$$

A predicate P (over $\mathbb{N}$) is *primitive recursive* iff its characteristic function C_P is primitive recursive.

More generally, an *n-ary predicate* P (over Σ^*) is any subset of $(\Sigma^*)^n$. We write that a tuple $(x_1, \ldots, x_n)$ *satisfies* P as $(x_1, \ldots, x_n) \in P$ or as $P(x_1, \ldots, x_n)$.

Definition 1.23. The *characteristic function* of a predicate P is the function $C_P \colon (\Sigma^*)^n \to \{a_1\}^*$ defined by

$$C_p(x_1, \ldots, x_n) = \begin{cases} a_1, & \text{iff } P(x_1, \ldots, x_n) \text{ holds} \\ \epsilon, & \text{iff } \textbf{not } P(x_1, \ldots, x_n). \end{cases}$$

A predicate P (over Σ^*) is *primitive recursive* iff its characteristic function C_P is primitive recursive.

Since we will only need to use primitive recursive predicates over $\mathbb{N}$ in the following chapters, for simplicity of exposition we will restrict ourselves to such predicates. The general case in treated in Machtey and Young [29].

It is easily shown that if P and Q are primitive recursive predicates (over $\mathbb{N}^n$), then $P \lor Q$, $P \land Q$ and $\neg P$ are also primitive recursive.

As an exercise, the reader may want to prove that the predicate,

prime(n) iff n is a prime number,

is a primitive recursive predicate.

For any fixed $k \geq 1$, the function

ord(k, n) = exponent of the kth prime in the prime factorization of n,

is a primitive recursive function.

We can also define functions by cases.

Proposition 1.5. *If $P_1, \ldots, P_m$ are pairwise disjoint primitive recursive n-ary predicates (which means that $P_i \cap P_j = \emptyset$ for all $i \neq j$) and $f_1, \ldots, f_{m+1}$ are primitive recursive functions on $\mathbb{N}^n$, the function $g\colon \mathbb{N}^n \to \mathbb{N}$ defined below is also primitive recursive:*

$$g(\overline{x}) = \begin{cases} f_1(\overline{x}), & \textit{iff } P_1(\overline{x}) \\ \quad\vdots \\ f_m(\overline{x}), & \textit{iff } P_m(\overline{x}) \\ f_{m+1}(\overline{x}), & \textit{otherwise.} \end{cases}$$

Here we write $\overline{x}$ for $(x_1, \ldots, x_n)$.

Proposition 1.5 also applies to functions and predicates with string arguments.

It is also useful to have bounded quantification and bounded minimization. Recall that we are restricting our attention to numerical predicates and functions, so all variables range over $\mathbb{N}$. Proofs of the results stated below can be found in Machtey and Young [29].

Definition 1.24. If P is an $(n+1)$-ary predicate, then the *bounded existential predicate* $(\exists y \leq x)P(y, \overline{z})$ holds iff some $y \leq x$ makes $P(y, \overline{z})$ true.

The *bounded universal predicate* $(\forall y \leq x)P(y, \overline{z})$ holds iff every $y \leq x$ makes $P(y, \overline{z})$ true.

Both $(\exists y \leq x)P(y, \overline{z})$ and $(\forall y \leq x)P(y, \overline{z})$ are $(n+1)$-ary predicates; that is, the input arguments are x and $\overline{z}$.

Proposition 1.6. *If P is an $(n+1)$-ary primitive recursive predicate, then $(\exists y \leq x)P(y, \overline{z})$ and $(\forall y \leq x)P(y, \overline{z})$ are also primitive recursive predicates.*

As an application, we can show that the equality predicate $u = v$ is primitive recursive. The following slight generalization of Proposition 1.6 will be needed in Section 4.3.

Proposition 1.7. *If P is an $(n+1)$-ary primitive recursive predicate and $f\colon \mathbb{N}^n \to \mathbb{N}$ is a primitive recursive function, then $(\exists y \leq f(\overline{z}))P(y, \overline{z})$ and $(\forall y \leq f(\overline{z}))P(y, \overline{z})$ are also primitive recursive predicates.*

Definition 1.25. If P is an $(n+1)$-ary predicate, then the *bounded minimization of P*, $\min(y \leq x)\, P(y, \overline{z})$, is the function defined such that $\min(y \leq x)\, P(y, \overline{z})$ is the least natural number $y \leq x$ such that $P(y, \overline{z})$ if such a y exists, $x + 1$ otherwise.

The *bounded maximization of P*, $\max(y \leq x)\, P(y, \overline{z})$, is the function defined such that $\max(y \leq x)\, P(y, \overline{z})$ is the largest natural number $y \leq x$ such that $P(y, \overline{z})$ if such a y exists, $x + 1$ otherwise.

Both $\min(y \leq x)\, P(y, \overline{z})$ and $\max(y \leq x)\, P(y, \overline{z})$ are functions from $\mathbb{N}^{n+1}$ to $\mathbb{N}$; that is, the input arguments are x and $\overline{z}$.

Proposition 1.8. *If P is an $(n+1)$-ary primitive recursive predicate, then $\min(y \leq x)\, P(y, \overline{z})$ and $\max(y \leq x)\, P(y, \overline{z})$ are primitive recursive functions.*

Bounded existential predicates and bounded universal predicates can also be defined for predicates with string arguments.

Definition 1.26. The *bounded existential predicate* $(\exists y/x)P(y, \overline{z})$ holds iff some prefix y of x makes $P(y, \overline{z})$ true. The *bounded universal predicate* $(\forall y/x)P(y, \overline{z})$ holds iff every prefix y of x makes $P(y, \overline{z})$ true. In both cases the input arguments are x and $\overline{z}$.

Again, if P is primitive recursive, then so are $(\exists y/x)P(y, \overline{z})$ and $(\forall y/x)P(y, \overline{z})$.

Bounded universal quantification can be used to prove that the equality predicate $eq(x, y)$ for strings is primitive recursive. This is surprisingly tricky. One needs a version of monus on strings, namely

$$x - y = \begin{cases} \epsilon, & \text{if } |x| \leq |y| \\ v, & \text{if } |x| > |y| \text{ and } x = uv \text{ with } |u| = |y|. \end{cases}$$

We leave it as an exercise to show that that the above function is primitive recursive.

One also needs the predicate $end(x) = end(y)$ which holds iff $x = y = \epsilon$ or x and y end with the same letter. It is easy to show that this predicate is primitive recursive. Then the predicate $|x| = |y|$ is primitive recursive since it holds iff $x - y = \epsilon$ and $y - x = \epsilon$.

Finally, the reader should verify that we have $eq(x, y)$ iff $|x| = |y|$ and

$$\forall z/x[end(z) = end(rev(rev(y) - (x - z))].$$

We can also define bounded minimization and maximization for predicates with string arguments.

Definition 1.27. The *bounded minimization* $\min(y/x)\, P(y, \overline{z})$ of P is the function defined such that $\min(y/x)\, P(y, \overline{z})$ is the shortest prefix y of x such that $P(y, \overline{z})$ if such a y exists, xa_1 otherwise.

The *bounded maximization* $\max(y/x)\, P(y, \overline{z})$ of P is the function defined such that $\max(y/x)\, P(y, \overline{z})$ is the longest prefix y of x such that $P(y, \overline{z})$ if such a y exists, xa_1 otherwise.

In both cases the input arguments are x and $\overline{z}$. If P is primitive recursive, then so are $\min(y/x)\, P(y, \overline{z})$ and $\max(y/x)\, P(y, \overline{z})$.

So far the primitive recursive functions do not yield all the Turing-computable functions. The following proposition also shows that restricting ourselves to total functions is too limiting.

Let $\mathcal{F}$ be any set of total functions $(f \colon (\Sigma^*)^n \to \Sigma^*)$ that contains the base functions and is closed under composition and primitive recursion (and thus, $\mathcal{F}$ contains all the primitive recursive functions).

Definition 1.28. We say that a function $f\colon \Sigma^* \times \Sigma^* \to \Sigma^*$ is *universal* for the one-argument functions in $\mathcal{F}$ iff for every function $g\colon \Sigma^* \to \Sigma^*$ in $\mathcal{F}$, there is some $n \in \mathbb{N}$ such that

$$f(a_1^n, u) = g(u)$$

for all $u \in \Sigma^*$.

Proposition 1.9. *For any countable set $\mathcal{F}$ of total functions containing the base functions and closed under composition and primitive recursion, if f is a universal function for the functions $g\colon \Sigma^* \to \Sigma^*$ in $\mathcal{F}$, then $f \notin \mathcal{F}$.*

Proof. Assume that the universal function f is in $\mathcal{F}$. Let g be the function such that

$$g(u) = f(a_1^{|u|}, u)a_1$$

for all $u \in \Sigma^*$. We claim that $g \in \mathcal{F}$. It is enough to prove that the function h such that

$$h(u) = a_1^{|u|}$$

is primitive recursive, which is easily shown.

Then, because f is universal, there is some m such that

$$g(u) = f(a_1^m, u)$$

for all $u \in \Sigma^*$. Letting $u = a_1^m$, we get

$$g(a_1^m) = f(a_1^m, a_1^m) = f(a_1^m, a_1^m)a_1,$$

a contradiction. □

Thus, either a universal function for $\mathcal{F}$ is partial, or it is not in $\mathcal{F}$.

In order to get a larger class of functions, we need the closure operation known as minimization.

1.10 The Partial Computable Functions

Minimization can be viewed as an abstract version of a while loop. First let us consider the simpler case of numerical functions.

Consider a function $g\colon \mathbb{N}^{m+1} \to \mathbb{N}$, with $m \geq 0$. We would like to know if for any fixed $n_1, \ldots, n_m \in \mathbb{N}$, the equation

$$g(n, n_1, \ldots, n_m) = 0 \quad \text{with respect to } n \in \mathbb{N}$$

has a solution $n \in \mathbb{N}$, and if so, we return the smallest such solution. Thus we are defining a (partial) function $f\colon \mathbb{N}^m \to \mathbb{N}$ such that

$$f(n_1, \ldots, n_m) = \min\{n \in \mathbb{N} \mid g(n, n_1, \ldots, n_m) = 0\},$$

with the understanding that $f(n_1, \ldots, n_m)$ is undefined otherwise. If g is computed by a RAM program, computing $f(n_1, \ldots, n_m)$ corresponds to the while loop

```
$n := 0$;
while $g(n, n_1, \ldots, n_m) \neq 0$ do
$n := n + 1$;
endwhile
let $f(n_1, \ldots, n_m) = n$.
```

Definition 1.29. For any function $g\colon \mathbb{N}^{m+1} \to \mathbb{N}$, where $m \geq 0$, the function $f\colon \mathbb{N}^m \to \mathbb{N}$ is defined by *minimization from* g, if the following conditions hold for all $n_1, \ldots, n_m \in \mathbb{N}$:

(1) $f(n_1, \ldots, n_m)$ is defined iff there is some $n \in \mathbb{N}$ such that $g(p, n_1, \ldots, n_m)$ is defined for all p, $0 \leq p \leq n$, and
$$g(n, n_1, \ldots, n_m) = 0.$$

(2) When $f(n_1, \ldots, n_m)$ is defined,
$$f(n_1, \ldots, n_m) = n,$$
where n is such that $g(n, n_1, \ldots, n_m) = 0$ and $g(p, n_1, \ldots, n_m) \neq 0$ for every p, $0 \leq p \leq n - 1$. In other words, n is the smallest natural number such that $g(n, n_1, \ldots, n_m) = 0$.

Following Kleene, we write

$$f(n_1, \ldots, n_m) = \mu n[g(n, n_1, \ldots, n_m) = 0].$$

Remark: When $f(n_1, \ldots, n_m)$ is defined, $f(n_1, \ldots, n_m) = n$, where n is the smallest natural number such that Condition (1) holds. It is very important to require that *all* the values $g(p, n_1, \ldots, n_m)$ be defined for all p, $0 \leq p \leq n$, when defining $f(n_1, \ldots, n_m)$. Failure to do so allows non-computable functions.

Minimization can be generalized to functions defined on strings as follows. Given a function $g\colon (\Sigma^*)^{m+1} \to \Sigma^*$, for any fixed $w_1, \ldots, w_m \in \Sigma^*$, we wish to solve the equation

$$g(u, w_1, \ldots, w_m) = \epsilon \quad \text{with respect to } u \in \Sigma^*,$$

and return the "smallest" solution u, if any. The only issue is what does smallest solution mean. We resolve this issue by restricting u to be a string of a_j's, for some fixed letter $a_j \in \Sigma$. Thus there are k variants of minimization corresponding to searching for a shortest string in $\{a_j\}^*$, for a fixed j, $1 \leq j \leq k$.

Let $\Sigma = \{a_1, \ldots, a_k\}$. For any function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

where $m \geq 0$, for every j, $1 \leq j \leq k$, the function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*$$

looks for the shortest string u over $\{a_j\}^*$ (for a fixed j) such that

$$g(u, w_1, \ldots, w_m) = \epsilon.$$

This corresponds to the following while loop:

```
$u := \epsilon$;
while $g(u, w_1, \ldots, w_m) \neq \epsilon$ do
$u := ua_j$;
endwhile
let $f(w_1, \ldots, w_m) = u$
```

The operation of minimization (sometimes called minimalization) is defined as follows.

Definition 1.30. Let $\Sigma = \{a_1, \ldots, a_k\}$. For any function

$$g\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m+1} \to \Sigma^*,$$

where $m \geq 0$, for every j, $1 \leq j \leq k$, the function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{m} \to \Sigma^*,$$

is defined by *minimization over* $\{a_j\}^*$ *from* g, if the following conditions hold for all $w_1, \ldots,$ $w_m \in \Sigma^*$:

(1) $f(w_1, \ldots, w_m)$ is defined iff there is some $n \geq 0$ such that $g(a_j^p, w_1, \ldots, w_m)$ is defined for all p, $0 \leq p \leq n$, and

$$g(a_j^n, w_1, \ldots, w_m) = \epsilon.$$

(2) When $f(w_1, \ldots, w_m)$ is defined,

$$f(w_1, \ldots, w_m) = a_j^n,$$

where n is such that

$$g(a_j^n, w_1, \ldots, w_m) = \epsilon$$

and

$$g(a_j^p, w_1, \ldots, w_m) \neq \epsilon$$

for every p, $0 \leq p \leq n - 1$.

We write

$$f(w_1, \ldots, w_m) = min_j u[g(u, w_1, \ldots, w_m) = \epsilon].$$

Remark: When $f(w_1, \ldots, w_m)$ is defined,

$$f(w_1, \ldots, w_m) = a_j^n,$$

where n is the smallest natural number such that Condition (1) holds. It is very important to require that *all* the values $g(a_j^p, w_1, \ldots, w_m)$ be defined for all p, $0 \leq p \leq n$, when defining $f(w_1, \ldots, w_m)$. Failure to do so allows non-computable functions.

Remark: Inspired by Kleene's notation in the case of numerical functions, we may use the *μ-notation*:

$$f(w_1, \ldots, w_m) = \mu_j u[g(u, w_1, \ldots, w_m) = \epsilon].$$

The class of partial computable functions is defined as follows.

Definition 1.31. Let $\Sigma = \{a_1, \ldots, a_k\}$. The class of *partial computable functions* (in the sense of Herbrand–Gödel–Kleene), also called *partial recursive functions* is the smallest class of partial functions (over Σ^*) which contains the base functions and is closed under composition, primitive recursion, and minimization.

The class of *computable functions* also called *recursive functions* is the subset of the class of partial computable functions consisting of functions defined for every input (i.e., total functions).

One of the major results of computability theory is the following theorem.

Theorem 1.10. *For an alphabet $\Sigma = \{a_1, \ldots, a_k\}$, every partial computable function (partial recursive function) is RAM-computable, and thus Turing-computable. Conversely, every RAM-computable function (or Turing-computable function) is a partial computable function (partial recursive function). Similarly, the class of computable functions (recursive functions) is equal to the class of Turing-computable functions that halt in a proper ID for every input, and to the class of RAM programs that halt for all inputs.*

Sketch of proof. First we prove that every partial computable function is RAM-computable. Since we already know from Theorem 1.4 that the RAM programs contain the base functions and are closed under composition and primitive recursion, it suffices to show that minimization can be implemented by a RAM program. The RAM program in flowchart form shown in Figure 1.8 implements minimization.

By Theorem 1.2, every RAM program can be converted to a Turing machine, so every partial computable function is Turing-computable.

For the converse, one can show that given a Turing machine, there is a primitive recursive function describing how to go from one ID to the next. Then minimization is used to guess whether a computation halts. The proof shows that every partial computable function needs minimization *at most once*. The characterization of the computable functions in terms of TM's follows easily. Details are given in Section 2.3. See also Machtey and Young [29] and Kleene I.M. [24] (Chapter XIII). □

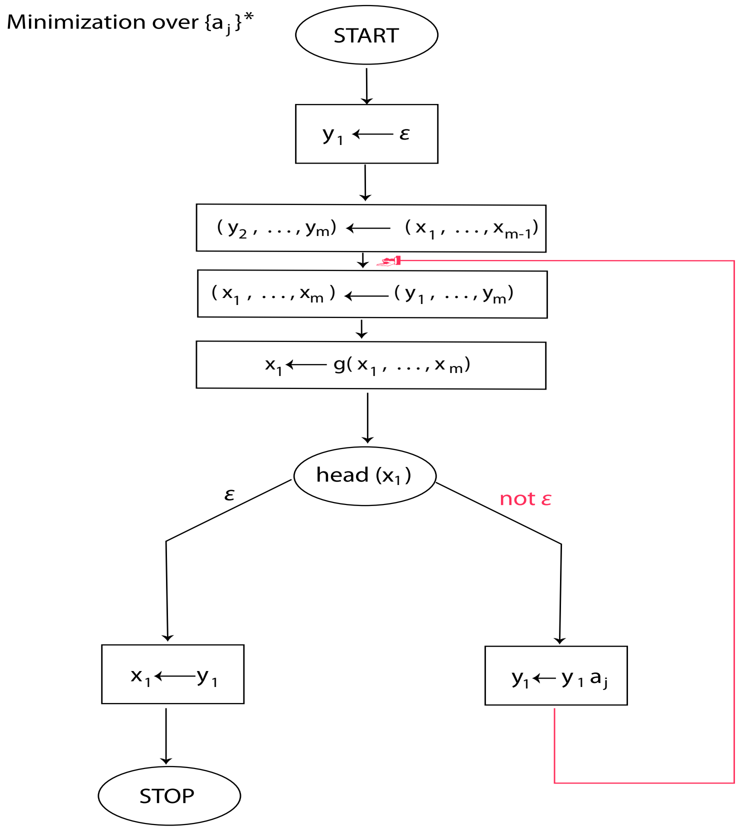


Figure 1.8: Closure under minimization.

We will prove directly in Section 4.3 that every RAM-computable function (over $\mathbb{N}$) is partial computable. This will be done by encoding RAM programs as natural numbers.

There are computable functions (recursive functions) that are not primitive recursive. Such an example is given by Ackermann's function.

Example 1.25. *Ackermann's function* is the function $A\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which is defined by the following recursive clauses:

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

It turns out that A is a computable function which is **not** primitive recursive. This is not easy to prove. It can be shown that

$$\begin{aligned}A(0,\,x) &= x+1\\ A(1,\,x) &= x+2\\ A(2,\,x) &= 2x+3\\ A(3,\,x) &= 2^{x+3}-3,\end{aligned}$$

and

$$A(4,x) = \left.2^{2^{\cdot^{\cdot^{2^{16}}}}}\right\}x \; -3,$$

with $A(4,0) = 16-3 = 13$.

For example

$$A(4,1) = 2^{16}-3, \quad A(4,2) = 2^{2^{16}}-3.$$

Actually, it is not so obvious that A is a total function, but it is.

Proposition 1.11. *Ackermann's function A is a total function.*

Proof. This is shown by induction, using the lexicographic ordering $\preceq$ on $\mathbb{N}\times\mathbb{N}$, which is defined as follows:

$$\begin{aligned}&(m,n) \preceq (m',n') \quad \text{iff either}\\ &m = m' \text{ and } n = n', \text{ or}\\ &m < m', \text{ or}\\ &m = m' \text{ and } n < n'.\end{aligned}$$

We write $(m,n) \prec (m',n')$ when $(m,n) \preceq (m',n')$ and $(m,n) \neq (m',n')$.

We prove that $A(m,n)$ is defined for all $(m,n) \in \mathbb{N}\times\mathbb{N}$ by complete induction over the lexicographic ordering on $\mathbb{N}\times\mathbb{N}$.

In the base case, $(m,n) = (0,0)$, and since $A(0,n) = n+1$, we have $A(0,0) = 1$, and $A(0,0)$ is defined.

For $(m,n) \neq (0,0)$, the induction hypothesis is that $A(m',n')$ is defined for all $(m',n') \prec (m,n)$. We need to conclude that $A(m,n)$ is defined.

If $m = 0$, since $A(0,n) = n+1$, $A(0,n)$ is defined.

If $m \neq 0$ and $n = 0$, since

$$(m-1,1) \prec (m,0),$$

by the induction hypothesis, $A(m-1,1)$ is defined, but $A(m,0) = A(m-1,1)$, and thus $A(m,0)$ is defined.

If $m \neq 0$ and $n \neq 0$, since

$$(m,n-1) \prec (m,n),$$

by the induction hypothesis, $A(m, n-1)$ is defined. Since

$$(m-1, A(m, n-1)) \prec (m, n),$$

by the induction hypothesis, $A(m-1, A(m, n-1))$ is defined. But $A(m, n) = A(m-1, A(m, n-1))$, and thus $A(m, n)$ is defined.

Thus, $A(m, n)$ is defined for all $(m, n) \in \mathbb{N} \times \mathbb{N}$. $\square$

It is possible to show that A is a computable (recursive) function, although the quickest way to prove it requires some fancy machinery (the recursion theorem; see Section 6.1). Proving that A is *not* primitive recursive is even harder.

A further study of the partial recursive functions requires the notions of pairing functions and of universal functions (or universal Turing machines).

Chapter 2

Equivalence of the Models of Computation

2.1 Simulation of a RAM Program by a Turing Machine

It is convenient to describe Turing machines using diagrams. We can use a labeled graph representation where each transition (p, a, b, m, q) is represented by the diagrams shown in Figure 2.1.

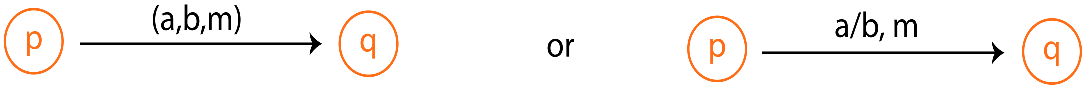


Figure 2.1: Representation of a Turing machine instruction.

There is another convenient notation which can be used, if for each state, all transitions entering that state cause the head to move in the same direction. If this condition is not satisfied, by splitting states, an equivalent Turing machine can be effectively constructed and we leave the construction as an exercise. The situation is now the following. Given an instruction $(p, a, b, m, q) \in \delta$, we have the diagram shown in Figure 2.2.

There is a sight problem if p is not entered by any transition. But then, either p is the start state, in which case we use the notation shown in Figure 2.3, or else p is inaccessible and we can get rid of quintuples starting with p. Otherwise, all transitions entering p cause the tape to move in the same direction m', and we draw the diagram shown in Figure 2.4.

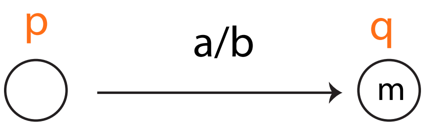


Figure 2.2: Representation of a Turing machine instruction.

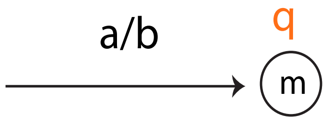


Figure 2.3: Transition from the start state.

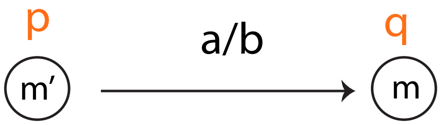


Figure 2.4: A typical transition.

Further simplifications are possible. When no confusion arises, we can omit state names. Transitions (p, a, a, m, q) are represented by the diagram of Figure 2.5, and transitions

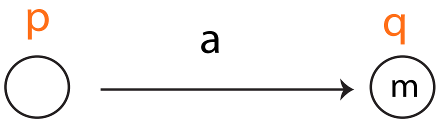


Figure 2.5: A simplified transition.

(p, a, a, m, p) are simply omitted. In other words, loops from a state to itself that do not change the current symbol being scanned are omitted.

For all blocking pairs (p, a), that is, pairs such that no quintuple in δ begins with (p, a), we draw an outgoing arrow from state p labeled a as shown in Figure 2.6.

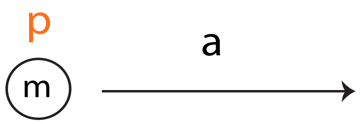


Figure 2.6: A blocking transition.

Example 2.1. Consider the Turing machine M with $K = \{q_0, q_1, q_2, q_3\}$, $\Gamma = \{a, b, B\}$, and δ consisting of the following quintuples:

$$\begin{gathered}
q_0, B \to B, R, q_3, \\
q_0, a \to b, R, q_1, \\
q_0, b \to a, R, q_1, \\
q_1, a \to b, R, q_1, \\
q_1, b \to a, R, q_1, \\
q_1, B \to B, L, q_2, \\
q_2, a \to a, L, q_2, \\
q_2, b \to b, L, q_2, \\
q_2, B \to B, R, q_3.
\end{gathered}$$

The diagram (using the above conventions) corresponding to the Turing machine M is shown in Figure 2.7.

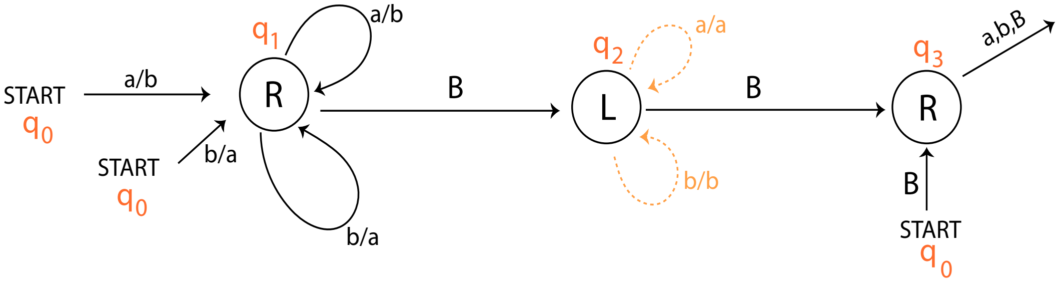


Figure 2.7: Diagram of the Turing machine M.

For any input $u \in \{a, b\}^*$, the output of the computation is the string v obtained from u by changing each "a" into a "b" and each "b" into an "a".

We now describe a construction which takes a RAM program as input and produces as output a Turing machine computing the same function as the function computed by the

RAM program. This construction provides a proof for Theorem 1.2 that we repeat for the convenience of the reader.

Theorem 2.1. *Every RAM-computable function is Turing-computable. Furthermore, given a RAM program P, we can effectively construct a Turing machine M computing the same function.*

Proof. Let P be a RAM program using m registers $R1, \ldots Rm$ and having n instructions. The contents $r_1, \ldots, r_m$ of the registers are represented on the Turing machine tape by the string

$$\#r1\#r2\#\cdots\#rm\#,$$

where $\#$ is a special marker and ri represents the string held by Ri. We also use Proposition 1.1, which allows us to restrict ourselves to RAM programs that use only instructions of the form

$$\begin{array}{lllll}
(1_j) & N & & \texttt{add}_j & Y \\
(2) & N & & \texttt{tail} & Y \\
(6_ja) & N & Y & \texttt{jmp}_j & N1a \\
(6_jb) & N & Y & \texttt{jmp}_j & N1b \\
(7) & N & & \texttt{continue} &
\end{array}$$

The simulating Turing machine M is built of n blocks connected for the same flow of control as the n instructions in P. The jth block of the Turing machine simulates the jth instruction in P.

The machine M begins with some initialization whose purpose is to make sure that the simulation starts with a tape of the form

$$\#r1\#r2\#\cdots\#rm\#$$

representing m registers, with $m{+}1$ symbols $\#$. Since the RAM program could have a number of input variables $t < m$, and it is necessary to add $m + 2 - t$ symbols $\#$. If the input is $x_1, x_2, \cdots, x_t$, the $t - 1$ commas are changed to $\#$, and we add $m + 1 - (t - 1) = m + 2 - t$ symbols $\#$. For example, if $m = 5$ and $t = 3$, the Turing input tape ab, bb, a becomes $\#ab\#bb\#a\#\#\#$. See Figure 2.8 for the Turing machine achieving this step.

To simplify our diagrams, let us assume that the RAM alphabet is $\Sigma = \{0, 1\}$. Then the alphabet of the Turing machine is $\Gamma = \{0, 1, \#, B\}$. Each RAM statement is translated as a Turing machine block as follows. We have four blocks, one for each instruction.

(a) $\texttt{add}_i$ Rq. See Figure 2.9.

(b) $\texttt{tail}$ Rq. See Figure 2.10.

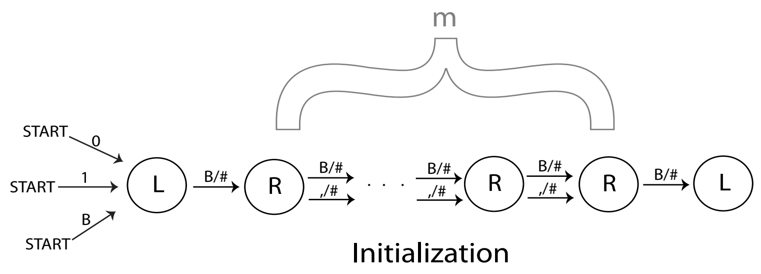


Figure 2.8: Initialization.

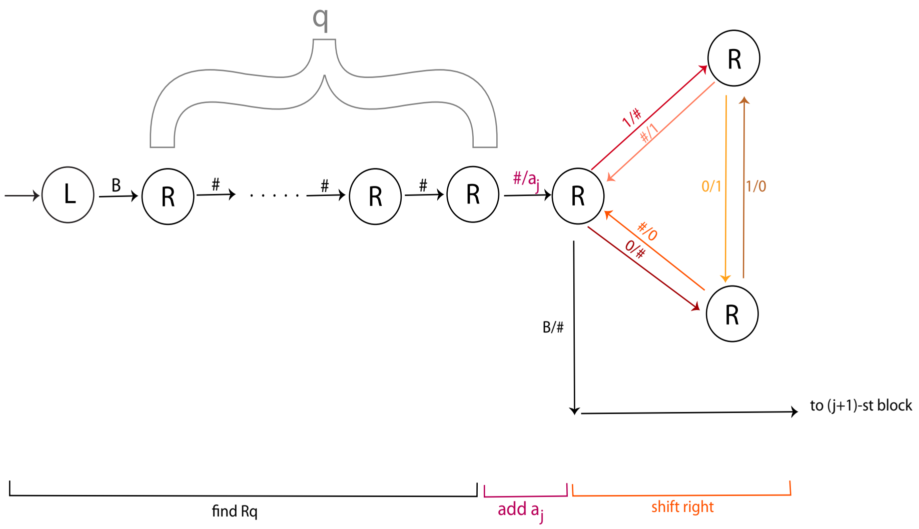


Figure 2.9: Simulation of an instruction $\texttt{add}_i$ Rq.

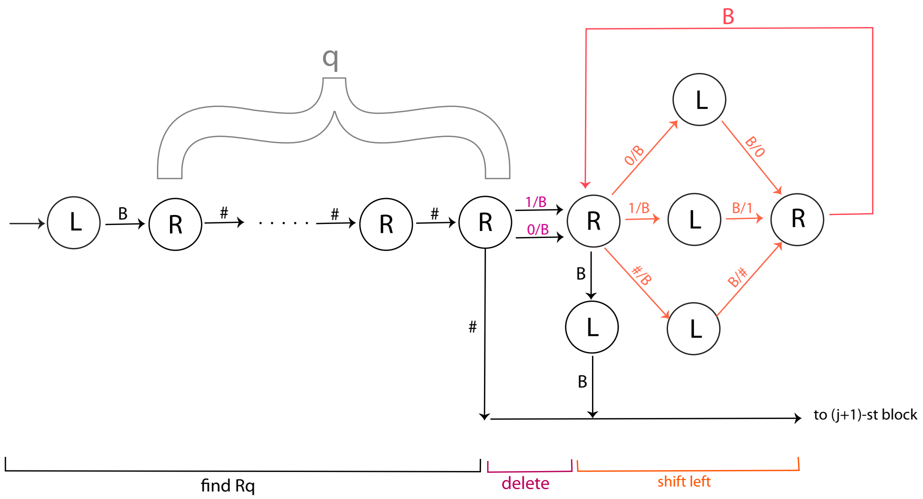


Figure 2.10: Simulation of an instruction $\texttt{tail}$ Rq.

(c) $\texttt{jmp}_i\ Z$

There are two variants of this case, since Z is either a jump above or a jump below. These two cases are handled similarly, the only difference being the address of the block to jump to. See Figure 2.11.

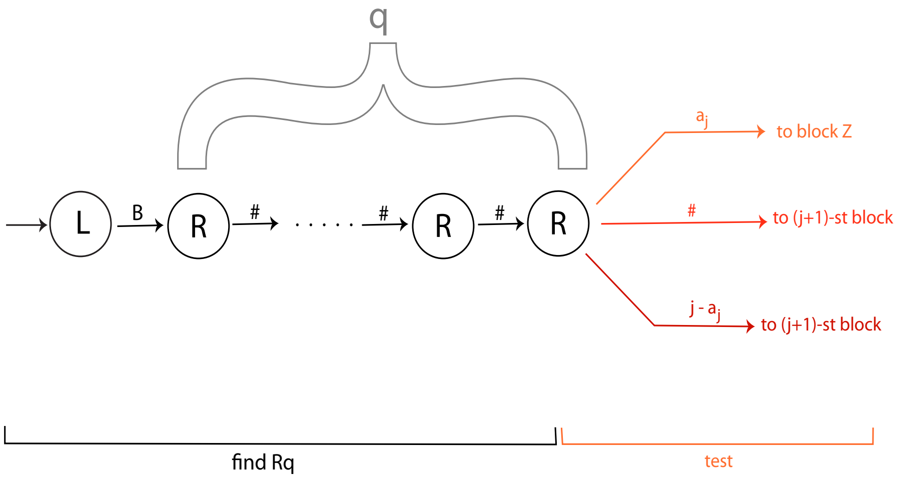


Figure 2.11: Simulation of an instruction $\texttt{jmp}_i\ Z$.

Finally, we clean up the tape by erasing all but the contents of $R1$ from the tape. This block corresponds to the last `continue` statement.

(d) Clean up phase. See Figure 2.12.

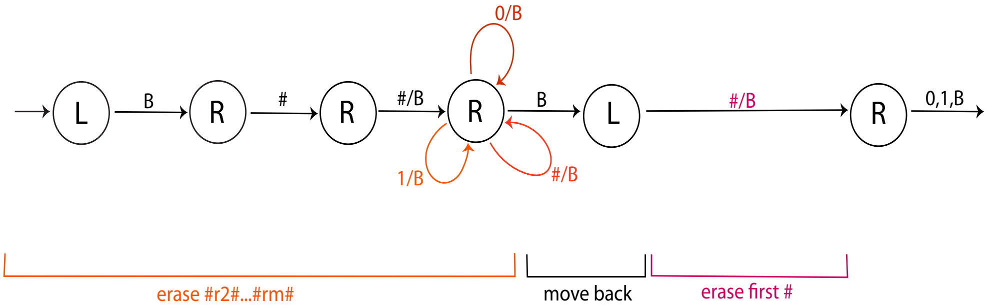


Figure 2.12: Clean up phase.

Also note that a `continue` statement which is not the last `continue` statement in the RAM program is translated as an arrow from the exit of the jth block to the entry of the $(j + 1)$th block.

Notice that the Turing machine produced by the construction has the nice property that it never moves left of the blank square immediately to the left of its leftmost #. In other words, the tape need only be unbounded to the right. We leave as an exercise to prove that every Turing-computable function is computable by a Turing machine which never moves more than one square to the left of its starting position. □

Example 2.2. Here is an example of the simulation for a RAM program with two input registers and a total of four registers. The input values are 101 in $R1$ and 00 in $R2$. The initialization phase is shown in Figure 2.13.

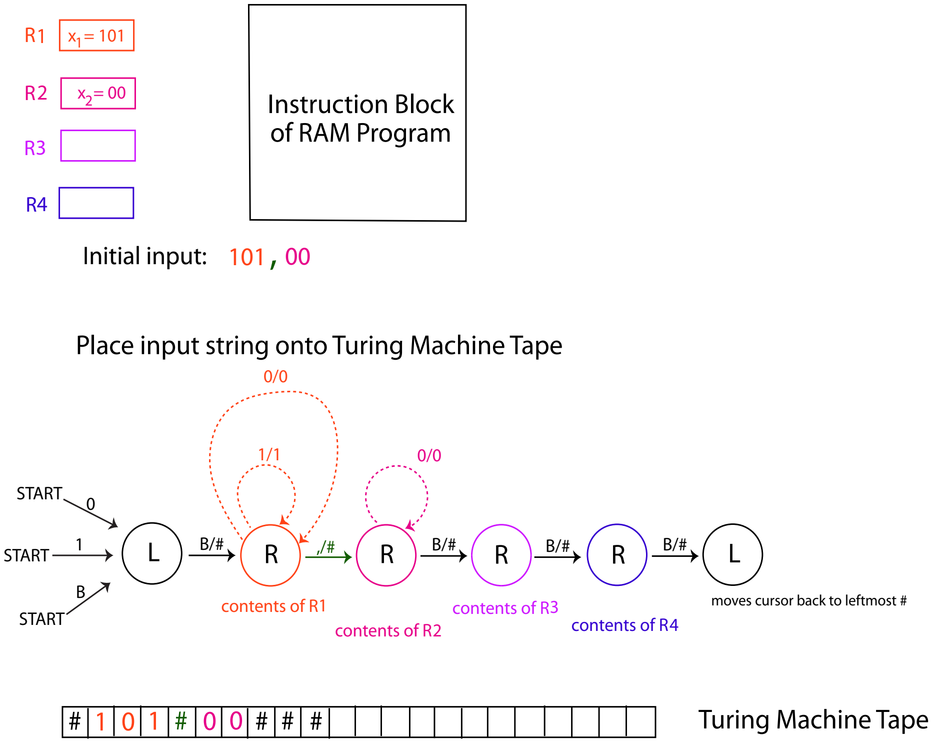


Figure 2.13: Initialization phase.

The simulation of the instruction $\texttt{add}_0$ $R1$ is shown in Figure 2.14.

The simulation of the instruction `tail` $R2$ is shown in Figure 2.15.

Next we show that every Turing computable function is RAM-computable.

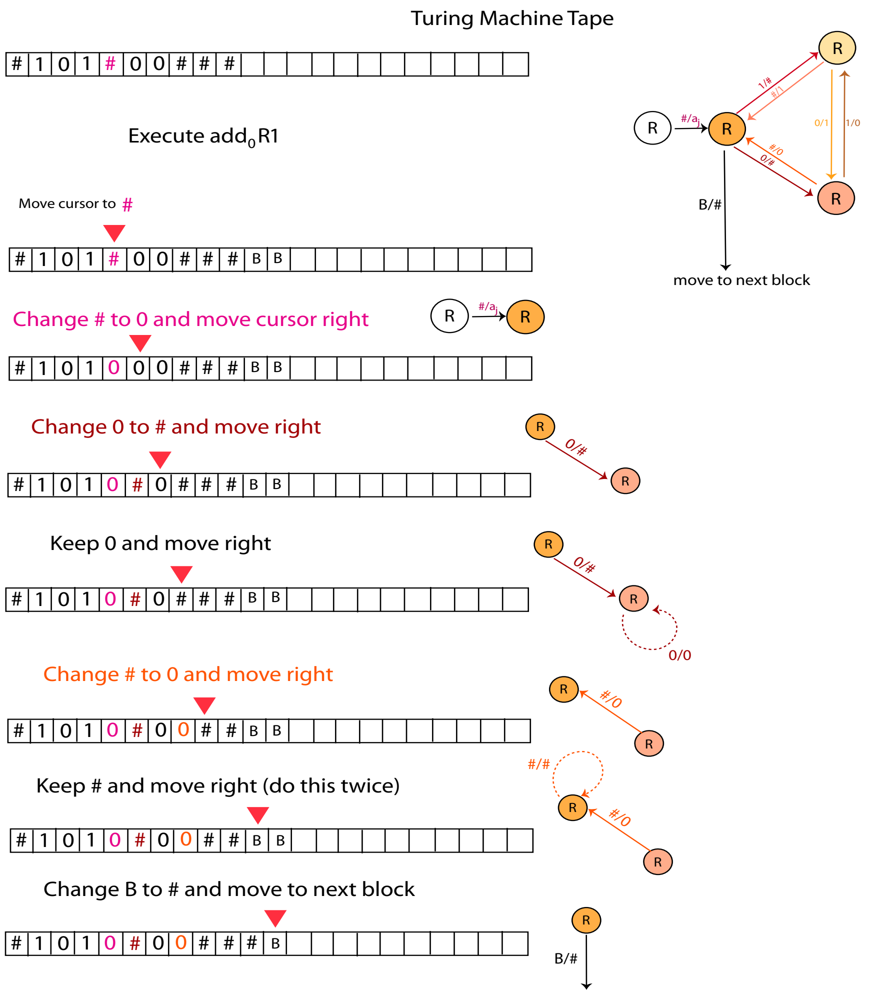


Figure 2.14: Simulation of the instruction $\texttt{add}_0$ $R1$.

2.2 Simulation of Turing Machine by a RAM Program

In this section we provide a proof of Theorem 1.3 which we repeat for the reader's convenience.

Theorem 2.2. *Every Turing-computable function is RAM-computable. Furthermore, given a Turing machine M, one can effectively construct a RAM program P computing the same function.*

Proof. Recall that we showed that the concatenation function *con* and the extended concatenation function con_n defined such that $con_n(x_1, \ldots, x_n) = x_1 \cdots x_n$ are primitive recursive

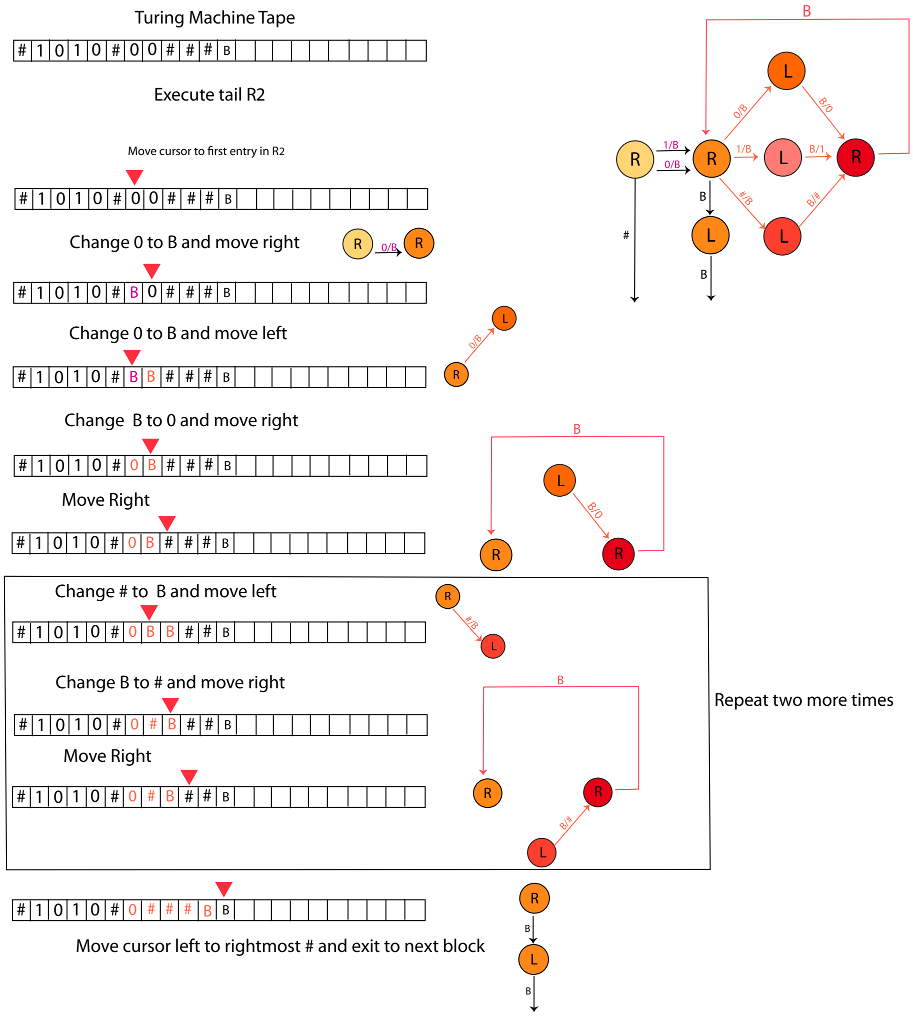


Figure 2.15: Simulation of the instruction `tail` $R2$.

and consequently RAM-computable. Also, RAM programs are closed under composition. This allows to write a RAM program as a composition of blocks, avoiding the tedious task of writing the program in full.

Let $M = (K, \Gamma, \Delta, \delta, q_0)$ be a Turing machine with $K = \{q_0, \ldots, q_m\}$ and $\Gamma = \{a_1, \ldots, a_k, B, \text{``,'}\}$, and let φ be the partial function of n arguments computed by M.

The idea of the proof is to design a RAM program P containing an encoding of the current ID of the Turing machine M in register $R1$, and to use other registers $R2, R3$ to simulate the effect of executing an instruction of M by updating the ID of M in $R1$. After some initialization, the program P contains the current ID of M in register $R1$. For each move of M, the program P updates the current ID to the next ID.

Initially, P takes the n input strings $x_1, \ldots, x_n$ and creates

$$\#ID_0\# = \#q_0x_1, x_2, \cdots, x_n\#$$

in register $R1$ and then simulates M. If and when M halts in a halting ID of the form B^kqwB^ℓ, the program P places w in $R1$ and stops. If the ID is improper, then P loops forever.

The alphabet for P is $\Sigma = \Gamma \cup K \cup \{\#\}$, and it is assumed that $\Gamma \cap K = \emptyset$ and that $\#$ is neither in Γ nor K. We let $a_{k+1} = B$ and $a_{k+2} = \#$.

When P simulates a move of M by updating the ID, register $R1$ contains the current ID, which is of the form ua_jpa_iv and satisfies the following properties: if $u = \epsilon$, then $a_j = \#$, and if v consists of single symbol, then $v = \#$.

During the first phase in which P updates the ID, P transfers u into register $R2$, a_j into register $R3$, and pa_iv is left in $R1$. Then it reads a_i and, depending on (p, a_i), it simulates the action of M. In order to remember p and a_i, the program P has labels of the form jp and jpi. Right moves are accomplished at the addresses $jpiR$ and $jpiR\#$. Left moves are accomplished at the addresses $jpiL$ and $jpiL\#$. The updated ID is placed back into $R1$. When a halting ID is found, P checks that this ID is proper. If the halting ID is proper, then the output is returned in $R1$, otherwise P loops forever. For simplicity we adopt a subroutine notation. We also omit the suffix a or b in the target labels of jumps, which is not a problem since all jumps in P are uniquely defined.

We initialize P with the following commands:

$$R1 = con_{2n+2}(\#, q_0, x_1, \text{``,''}, \cdots, \text{``,''}, x_n, \#)$$

```
BEGIN   clr      R2
        clr      R3
        jmp      TEST
NU      tail     R1
TEST    R1       jmp_1         A1
                   ⋮
        R1       jmp_{k+2}     A(k+2)
        R1       jmp_{q_0}     Q0
                   ⋮
        R1       jmp_{q_m}     Qm
```

The subroutine Ai is the following program:

```
$Ai$          $R3$          jmp$_1$       $ui1$
              ⋮
              $R3$          jmp$_{k+2}$   $ui(k+2)$
              add$_i$       $R3$
              jmp           $NU$
$ui1$         add$_1$       $R2$
              jmp           $upr3$
              ⋮
$ui(k+2)$     add$_{k+2}$   $R2$
              jmp           $upr3$
$upr3$        tail          $R3$
              add$_i$       $R3$
              jmp           $NU$
```

To remember a_jp, for each $p, 0 \leq p \leq m$, we have

```
$Qp$   $R3$   jmp$_1$       $1p$
       ⋮
       $R3$   jmp$_{k+2}$   $(k+2)p$
```

To remember a_jpa_i, for each $p, 0 \leq p \leq m$, we have

```
$jp$   tail          $R1$
$R1$   jmp$_1$       $jp1$
       ⋮
$R1$   jmp$_{k+1}$   $jp(k+1)$
```

Next we have three cases.

(1) (Right move) To simulate the instruction (p, a, b, R, q) corresponding to the transition on ID's given by

$$ua_jpa_iv \to ua_jbqv, \quad v \neq \#$$

we have the program

```
$jpi$    tail                          $R1$
$R1$     jmp$_1$                       $jpiR$
         ⋮
$R1$     jmp$_{k+1}$                   $jpiR$
$R1$     jmp$_{k+2}$                   $jpiR\#$
$jpiR$   $R1 = con_3(R2, a_jbq, R1)$
         jmp                           $BEGIN$
```

To simulate the transition

$$ua_jpa_i \rightarrow ua_jbqB$$

corresponding to the case where $v = \#$, in which case a blank needs to be inserted as the rightmost symbol on the tape, we have the program

```
$jpiR\#$   $R1 = con_2(R2, a_jbqB\#)$
           jmp                          $BEGIN$
```

(2) (Left move) To simulate the instruction (p, a, b, L, q), corresponding to the transition on ID's given by

$$ua_jpa_iv \rightarrow uqa_jbv, \quad u \neq \epsilon$$

we have the program

```
$jpi$    tail                          $R1$
$R1$     $\mathtt{jmp}_1$                    $jpiL$
           ⋮
$R1$     $\mathtt{jmp}_{k+1}$                $jpiL$
$R1$     $\mathtt{jmp}_{k+2}$                $jpiL\#$
$jpiL$   $R1 = con_3(R2, qa_jb, R1)$
         jmp                           $BEGIN$
```

To simulate the transition

$$pa_iv \rightarrow qBbv$$

corresponding to the case where $u = \epsilon$, in which case a blank needs to be inserted as the lefmost symbol on the tape, we have the program

```
$jpiL\#$   $R1 = con_2(\#qBb, R1)$
           jmp                    $BEGIN$
```

(3) If no quintuple begins with (p, a_i), then upa_iv is a halting ID. We test if it is proper. For each such jpi, we have the program shown below.

```
$jpi$   tail     $R1$
        jmp    $PROPER$
```

The program PROPER checks that an ID is proper. It should be noted that this is unnecessary if the Turing machine has the property that if it halts, then the ID is proper. This can be achieved by modifying the Turing machine so that if it halts in an improper ID, then it loops.

First, the program PROPER checks that the ID starts with a string of the form $\#B^kq$. Next it places the output in $R1$, and finally it checks that the ID ends with $B^\ell\#$.

```
PROPER  $R1 = con_3(R2, a_j p a_i, R1)$
        R2                      ←         R1
        R2                      jmp_#     B
        jmp                               LOOP
HEAD    R2                      jmp_B     B
        R2                      jmp_{q_0} Q
        ⋮
        R2                      jmp_{q_m} Q
        jmp                               LOOP
B       tail                     R2
        jmp                               HEAD
Q       clr                      R1

                MORE  tail   R2
                      R2     jmp_1   RES1
                      ⋮
                      R2     jmp_k   RESk
                      R2     jmp_B   BTAIL
                      R2     jmp_#   STOP
                      jmp            LOOP
```

For each $i, 1 \leq i \leq k$, we have the program

```
RESi    add_i      R1
        jmp               MORE
BTAIL   tail       R2
        R2       jmp_B    BTAIL
        R2       jmp_#    STOP
        jmp               LOOP
LOOP    jmp               LOOP
STOP    continue
```

Example 2.3. Here is an example of the simulation of the Turing machine of Example 1.5 that exchanges a's and b's by a RAM program. The input is ab. The simulation of the transition $q_0ab \to bq_1b$ is shown in Figure 2.16. The simulation of the transition $bq_1b \to baq_1B$ is shown in Figure 2.17.

We leave the following proposition as an exercise.

Proposition 2.3. *Given a Turing machine M computing a function φ, we can effectively construct a Turing machine M' also computing φ with the following additional properties.*

(1) M' halts in a proper ID iff M halts in a proper ID.

(2) M' loops iff either M loops or M halts in an improper ID.

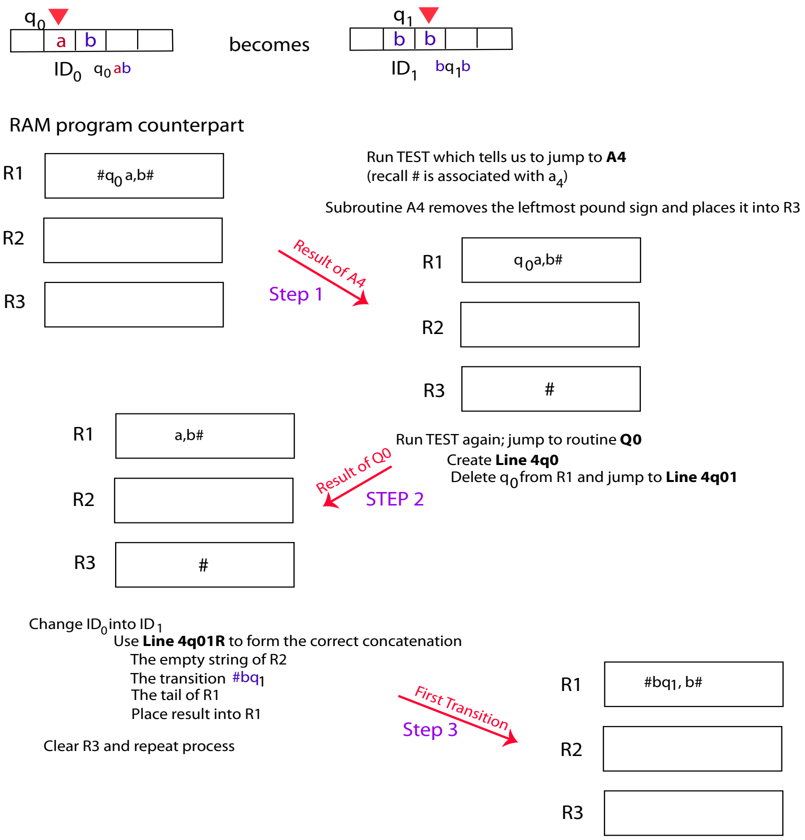

q0
a b
ID0 q0ab
becomes
q1
b b
ID1 bq1b
RAM program counterpart
R1 #q0 a,b#
R2
R3
Run TEST which tells us to jump to A4
(recall # is associated with a4)
Subroutine A4 removes the leftmost pound sign and places it into R3
Result of A4
Step 1
R1 q0a,b#
R2
R3 #
R1 a,b#
R2
R3 #
Run TEST again; jump to routine Q0
Create Line 4q0
Delete q0 from R1 and jump to Line 4q01
Result of Q0
STEP 2
Change ID0 into ID1
Use Line 4q01R to form the correct concatenation
The empty string of R2
The transition #bq1
The tail of R1
Place result into R1
Clear R3 and repeat process
First Transition
Step 3
R1 #bq1, b#
R2
R3

Figure 2.16: Simulation of the transition $q_0ab \to bq_1b$.

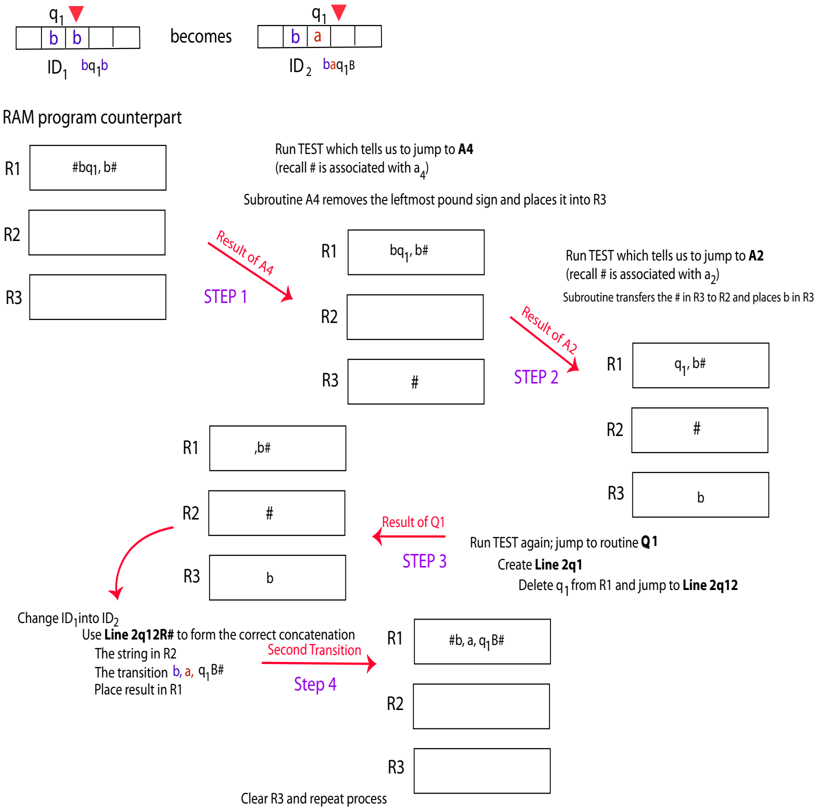
The construction is possible because a Turing machine is capable of checking whether or not a halting ID of M is proper, and if impoper, it loops forever. The construction is very similar to the program PROPER, as a Turing machine. □

2.3 Every Turing Computable Function is Partial Computable a la Herbrand–Gödel–Kleene

The key to the proof that every Turing-computable function is a partial computable function in the sense of Herbrand–Gödel–Kleene is that we can define a primitive recursive function which simulates the transitions of a Turing machine in terms of instantaneous descriptions (ID's).

Instantaneous descriptions are represented as strings $\#upav\#$, where p is a state, $a \in \Gamma$, and $u, v \in \Gamma^*$.

Given a Turing machine $M = (K, \Gamma, \Delta, \delta, q_0)$ (with $\Sigma = \{a_1, \ldots, a_k\}$) we define the



Run TEST which tells us to jump to **A4**
(recall # is associated with a_4)

Subroutine A4 removes the leftmost pound sign and places it into R3

Run TEST which tells us to jump to **A2**
(recall # is associated with a_2)

Subroutine transfers the # in R3 to R2 and places b in R3

Run TEST again; jump to routine **Q1**
Create **Line 2q1**
Delete q_1 from R1 and jump to **Line 2q12**

Change ID_1 into ID_2
Use **Line 2q12R#** to form the correct concatenation
The string in R2
The transition b, a, q_1B#
Place result in R1

Clear R3 and repeat process

Figure 2.17: Simulation of the transition $bq_1b \to baq_1B$.

following pairs of ID's describing the transitions of M.

(1) For every (move right) instruction $(p, a, b, R, q) \in \delta$, we have the pairs

$$\begin{gathered}(paa_1, bqa_1)\\ \vdots\\ (paa_k, bqa_k)\\ (pa\#, bqB\#).\end{gathered}$$

(2) For every (move left) instruction $(p, a, b, L, q) \in \delta$, we have the pairs

$$\begin{gathered}(a_1pa, qa_1b)\\ \vdots\\ (a_kpa, qa_kb)\\ (\#pa, \#qBb).\end{gathered}$$

The above set of pairs is denoted TRANS, and it is assumed to be ordered in some fashion. As an abbreviation each pair is denoted $\ell_i \to r_i$, for example, $paa_1 \to bqa_1$ and $a_1pa \to qa_1b$. We assume that there are N such pairs (this is the number of quintuples in δ).

We also have a list BLOCKED of strings pa such that no quintuple in δ starts with (p, a), say

$$p_{i_1}a_{i_1}, \ldots, p_{i_m}a_{i_m}.$$

An illustration of the rules $\ell_i \to r_i$ is shown in Figure 2.18.

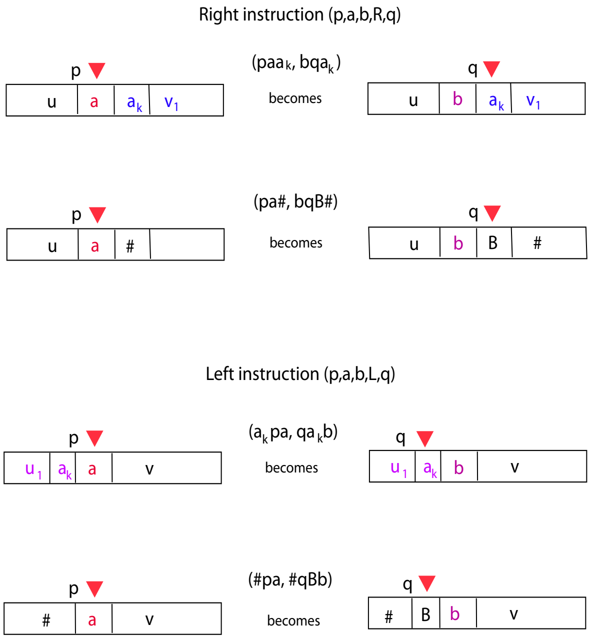


Figure 2.18: Illustration of the rules associated to transitions.

We will use a number of primitive recursive functions.

Proposition 2.4. *The following functions are primitive recursive.*

(1) $\mathrm{Occ}(x, y)$*, where* $\mathrm{Occ}(x, y)$ *holds iff* x *is a substring of* y*.*

(2) $u(x, z) =$ *the prefix of* z *the left of the leftmost occurrence of* x *in* z *if* $\mathrm{Occ}(x, z)$*.*

(3) $v(x, z) =$ *the suffix of* z *the right of the leftmost occurrence of* x *in* z *if* $\mathrm{Occ}(x, z)$*.*

(4) $\mathrm{rep}(x, y, z) =$ *the result of replacing the leftmost occurrence of* x *by* y *in* z *if* $\mathrm{Occ}(x, z)$*.*

Proof. Recall that concatenation and extended concatenation are primitive recursive.

(1) $\mathrm{Occ}(x, y)$ iff $(\exists z/y)(\exists w/y)[z = wx]$.

(2) $u(x, z) = \min y/z(\exists w/z)[yx = w]$.

(3) $v(x, z) = z - u(x, z)x$ (here $-$ is the version of monus on strings defined just after Definition 1.26).

(4) $\mathrm{rep}(x, y, z) = u(x, z)yv(x, z)$.

Note that for every ID, there is at most one occurrence of ℓ_i or r_i for some $\ell_i \to r_i$ in TRANS. This is why it doesn't hurt to pick the leftmost occurrence. □

The predicate Occ is illustrated in Figure 2.19.

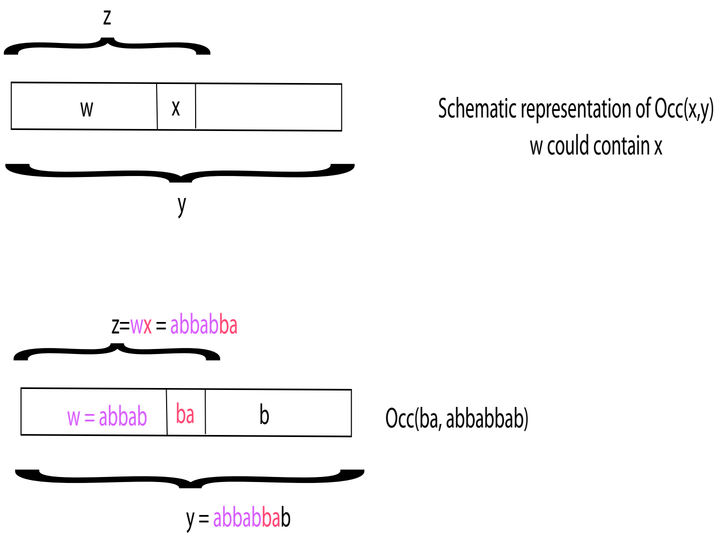


Figure 2.19: Illustration of the predicate Occ.

The functions u and v are illustrated in Figure 2.20. The function rep is illustrated in Figure 2.21.

Proposition 2.5. *For any Turing machine M, the following functions are primitive recursive.*

(1) The function T such that $T(ID_0, y) = ID$ iff $ID_0 \vdash^*_{|y|} ID$ in $|y|$ steps.

(2) $\mathrm{HALT}(ID)$ iff ID is a halting ID.

(3) $\mathrm{STOP}(y, ID)$ iff M halts in a halting ID after $|y|$ steps.

Proof. Note that we do not actually care what $T, \mathrm{HALT}, \mathrm{STOP}$ do if ID_0 and ID are not proper representations of ID's.

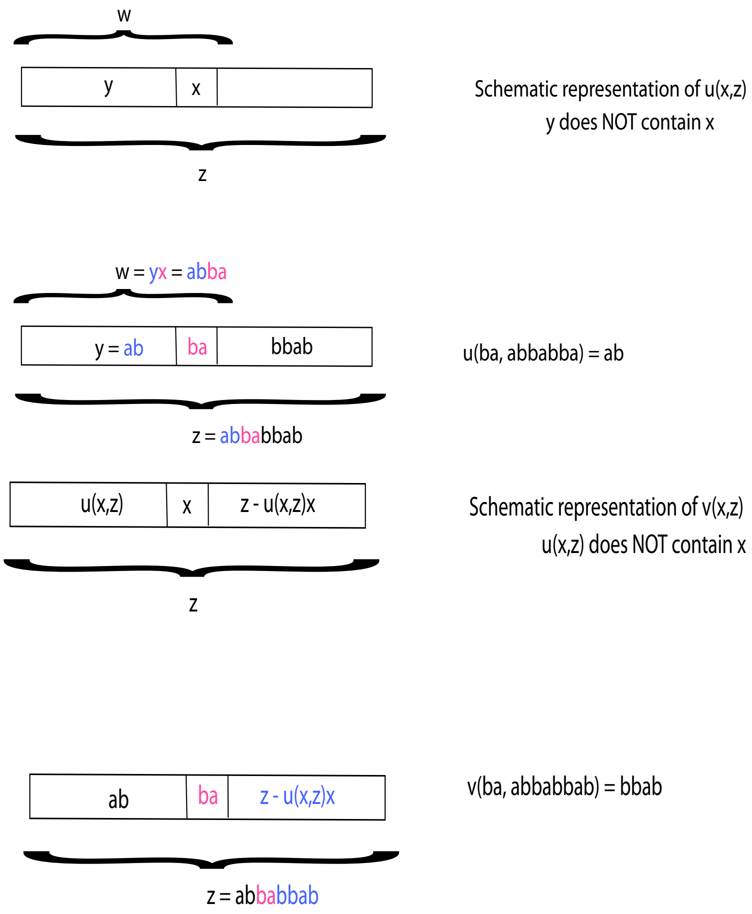


Figure 2.20: Illustration of the functions u and v.

(1) The function T is defined as follows, which shows that it is primitive recursive.

$$T(x, \epsilon) = x$$

$$T(x, ya_i) = \begin{cases} \mathrm{rep}(\ell_1, r_1, T(x,y)) & \text{iff } \mathrm{Occ}(\ell_1, T(x,y)) \\ \mathrm{rep}(\ell_2, r_2, T(x,y)) & \text{iff } \mathrm{Occ}(\ell_2, T(x,y)) \land \neg\mathrm{Occ}(\ell_1, T(x,y)) \\ \qquad\vdots & \\ \mathrm{rep}(\ell_N, r_N, T(x,y)) & \text{iff } \mathrm{Occ}(\ell_N, T(x,y)) \land \neg\mathrm{Occ}(\ell_1, T(x,y)) \\ & \land \cdots \land \neg\mathrm{Occ}(\ell_{N-1}, T(x,y)) \\ & T(x,y) \text{ otherwise.} \end{cases}$$

The function T is illustrated in Figure 2.22.

If $T(x, y)$ represents the ID $\#upav\#$ obtained after performing $|y|$ steps starting from the ID x, then $T(x, ya_i)$ represents the ID obtained by applying an instruction starting

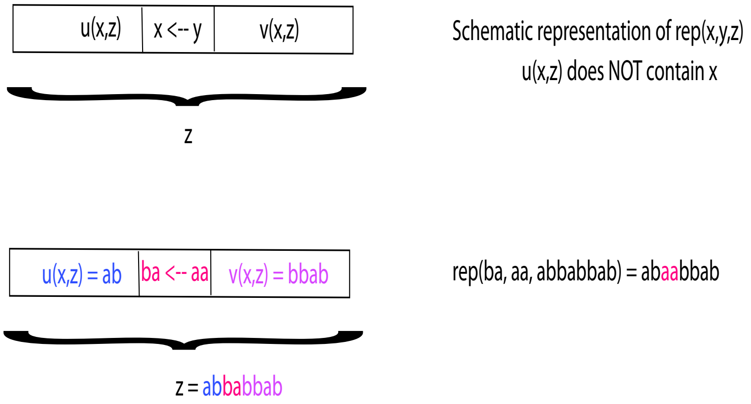

u(x,z)
x <-- y
v(x,z)
z
Schematic representation of rep(x,y,z)
u(x,z) does NOT contain x
u(x,z) = ab
ba <-- aa
v(x,z) = bbab
z = abbabbab
rep(ba, aa, abbabbab) = abaabbab

Figure 2.21: Illustration of the function rep.

with (p, a), if any. To see if such an instruction applies we test sequentially starting from $k = 1$ whether the left-hand side ℓ_k of a transition $\ell_k \to r_k$ occurs in $T(x, y)$, which is performed by $\mathrm{Occ}(\ell_k, T(x, y))$, the tests $\mathrm{Occ}(\ell_{k_1}, T(x, y))$ for all $k_1 < k$ being negative. If so, ℓ_k is replaced by r_k in the ID $T(x, y)$ to mimic the TM transition corresponding to $\ell_r \to r_k$, which is achieved by $\mathrm{rep}(\ell_k, r_k, T(x, y))$. Since the purpose of y is to count the number of steps, only $|y|$ matters, so we may assume that y is a string of a_1s.

(2) The function HALT is defined as follows.

$\mathrm{HALT}(x)$ iff $[\mathrm{Occ}(p_{i_1}a_{i_1}, x) \lor \cdots \lor \mathrm{Occ}(p_{i_m}a_{i_m}, x)]$.

(3) $\mathrm{STOP}(y, ID)$ iff $\mathrm{HALT}(T(x, y))$. □

If M is a Turing machine computing a function of n arguments $x_1, \ldots, x_n$, the starting ID is defined as

$$ID_0 = \#q_0x_1, x_2, \cdots, x_n\#$$

Let INIT be the function given by

$$\mathrm{INIT}(x_1, \ldots, x_n) = \#x_1, \ldots, x_n\#.$$

Obviously INIT is primitive recursive. Then for all $x_1, \ldots, x_n \in \Sigma^*$, we have

$$ID_0 \vdash^*_{|y|} ID \quad \text{and } ID \text{ is a halting ID}$$

iff

$$T(\mathrm{INIT}(x_1, \ldots, x_n), \mathrm{min}_1 y[\mathrm{STOP}(y, \mathrm{INIT}(x_1, \ldots, x_n))]) = ID.$$

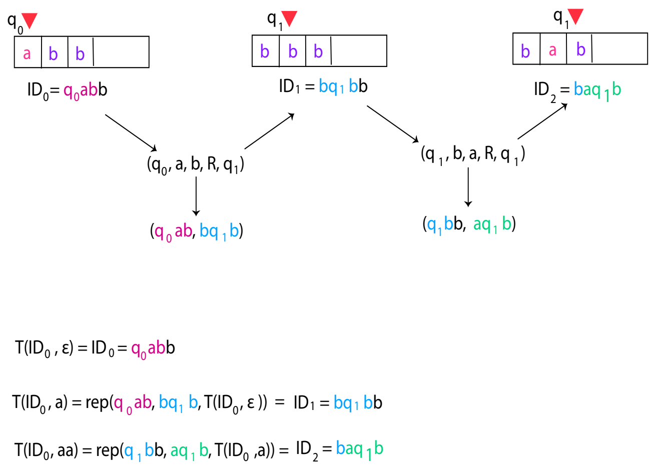


Figure 2.22: Illustration of the function T.

Let RES be the function that cleans up a halting ID to produce the output. The function RES is defined by primitive recursion as follows (recall that *rev* is the reverse function and *con* is the concatenation function).

$$\begin{aligned}
\mathrm{RES}(\epsilon) &= \epsilon \\
\mathrm{RES}(x\#) &= \mathrm{RES}(x) \\
\mathrm{RES}(xB) &= \mathrm{RES}(x) \\
\mathrm{RES}(xa_i) &= con(\mathrm{RES}(x), a_i), \quad 1 \leq i \leq k \\
\mathrm{RES}(xq) &= \mathrm{RES}(rev(x)), \quad q \in K.
\end{aligned}$$

We leave it as an exercise to prove that for any halting ID of the form $\#B^k quB^\ell\#$ with $u \in \Sigma^*$, we have

$$\mathrm{RES}(\#B^k quB^\ell\#) = u.$$

Combining all the facts we established we obtain the following result.

Theorem 2.6. *Every Turing computable function φ of n arguments is partial computable in the sense of Herbrand–Gödel–Kleene. Moreover, given a Turing machine M, we can effectively find a definition of φ of the form*

$$\varphi(x_1, \ldots, x_n) = \mathrm{RES}(T(\mathrm{INIT}(x_1, \ldots, x_n), \mathrm{min}_1 y[\mathrm{STOP}(y, \mathrm{INIT}(x_1, \ldots, x_n))])).$$

As a corollary we have the following nontrivial result.

Corollary 2.7. *Every partial computable function φ can be effectively obtained in the form $\varphi = f \circ \mathrm{min}_1\, g$, where f and g are primitive recursive functions.*

Consequently, every partial computable function has a definition in which minimization is applied at most once.

Chapter 3

The Lambda-Calculus

The original motivation of Alonzo Church for inventing the λ-calculus was to provide a type-free foundation for mathematics (alternate to set theory) based on higher-order logic and the notion of *function* in the early 1930's (1932, 1933). This attempt to provide such a foundation for mathematics failed due to a form of Russell's paradox. Church was clever enough to turn the technical reason for this failure, the existence of fixed-point combinators, into a success, namely to view the λ-calculus as a formalism for defining the notion of *computability* (1932,1933,1935). The λ-calculus is indeed one of the first computation models, slightly preceding the Turing machine.

Kleene proved in 1936 that all the computable functions (recursive functions) in the sense of Herbrand and Gödel are definable in the λ-calculus, showing that the λ-calculus has *universal computing power*. In 1937, Turing proved that Turing machines compute the same class of computable functions. (This paper is very hard to read, in part because the definition of a Turing machine is not included in this paper). In short, the λ-calculus and Turing machines have *the same computing power*. Here we have to be careful. To be precise we should have said that all the *total* computable functions (total recursive functions) are definable in the λ-calculus. In fact, it is also true that all the *partial* computable functions (partial recursive functions) are definable in the λ-calculus but this requires more care.

Since the λ-calculus does not have any notion of tape, register, or any other means of storing data, it quite amazing that the λ-calculus has so much computing power.

The λ-calculus is based on three concepts:

(1) Application.

(2) Abstraction (also called λ-abstraction).

(3) β-reduction (and β-conversion).

If f is a function, say the exponential function $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(n) = 2^n$, and if n a natural number, then the result of applying f to a natural number, say 5, is written as

$$(f\,5)$$

instead of $f(5)$, and is called an *application*. Here we can agree that f and 5 do not have the same *type*, in the sense that f is a function and 5 is a number, so applications such as $(f\ f)$ or $(5\,5)$ do not make sense, but the λ-calculus is *type-free* so expressions such as $(f\ f)$ as allowed. This may seem silly, and even possibly undesirable, but allowing self application turns out to a major reason for the computing power of the λ-calculus.

Given an expression M containing a variable x, say

$$M(x) = x^2 + x + 1,$$

as x ranges over $\mathbb{N}$, we obtain the function represented in standard mathematical notation by $x \mapsto x^2 + x + 1$. If we supply the input value 5 for x, then the value of the function is $5^2 + 5 + 1 = 31$. Church introduced the notation

$$\lambda x.\,(x^2 + x + 1)$$

for the function $x \mapsto x^2 + x + 1$. Here, we have an *abstraction*, in the sense that the static expression $M(x)$ for x fixed becomes an "abstract" function denoted $\lambda x.\,M$, where x can be instantiated to any input value.

It would be pointless to only have the two concepts of application and abstraction. The glue between these two notions is a form of evaluation called *β-reduction*.[1] Given a λ-abstraction $\lambda x.\,M$ and some other term N (thought of as an argument), we have the "evaluation" rule, we say *β-reduction*,

$$(\lambda x.\,M)N \xrightarrow{+}_{\beta} M[x := N],$$

where $M[x := N]$ denotes the result of substituting N for all occurrences of x in M. For example, if $M = x^2 + x + 1$ and $N = 2y + 1$, we have

$$(\lambda x.\,(x^2 + x + 1))(2y + 1) \xrightarrow{+}_{\beta} (2y + 1)^2 + 2y + 1 + 1.$$

Observe that β-reduction is a *purely formal* operation (plugging N wherever x occurs in M), and that the expression $(2y+1)^2+2y+1+1$ is *not* instantly simplified to $4y^2+6y+3$. In the λ-calculus, the natural numbers as well as the arithmetic operations $+$ and $\times$ need to be represented as λ-terms in such a way that they "evaluate" correctly using only β-conversion. In this sense, the λ-calculus is an incredibly low-level programming language. Nevertheless, the λ-calculus is the core of various *functional programming languages* such as *OCaml*, *ML*, *Miranda* and *Haskell*, among others.

We now proceed with precise definitions and results. But first we ask the reader not to think of functions as the functions we encounter in analysis or algebra. Instead think of functions as *rules for computing* (by moving and plugging arguments around), a more combinatory (which does not mean combinatorial) viewpoint.

This chapter relies heavily on the masterly expositions by Barendregt [3, 4]. We also found inspiration from very informative online material by Henk Barendregt, Peter Selinger, and J.R.B. Cockett, whom we thank. Hindley and Seldin [22] and Krivine [26] are also excellent sources (and not as advanced as Barendregt [3]).

[1] Apparently, Church was fond of Greek letters.

3.1 Syntax of the Lambda-Calculus

We begin by defining the *lambda-calculus*, also called *untyped lambda-calculus* or *pure lambda-calculus*, to emphasize that the terms of this calculus are not typed. This formal system consists of

1. A set of terms, called λ-*terms*.

2. A notion of reduction, called β-*reduction*, which allows a term M to be transformed into another term N in a way that mimics a kind of evaluation.

First we define (pure) λ-terms. We have a countable set of variables $\{x_0, x_1, \ldots, x_n \ldots\}$ that correspond to the atomic λ-terms.

Definition 3.1. The λ-*terms* M are defined inductively as follows:

(1) If x_i is a variable, then x_i is a λ-term.

(2) If M and N are λ-terms, then (MN) is a λ-term called an *application*.

(3) If M is a λ-term, and x is a variable, then the expression $(\lambda x.\, M)$ is a λ-term called a λ-*abstraction*.

Note that the only difference between the λ-terms of Definition 3.1 and the raw simply-typed λ-terms of Definition 3.7 is that in Clause (3), in a λ-abstraction term $(\lambda x.\, M)$, the variable x occurs without any type information, whereas in a simply-typed λ-abstraction term $(\lambda x\colon \sigma.\, M)$, the variable x is assigned the type σ. At this stage this is only a cosmetic difference because raw λ-terms are not yet assigned types. But there are type-checking rules for assigning types to raw simply-typed λ-terms that *restrict application*, so the set of simply-typed λ-terms that type-check is much more restricted than the set of (untyped) λ-terms. In particular, no simply-typed λ-term that type-checks can be a self-application (MM). The fact that self-application is allowed in the untyped λ-calculus is what gives it its computational power (through fixed-point combinators, see Section 3.5).

Definition 3.2. The *depth* $d(M)$ of a λ-term M is defined inductively as follows.

1. If M is a variable x, then $d(x) = 0$.

2. If M is an application (M_1M_2), then $d(M) = \max\{d(M_1), d(M_2)\} + 1$.

3. If M is a λ-abstraction $(\lambda x.\, M_1)$, then $d(M) = d(M_1) + 1$.

It is pretty clear that λ-terms have representations as (ordered) labeled trees.

Definition 3.3. Given a λ-term M, the *tree* tree(M) representing M is defined inductively as follows:

1. If M is a variable x, then tree(M) is the one-node tree labeled x.

2. If M is an application (M_1M_2), then tree(M) is the tree with a binary root node labeled $.$, and with a left subtree tree(M_1) and a right subtree tree(M_2).

3. If M is a λ-abstraction $(\lambda x.\, M_1)$, then tree(M) is the tree with a unary root node labeled λx, and with one subtree tree(M_1).

Definition 3.3 is illustrated in Figure 3.1.

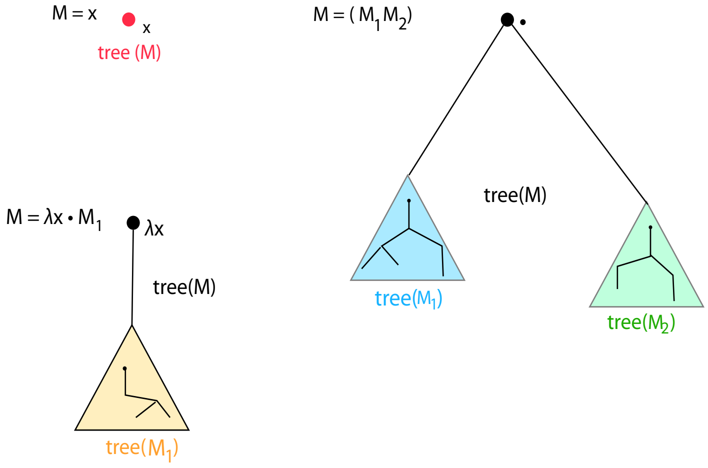


Figure 3.1: The tree tree(M) associated with a pure λ-term M.

Obviously, the depth $d(M)$ of λ-term is the depth of its tree representation tree(M).

Unfortunately λ-terms contain a profusion of parentheses so some conventions are commonly used.

(1) A term of the form
$$(\cdots((FM_1)M_2)\cdots M_n)$$
is abbreviated (association to the left) as
$$FM_1\cdots M_n.$$

(2) A term of the form
$$(\lambda x_1.\,(\lambda x_2.\,(\cdots(\lambda x_n.\,M)\cdots)))$$
is abbreviated (association to the right) as
$$\lambda x_1\cdots x_n.\,M.$$

Matching parentheses may be dropped or added for convenience.

Example 3.1. Here are some examples of λ-terms (and their abbreviation):

$$\begin{array}{ll}
y & y \\
(yx) & yx \\
(\lambda x.\,(yx)) & \lambda x.\,yx \\
((\lambda x.\,(yx))z) & (\lambda x.\,yx)z \\
(((\lambda x.\,(\lambda y.\,(yx)))z)w) & (\lambda xy.\,yx)zw.
\end{array}$$

Note that $\lambda x.\,yx$ is an abbreviation for $(\lambda x.\,(yx))$, not $((\lambda x.\,y)x)$ (which is abbreviated as $(\lambda x.\,y)x$).

The variables occurring in a λ-term are free or bound.

Definition 3.4. For any λ-term M, the set $FV(M)$ of *free variables* of M and the set $BV(M)$ of *bound variables* in M are defined inductively as follows.

(1) If $M = x$ (a variable), then

$$FV(x) = \{x\}, \quad BV(x) = \emptyset.$$

(2) If $M = (M_1M_2)$, then

$$FV(M) = FV(M_1) \cup FV(M_2), \quad BV(M) = BV(M_1) \cup BV(M_2).$$

(3) if $M = (\lambda x.\,M_1)$, then

$$FV(M) = FV(M_1) - \{x\}, \quad BV(M) = BV(M_1) \cup \{x\}.$$

If $x \in FV(M_1)$, we say that the occurrences of the variable x *occur in the scope of* λ.

A λ-term M is *closed* or a *combinator* if $FV(M) = \emptyset$, that is, if it has no free variables.

Example 3.2. We have

$$FV\big((\lambda x.\,yx)z\big) = \{y, z\}, \qquad BV\big((\lambda x.\,yx)z\big) = \{x\},$$

and

$$FV\big((\lambda xy.\,yx)zw\big) = \{z, w\}, \qquad BV\big((\lambda xy.\,yx)zw\big) = \{x, y\}.$$

Before proceeding with the notion of substitution we must address an issue with bound variables. The point is that bound variables are really *place-holders* so they can be renamed freely without changing the reduction behavior of the term as long as they do not clash with free variables. For example, the terms $\lambda x.\,(x(\lambda y.\,x(yx))$ and $\lambda x.\,(x(\lambda z.\,x(zx))$ should be considered as equivalent. Similarly, the terms $\lambda x.\,(x(\lambda y.\,x(yx))$ and $\lambda w.\,(w(\lambda z.\,w(zw))$ should be considered as equivalent.

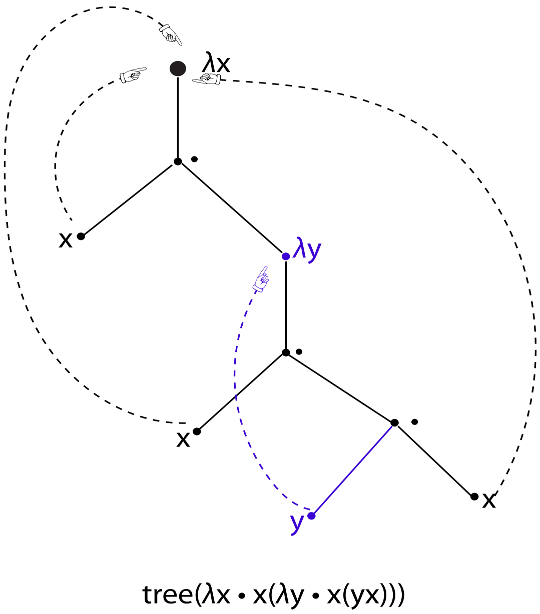


Figure 3.2: The tree representation of a λ-term with backpointers.

One way to deal with this issue is to use the tree representation of λ-terms given in Definition 3.3. For every leaf labeled with a bound variable x, we draw a backpointer to an ancestor of x determined as follows. Given a leaf labeled with a bound variable x, climb up to the closest ancestor labeled λx, and draw a backpointer to this node. Then all bound variables can be erased. An example is shown in Figure 3.2 for the term $M = \lambda x.\, x(\lambda y.\, (x(yx)))$.

A clever implementation of the idea of backpointers is the formalism of *de Bruijn indices*; see Pierce [33] (Chapter 6) or Barendregt [3] (Appendix C).

Church introduced the notion of *α-conversion* to deal with this issue. First we need to define substitutions.

Definition 3.5. A *substitution* φ is a finite set of pairs $\varphi = \{(x_1, N_1), \ldots, (x_n, N_n)\}$, where the x_i are distinct variables and the N_i are λ-terms. We write

$$\varphi = [N_1/x_1, \ldots, N_n/x_n] \quad \text{or} \quad \varphi = [x_1 := N_1, \ldots, x_n := N_n].$$

The second notation indicates more clearly that each term N_i is substituted for the variable x_i, and it seems to have been almost universally adopted.

Definition 3.6. Given a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, for any variable x_i, we denote by φ_{-x_i} the new substitution where the pair (x_i, N_i) is replaced by the pair (x_i, x_i) (that is, the new substitution leaves x_i unchanged).

Definition 3.7. Given any λ-term M and any substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, we define the λ-term $M[\varphi]$, *the result of applying the substitution* φ *to* M, as follows.

(1) If $M = y$, with $y \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = y = M$.

(2) If $M = x_i$ for some $i \in \{1, \ldots, n\}$, then $M[\varphi] = N_i$.

(3) If $M = (PQ)$, then $M[\varphi] = (P[\varphi]Q[\varphi])$.

(4) If $M = \lambda x.\, N$ and $x \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = \lambda x.\, N[\varphi]$,

(5) If $M = \lambda x.\, N$ and $x = x_i$ for some $i \in \{1, \ldots, n\}$, then $M[\varphi] = \lambda x.\, N[\varphi]_{-x_i}$.

The term M is *safe* for the substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$ if $BV(M) \cap (FV(N_1) \cup \cdots \cup FV(N_n)) = \emptyset$, that is, if the free variables in the substitution *do not* become bound.

Note that Clause (5) ensures that a substitution *only substitutes the terms* N_i *for the variables* x_i *free in* M. Thus if M is a *closed* term, then for *every* substitution φ, we have $M[\varphi] = M$.

Example 3.3. Here are some examples of substitution.

$$\begin{aligned}
y[x := \lambda x.\, (xz)(xz)] &= y \\
x[x := \lambda x.\, (xz)(xz)] &= \lambda x.\, (xz)(xz) \\
(xz)(yz)[y := (vv); z := (\lambda u.\, v)] &= (x(\lambda u.\, v))((vv)(\lambda u.\, v)) \\
\lambda x.\, (xz)(yz)[y := (vv); z := (\lambda u.\, v)] &= \lambda x.\, (x(\lambda u.\, v))((vv)(\lambda u.\, v)) \\
\lambda z.\, (z(xz))[x := (\lambda u.\, (uu)); z = (uu)] &= \lambda z.\, (z((\lambda u.\, (uu))z)).
\end{aligned}$$

There is a problem with the present definition of a substitution in Cases (4) and (5), which is that the result of substituting a term N_i containing the variable x free causes this variable to become bound after the substitution. We say that x is *captured*.

Example 3.4. If we make the substitution

$$\lambda x.\, (xz)(yz)[y := (xx); z := (\lambda u.\, v)] = \lambda x.\, (x(\lambda u.\, v))((xx)(\lambda u.\, v)),$$

the variable x occurring free in the term (xx) now has three bound occurrences in the term $\lambda x.\, (x(\lambda u.\, v))((xx)(\lambda u.\, v))$. We should only apply a substitution φ to a term M if M is safe for φ. We should rename the bound variable x in the term $\lambda x.\, (xz)(yz)$, say as w, obtaining the term $\lambda w.\, (wz)(yz)$, and then there is no capture of variable when we make the substitution

$$\lambda w.\, (wz)(yz)[y := (xx); z := (\lambda u.\, v)] = \lambda w.\, (w(\lambda u.\, v))((xx)(\lambda u.\, v)).$$

To remedy this problem, Church defined *α-conversion*.

Definition 3.8. The binary relation $\longrightarrow_\alpha$ on λ-terms called *immediate α-conversion*[2] is the smallest relation satisfying the following properties: for all λ-terms M, N, P, Q.

$$\lambda x.\, M \longrightarrow_\alpha \lambda y.\, M[x := y], \quad \text{for all } y \notin FV(M) \cup BV(M)$$

$$\begin{aligned}
&\text{if} \quad M \longrightarrow_\alpha N \quad \text{then} \quad MQ \longrightarrow_\alpha NQ \quad \text{and} \quad PM \longrightarrow_\alpha PN \\
&\text{if} \quad M \longrightarrow_\alpha N \quad \text{then} \quad \lambda x.\, M \longrightarrow_\alpha \lambda x.\, N.
\end{aligned}$$

The least equivalence relation $\equiv_\alpha = (\longrightarrow_\alpha \cup \longrightarrow_\alpha^{-1})^*$ containing $\longrightarrow_\alpha$ (the reflexive and transitive closure of $\longrightarrow_\alpha \cup \longrightarrow_\alpha^{-1}$) is called *$\alpha$-conversion*. Here $\longrightarrow_\alpha^{-1}$ denotes the converse of the relation $\longrightarrow_\alpha$, that is, $M \longrightarrow_\alpha^{-1} N$ iff $N \longrightarrow_\alpha M$.

Example 3.5. We have

$$\lambda fx.\, f(f(x)) = \lambda f.\, \lambda x.\, f(f(x)) \longrightarrow_\alpha \lambda f.\, \lambda y.\, f(f(y)) \longrightarrow_\alpha \lambda g.\, \lambda y.\, g(g(y)) = \lambda gy.\, g(g(y)).$$

Now given a λ-term M and a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, before applying φ to M we *first perform some α-conversion* to obtain a term $M' \equiv_\alpha M$ whose set of bound variables $BV(M')$ is disjoint from $FV(N_1) \cup \cdots \cup FV(N_n)$ so that M' *is safe for* φ, and the result of the substitution is $M'[\varphi]$.

Example 3.6. We have

$$\big(\lambda yz.\, (xy)z\big)[x := yz] \equiv_\alpha \big(\lambda uv.\, (xu)v\big)[x := yz] = \lambda uv.\, ((yz)u)v.$$

From now on, we consider two λ-terms M and M' such that $M \equiv_\alpha M'$ as identical (to be rigorous, we deal with equivalence classes of terms with respect to α-conversion). Even the experts are lax about α-conversion so we happily go along with them. The convention is that *bound variables are always renamed to avoid clashes* (with free or bound variables).

Note that the representation of λ-terms as trees with back-pointers also ensures that substitutions are safe. However, this requires some extra effort. No matter what, it takes some effort to deal properly with bound variables.

3.2 β-Reduction and β-Conversion; the Church–Rosser Theorem

The computational engine of the λ-calculus is β-reduction.

Definition 3.9. The relation $\longrightarrow_\beta$, called *immediate β-reduction*, is the smallest relation satisfying the following properties for all λ-terms M, N, P, Q:

$$(\lambda x.\, M)N \longrightarrow_\beta M[x := N], \quad \text{where } M \text{ is safe for } [x := N]$$

[2] We told you that Church was fond of Greek letters.

$$\begin{aligned}&\text{if} \quad M \longrightarrow_\beta N, \quad \text{then} \quad MQ \longrightarrow_\beta NQ \quad \text{and} \quad PM \longrightarrow_\beta PN\\ &\text{if} \quad M \longrightarrow_\beta N, \quad \text{then} \quad \lambda x.\, M \longrightarrow_\beta \lambda x.\, N.\end{aligned}$$

The transitive closure of $\longrightarrow_\beta$ is denoted by $\xrightarrow{+}_\beta$, the reflexive and transitive closure of $\longrightarrow_\beta$ is denoted by $\xrightarrow{*}_\beta$, and we define *β-conversion*, denoted by $\xleftrightarrow{*}_\beta$, as the smallest equivalence relation $\xleftrightarrow{*}_\beta = (\longrightarrow_\beta \cup \longrightarrow_\beta^{-1})^*$ containing $\longrightarrow_\beta$. A subterm of the form $(\lambda x.\, M)N$ occurring in another term is called a *β-redex*. A λ-term M is a *β-normal form* if there is no λ-term N such that $M \longrightarrow_\beta N$, equivalently if M contains *no* β-redex.

Example 3.7. The subterm $\big((\lambda x.\, (\lambda y.\, x))u\big)$ is a β-redex in the term $\big((\lambda x.\, (\lambda y.\, x))u\big)v$. We have

$$(\lambda xy.\, x)uv = \big((\lambda x.\, (\lambda y.\, x))u\big)v \longrightarrow_\beta ((\lambda y.\, x)[x := u])v = (\lambda y.\, u)v \longrightarrow_\beta u[y := v] = u.$$

The subterm $\big((\lambda x.\, (\lambda y.\, y))u\big)$ is a β-redex in the term $\big((\lambda x.\, (\lambda y.\, y))u\big)v$. We have

$$(\lambda xy.\, y)uv = \big((\lambda x.\, (\lambda y.\, y))u\big)v \longrightarrow_\beta ((\lambda y.\, y)[x := u])v = (\lambda y.\, y)v \longrightarrow_\beta y[y := v] = v.$$

This shows that $\lambda xy.\, x$ behaves like the projection onto the first argument and $\lambda xy.\, y$ behaves like the projection onto the second.

Example 3.8. More interestingly, if we let $\boldsymbol{\omega} = \lambda x.\, (xx)$, then

$$\boldsymbol{\Omega} = \boldsymbol{\omega}\boldsymbol{\omega} = (\lambda x.\, (xx))(\lambda x.\, (xx)) \longrightarrow_\beta (xx)[x := \lambda x.\, (xx)] = \boldsymbol{\omega}\boldsymbol{\omega} = \boldsymbol{\Omega}.$$

The above example shows that β-reduction sequences may be infinite. This is a curse and a miracle of the λ-calculus!

Example 3.9. There are even β-reductions where the evolving term grows in size.

$$\begin{aligned}(\lambda x.\, xxx)(\lambda x.\, xxx) &\xrightarrow{+}_\beta (\lambda x.\, xxx)(\lambda x.\, xxx)(\lambda x.\, xxx)\\ &\xrightarrow{+}_\beta (\lambda x.\, xxx)(\lambda x.\, xxx)(\lambda x.\, xxx)(\lambda x.\, xxx)\\ &\xrightarrow{+}_\beta \cdots\end{aligned}$$

Example 3.10. The term

$$M = \Bigg(\lambda z.\, \Big(\big((\lambda x.\, (xz))u\big)\big((\lambda y.\, (yz))v\big)\Big)\Bigg)w$$

has three β-redexes:

(1) The term M itself.

(2) The subterm $\big((\lambda x.\, (xz))u\big)$.

(3) The subterm $\big((\lambda y.\,(yz))v\big)$.

Example 3.11. The term

$$M = \Big(\lambda z.\,\big((\lambda y.\,(zyy))z\big)\Big)w$$

has two β-redexes:

(1) The term M itself.

(2) The subterm $\big((\lambda y.\,(zyy))z\big)$.

In general, a λ-term contains many different β-redex. One then might wonder if there is any sort of relationship between any two terms M_1 and M_2 arising through two β-reduction sequences $M \xrightarrow{*}_\beta M_1$ and $M \xrightarrow{*}_\beta M_2$ starting with the same term M. The answer is given by the following famous theorem.

Theorem 3.1. *(Church–Rosser Theorem) The following two properties hold.*

(1) The λ-calculus is **confluent***: for any three λ-terms M, M_1, M_2, if $M \xrightarrow{*}_\beta M_1$ and $M \xrightarrow{*}_\beta M_2$, then there is some λ-term M_3 such that $M_1 \xrightarrow{*}_\beta M_3$ and $M_2 \xrightarrow{*}_\beta M_3$. See Figure 3.3.*

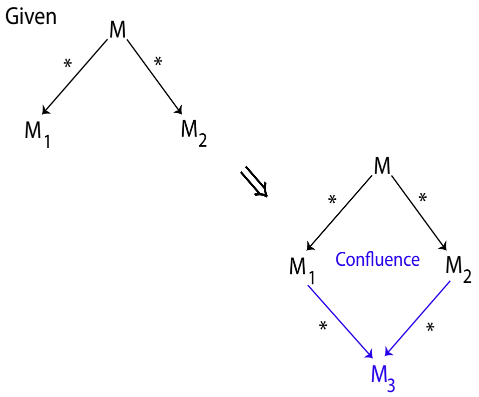


Figure 3.3: The confluence property.

(2) The λ-calculus has the **Church–Rosser property***: for any two λ-terms M_1, M_2, if $M_1 \xleftrightarrow{*}_\beta M_2$, then there is some λ-term M_3 such that $M_1 \xrightarrow{*}_\beta M_3$ and $M_2 \xrightarrow{*}_\beta M_3$. See Figure 3.4.*

Furthermore (1) and (2) are equivalent, and if a λ-term M β-reduces to a β-normal form N, then N is unique (up to α-conversion).

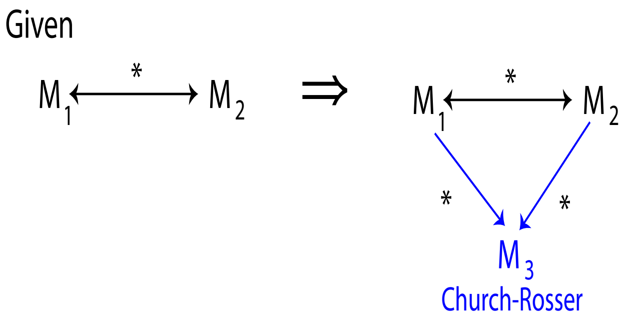


Figure 3.4: The Church–Rosser property.

Proof. We are not aware of any easy proof of Part (1) or Part (2) of Theorem 3.1, but the equivalence of (1) and (2) is easily shown by induction.

Assume that (2) holds. Since $\xrightarrow{*}_\beta$ is contained in $\overset{*}{\longleftrightarrow}_\beta$, if $M \xrightarrow{*}_\beta M_1$ and $M \xrightarrow{*}_\beta M_2$, then $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$, and since (2) holds, then there is some λ-term M_3 such that $M_1 \xrightarrow{*}_\beta M_3$ and $M_2 \xrightarrow{*}_\beta M_3$, which is (1).

To prove that (1) implies (2) we need the following observation.

Since $\overset{*}{\longleftrightarrow}_\beta = (\longrightarrow_\beta \cup \longrightarrow_\beta^{-1})^*$, we see immediately that $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$ iff either

(a) $M_1 = M_2$, or

(b) there is some M_3 such that $M_1 \longrightarrow_\beta M_3$ and $M_3 \overset{*}{\longleftrightarrow}_\beta M_2$, or

(c) there is some M_3 such that $M_3 \longrightarrow_\beta M_1$ and $M_3 \overset{*}{\longleftrightarrow}_\beta M_2$.

Assume (1). We proceed by induction on the number of steps in $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$. If $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$, as discussed before, there are three cases.

Case a. Base case, $M_1 = M_2$. Then (2) holds with $M_3 = M_1 = M_2$.

Case b. There is some M_3 such that $M_1 \longrightarrow_\beta M_3$ and $M_3 \overset{*}{\longleftrightarrow}_\beta M_2$. Since $M_3 \overset{*}{\longleftrightarrow}_\beta M_2$ contains one less step than $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$, by the induction hypothesis there is some M_4 such that $M_3 \xrightarrow{*}_\beta M_4$ and $M_2 \xrightarrow{*}_\beta M_4$, and then $M_1 \longrightarrow_\beta M_3 \xrightarrow{*}_\beta M_4$ and $M_2 \xrightarrow{*}_\beta M_4$, proving (2). See Figure 3.5.

Case c. There is some M_3 such that $M_3 \longrightarrow_\beta M_1$ and $M_3 \overset{*}{\longleftrightarrow}_\beta M_2$. Since $M_3 \overset{*}{\longleftrightarrow}_\beta M_2$ contains one less step than $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$, by the induction hypothesis there is some M_4 such that $M_3 \xrightarrow{*}_\beta M_4$ and $M_2 \xrightarrow{*}_\beta M_4$. Now $M_3 \longrightarrow_\beta M_1$ and $M_3 \xrightarrow{*}_\beta M_4$, so by (1) there is some M_5 such that $M_1 \xrightarrow{*}_\beta M_5$ and $M_4 \xrightarrow{*}_\beta M_5$. Putting derivations together we get $M_1 \xrightarrow{*}_\beta M_5$ and $M_2 \xrightarrow{*}_\beta M_4 \xrightarrow{*}_\beta M_5$, which proves (2). See Figure 3.6.

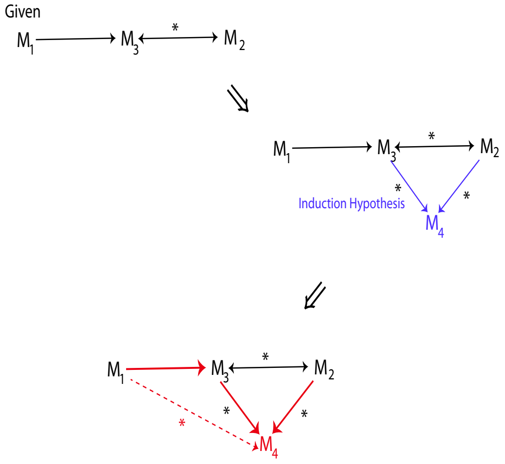


Figure 3.5: Case b.

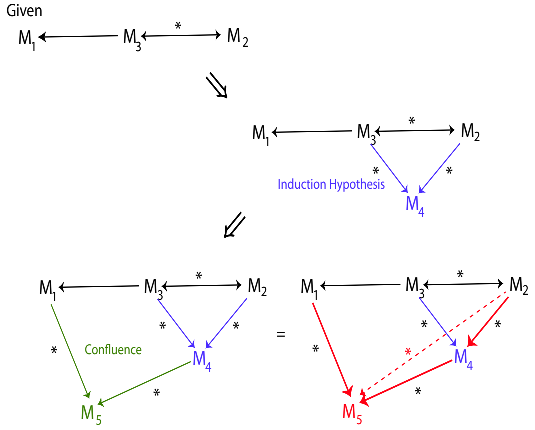


Figure 3.6: Case c.

Suppose $M \xrightarrow{*}_\beta N_1$ and $M \xrightarrow{*}_\beta N_2$ where N_1 and N_2 are both β-normal forms. Then by confluence there is some N such that $N_1 \xrightarrow{*}_\beta N$ and $N_2 \xrightarrow{*}_\beta N$. Since N_1 and N_2 are both β-normal forms, we must have $N_1 = N = N_2$ (up to α-conversion).

Barendregt gives an elegant proof of the confluence property in [3] (Chapter 11). □

Another immediate corollary of the Church–Rosser theorem is that if $M \overset{*}{\longleftrightarrow}_\beta N$ and if N is a β-normal form, then in fact $M \xrightarrow{*}_\beta N$. We leave this fact as an exerise

This fact will be useful in showing that the recursive functions are computable in the λ-calculus.

Example 3.12. Consider the term

$$M = \Big(\lambda z.\, \Big(\big((\lambda x.\, (xz))u\big)\big((\lambda y.\, (yz))v\big)\Big)\Big)w.$$

We have the reduction sequences

$$\begin{aligned} M = \Big(\lambda z.\, \Big(\big((\lambda x.\, (xz))u\big)\big((\lambda y.\, (yz))v\big)\Big)\Big)w &\longrightarrow_\beta \Big(\lambda z.\, \Big((uz)\big((\lambda y.\, (yz))v\big)\Big)\Big)w \\ &\longrightarrow_\beta \Big(\lambda z.\, \big((uz)(vz)\big)\Big)w = M_1 \end{aligned}$$

and

$$\begin{aligned} M &= \Big(\lambda z.\, \Big(\big((\lambda x.\, (xz))u\big)\big((\lambda y.\, (yz))v\big)\Big)\Big)w \\ &\longrightarrow_\beta \Big(\big((\lambda x.\, (xw))u\big)\big((\lambda y.\, (yw))v\big)\Big) = M_2. \end{aligned}$$

Confluence holds because

$$M_1 = \Big(\lambda z.\, \big((uz)(vz)\big)\Big)w \longrightarrow_\beta ((uw)(vw)) = M_3$$

and

$$\begin{aligned} M_2 = \Big(\big((\lambda x.\, (xw))u\big)\big((\lambda y.\, (yw))v\big)\Big) &\longrightarrow_\beta \Big((uw)\big((\lambda y.\, (yw))v\big)\Big) \\ &\longrightarrow_\beta ((uw)(vw)) = M_3. \end{aligned}$$

Example 3.13. Consider the term

$$M = \Big(\lambda z.\, \big((\lambda y.\, (zyy))z\big)\Big)w.$$

We have the reductions

$$M = \Big(\lambda z.\, \big((\lambda y.\, (zyy))z\big)\Big)w \longrightarrow_\beta \big(\lambda z.\, (zzz)\big)w = M_1$$

and

$$M = \Big(\lambda z.\, \big((\lambda y.\, (zyy))z\big)\Big)w \longrightarrow_\beta \big((\lambda y.\, (wyy))w\big) = M_2.$$

We have confluence because

$$M_1 = \big(\lambda z.\, (zzz)\big)w \longrightarrow_\beta www = M_3$$

and

$$M_2 = \big((\lambda y.\, (wyy))w\big) \longrightarrow_\beta www = M_3.$$

3.3 Some Useful Combinators

In this section we provide some evidence for the expressive power of the λ-calculus.

First we make a remark about the representation of functions of several variables in the λ-calculus. The λ-calculus makes the implicit assumption that a function has a single argument. This is the idea behind application: given a term M viewed as a function and an argument N, the term (MN) represents the result of applying M to the argument N, *except that the actual evaluation is suspended*. Evaluation is performed by β-conversion. To deal with functions of several arguments we use a method known as *Currying* (after Haskell Curry). In this method, a function of n arguments is viewed as a function of one argument *taking a function of* $n-1$ *arguments as argument*. Consider the case of two arguments, the general case being similar. Consider a function $f\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$. For any fixed x, we define the function $F_x\colon \mathbb{N}\to\mathbb{N}$ given by

$$F_x(y) = f(x,y) \qquad y\in\mathbb{N}.$$

Using the λ-notation we can write

$$F_x = \lambda y.\, f(x,y),$$

and then the function $x\mapsto F_x$, which is a function from $\mathbb{N}$ to the *set of functions* $[\mathbb{N}\to\mathbb{N}]$ (also denoted $\mathbb{N}^{\mathbb{N}}$), is denoted by the λ-term

$$F = \lambda x.\, F_x = \lambda x.\, (\lambda y.\, f(x,y)).$$

And indeed,

$$(FM)N \xrightarrow{+}_{\beta} F_M N \xrightarrow{+}_{\beta} f(M,N).$$

Remark: Currying is a way to realizing the isomorphism between the sets of functions $[\mathbb{N}\times\mathbb{N}\to\mathbb{N}]$ and $[\mathbb{N}\to[\mathbb{N}\to\mathbb{N}]]$ (or in the standard set-theoretic notation, between $\mathbb{N}^{\mathbb{N}\times\mathbb{N}}$ and $(\mathbb{N}^{\mathbb{N}})^{\mathbb{N}}$. Does this remind you of the identity

$$(m^n)^p = m^{np}?$$

It should.

The function space $[\mathbb{N}\to\mathbb{N}]$ is called an *exponential*. There is a very abstract way to view all this which is to say that we have an instance of a Cartesian closed category (CCC).

Proposition 3.2. *If* $\mathbf{I}, \mathbf{K}, \mathbf{K}_*$, *and* $\mathbf{S}$ *are the combinators defined by*

$$\begin{aligned}
\mathbf{I} &= \lambda x.\, x\\
\mathbf{K} &= \lambda xy.\, x\\
\mathbf{K}_* &= \lambda xy.\, y\\
\mathbf{S} &= \lambda xyz.\, (xz)(yz),
\end{aligned}$$

then for all λ*-terms* M, N, P*, we have*

$$\begin{aligned}
\mathbf{I}M &\xrightarrow{+}_{\beta} M\\
\mathbf{K}MN &\xrightarrow{+}_{\beta} M\\
\mathbf{K}_{*}MN &\xrightarrow{+}_{\beta} N\\
\mathbf{S}MNP &\xrightarrow{+}_{\beta} (MP)(NP)\\
\mathbf{KI} &\xrightarrow{+}_{\beta} \mathbf{K}_{*}\\
\mathbf{SKK} &\xrightarrow{+}_{\beta} \mathbf{I}.
\end{aligned}$$

Proof. The first equation is trivial and the second and third equations were shown in Example 3.7. The fourth equation is shown as follows. We have

$$\begin{aligned}
\mathbf{S}MNP = \big(\lambda xyz.\,(xz)(yz)\big)MNP &\longrightarrow_{\beta} \big(\big(\lambda yz.\,(xz)(yz)\big)[x := M]\big)NP\\
&= \big(\lambda yz.\,(Mz)(yz)\big)NP\\
&\longrightarrow_{\beta} \big(\big(\lambda z.\,(Mz)(yz)\big)[y := N]\big)P\\
&= \big(\lambda z.\,(Mz)(Nz)\big)P\\
&\longrightarrow_{\beta} \big((Mz)(Nz)\big)[z := P] = (MP)(NP).
\end{aligned}$$

The last equation is shown as follows.

$$\begin{aligned}
\mathbf{SKK} &= \big(\lambda xyz.\,(xz)(yz)\big)\mathbf{KK}\\
&\longrightarrow_{\beta} \big(\big(\lambda yz.\,(xz)(yz)\big)[x := \mathbf{K}]\big)\mathbf{K}\\
&= \big(\big(\lambda yz.\,(\mathbf{K}z)(yz)\big)\mathbf{K}\\
&\longrightarrow_{\beta} \big(\big(\lambda z.\,(\mathbf{K}z)(yz)\big)[y := \mathbf{K}]\\
&= \lambda z.\,(\mathbf{K}z)(\mathbf{K}z)\\
&\xrightarrow{+}_{\beta} \lambda z.\,z = \mathbf{I}.
\end{aligned}$$

The fitfth equation is left as an exercise. □

The need for a conditional construct if then else such that if $\mathbf{T}$ then P else Q yields P and if $\mathbf{F}$ then P else Q yields Q is indispensable to write nontrivial programs. There is a trick to encode the boolean values $\mathbf{T}$ and $\mathbf{F}$ in the λ-calculus to mimick the above behavior of if B then P else Q, provided that B is a truth value. Since everything in the λ-calculus is a function, the booleans values $\mathbf{T}$ and $\mathbf{F}$ are encoded as λ-terms. At first, this seems quite odd, but what counts is the behavior of if B then P else Q, and it works!

The truth values $\mathbf{T}, \mathbf{F}$ and the conditional construct if B then P else Q can be encoded in the λ-calculus as follows.

Proposition 3.3. *Consider the combinators given by* $\mathbf{T} = \mathbf{K}, \mathbf{F} = \mathbf{K}_{*}$*, and*

$$\textsf{if then else} = \lambda b.\,(\lambda x.\,(\lambda y.\,(bx)y)) = \lambda bxy.\,bxy.$$

Then for all λ-terms M, P, Q we have

$$\textsf{if } M \textsf{ then } P \textsf{ else } Q = (((\textsf{if then else})M)P)Q \xrightarrow{+}_{\beta} (MP)Q = MPQ.$$

In particular,

$$\begin{aligned}\textsf{if } \mathbf{T} \textsf{ then } P \textsf{ else } Q &\xrightarrow{+}_{\beta} P \\ \textsf{if } \mathbf{F} \textsf{ then } P \textsf{ else } Q &\xrightarrow{+}_{\beta} Q.\end{aligned}$$

Proof. We have

$$\begin{aligned}\textsf{if } M \textsf{ then } P \textsf{ else } Q &= (((\textsf{if then else})M)P)Q \\ &= \big(\big(\big(\lambda b.\,(\lambda x.\,(\lambda y.\,(bx)y))\big)M\big)P\big)Q \\ &\longrightarrow_{\beta} \big(\big(\big(\lambda x.\,(\lambda y.\,(bx)y)\big)[b := M]\big)P\big)Q = \big(\big(\lambda x.\,(\lambda y.\,(Mx)y)\big)P\big)Q \\ &\longrightarrow_{\beta} \big(\big(\lambda y.\,(Mx)y\big)[x := P]\big)Q = \big(\lambda y.\,(MP)y\big)\big)Q \\ &\longrightarrow_{\beta} ((MP)y)[y := Q] = (MP)Q.\end{aligned}$$

The other two reductions follow by Proposition 3.2. □

The boolean operations $\wedge, \vee, \neg$ can be defined in terms of if then else. For example,

$$\begin{aligned}\textsf{Not } b &= \textsf{if } b \textsf{ then } \mathbf{F} \textsf{ else } \mathbf{T} \\ \textsf{And } b_1 b_2 &= \textsf{if } b_1 \textsf{ then } (\textsf{if } b_2 \textsf{ then } \mathbf{T} \textsf{ else } \mathbf{F}) \textsf{ else } \mathbf{F} \\ \textsf{Or } b_1 b_2 &= \textsf{if } b_1 \textsf{ then } \mathbf{T} \textsf{ else } (\textsf{if } b_2 \textsf{ then } \mathbf{T} \textsf{ else } \mathbf{F}).\end{aligned}$$

Remark: If B is a term different from $\mathbf{T}$ or $\mathbf{F}$, then if $\mathbf{B}$ then P else Q may not reduce at all, or reduce to something different from P or Q. The problem is that the conditional statement that we designed only works properly if the input B is of the correct type, namely a boolean. If we give garbage as input, then we can't expect a correct result. The λ-calculus being type-free, it is unable to check for the validity of the input. In this sense this is a defect, but it also accounts for its power.

The ability to construct ordered pairs is also crucial.

Proposition 3.4. *For any two λ-terms M and N consider the combinator $\langle M, N\rangle$ and the combinator π_1 and π_2 given by*

$$\begin{aligned}\langle M, N\rangle &= \lambda z.\, zMN = \lambda z.\,\textsf{if } z \textsf{ then } M \textsf{ else } N \\ \pi_1 &= \lambda z.\, z\mathbf{K} \\ \pi_2 &= \lambda z.\, z\mathbf{K}_*.\end{aligned}$$

Then

$$\begin{aligned}\pi_1\langle M, N\rangle &\xrightarrow{+}_{\beta} M \\ \pi_2\langle M, N\rangle &\xrightarrow{+}_{\beta} N \\ \langle M, N\rangle\mathbf{T} &\xrightarrow{+}_{\beta} M \\ \langle M, N\rangle\mathbf{F} &\xrightarrow{+}_{\beta} N.\end{aligned}$$

The proof of the first equation is given in Example 3.14 and the proof of the other three is left as an exercise.

Example 3.14. We have

$$\begin{aligned} \pi_1\langle M, N\rangle = \big(\lambda z.\, z\mathbf{K}\big)\big(\lambda z.\, zMN\big) & \\ \longrightarrow_\beta\ \big(z\mathbf{K}\big)[z := \lambda z.\, zMN] &= \big(\lambda z.\, zMN\big)\mathbf{K} \\ \longrightarrow_\beta\ (zMN)[z := \mathbf{K}] &= \mathbf{K}MN \xrightarrow{+}_\beta M, \end{aligned}$$

by Proposition 3.2.

Observe that if we define the combinator

$$\mathbf{pair} = \lambda x.\, (\lambda y.\, (\lambda z.\, (zx)y)) = \lambda xyz.\, zxy,$$

then

$$\mathbf{pair}\, MN \xrightarrow{+}_\beta \lambda z.\, zMN = \langle M, N\rangle.$$

The combinator **pair** is very closely related to the combinator $\mathsf{if\,then\,else} = \lambda zxy.\, zxy$. Both combinators contain the term zxy, but in **pair**, the variables are abstracted in the order xyz, and in $\mathsf{if\,then\,else}$, they are abstracted in the order zxy. So

$$\begin{aligned} \mathbf{pair}\, PQM &\xrightarrow{+}_\beta MPQ \\ (\mathsf{if\ then\ else}\ )MPQ &\xrightarrow{+}_\beta MPQ. \end{aligned}$$

In the next section we show how to encode the natural numbers in the λ-calculus and how to compute various arithmetical functions.

3.4 Representing the Natural Numbers

Historically the natural numbers were first represented in the λ-calculus by Church in the 1930's. Later in 1976 Barendregt came up with another representation which is more convenient to show that the recursive functions are λ-definable. We start with Church's representation.

First, given any two λ-terms F and M, for any natural number $n \in \mathbb{N}$, we define $F^n(M)$ inductively as follows:

$$\begin{aligned} F^0(M) &= M \\ F^{n+1}(M) &= F(F^n(M)). \end{aligned}$$

Definition 3.10. (Church Numerals) The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by

$$\mathbf{c}_n = \lambda fx.\, f^n(x).$$

So $\mathbf{c}_0 = \lambda fx.\, x = \mathbf{K}_*$, $\mathbf{c}_1 = \lambda fx.\, fx$, $\mathbf{c}_2 = \lambda fx.\, f(fx)$, $\mathbf{c}_3 = \lambda fx.\, f(f(fx))$, *etc.* The Church numerals are β-normal forms.

Observe that

$$\mathbf{c}_0 Fz = (\mathbf{c}_0 F)z = ((\lambda fx.\, x)F)z \xrightarrow{+}_\beta z \tag{$\dagger_1$}$$

$$\mathbf{c}_n Fz = (\mathbf{c}_n F)z = ((\lambda fx.\, f^n(x))F)z \xrightarrow{+}_\beta F^n(z), \quad n \geq 1. \tag{$\dagger_2$}$$

This shows that $\mathbf{c}_n$ *iterates n times the function represented by the term F on initial input z.* This is the trick behind the definition of the Church numerals. This suggests the following definition.

Definition 3.11. The *iteration combinator* **Iter** is given by

$$\mathbf{Iter} = \lambda nfx.\, nfx.$$

Observe that

$$\mathbf{Iter}\, \mathbf{c}_n\, F\, X = (\mathbf{Iter}\, \mathbf{c}_n\, F)X \xrightarrow{+}_\beta (\mathbf{c}_n\, F)X \xrightarrow{+}_\beta F^n X, \tag{$\dagger_3$}$$

that is, the result of iterating F for n steps starting with the initial term X.

Remark: The combinator **Iter** is actually equal to the combinator

$$\textsf{if then else} = \lambda bxy.\, bxy$$

of Definition 3.3. Remarkably, if n (or b) is a boolean, then this combinator behaves like a conditional, but if n (or b) is a Church numeral, then it behaves like an iterator. A closely related combinator is **Fold**, defined by

$$\mathbf{Fold} = \lambda xfn.\, nxf.$$

The only difference is that the abstracted variables are listed in the order x, f, n, instead of n, f, x. In fact,

$$\mathbf{Fold} = \mathbf{pair},$$

as defined just after Example 3.14. This version of an iterator is used when the Church numerals are defined as $\lambda xf.\, f^n(x)$ instead of $\lambda fx.\, f^n(x)$, where x and f are permuted in the λ-binder.

Let us show how some basic functions on the natural numbers can be defined.

Example 3.15. We begin with the constant function $\mathbf{Z}$ given by $\mathbf{Z}(n) = 0$ for all $n \in \mathbb{N}$. We claim that $\mathbf{Z_c} = \lambda x.\, \mathbf{c_0}$ works. Indeed, we have

$$\mathbf{Z_c}\, \mathbf{c}_n = (\lambda x.\, \mathbf{c_0})\mathbf{c}_n \longrightarrow_\beta \mathbf{c_0}[x := \mathbf{c}_n] = \mathbf{c_0},$$

since $\mathbf{c_0}$ is a closed term.

Example 3.16. The successor function **Succ** is given by

$$\mathbf{Succ}(n) = n + 1.$$

We claim that

$$\mathbf{Succ_c} = \lambda nfx.\, f(nfx)$$

computes **Succ**. Indeed we have

$$\begin{aligned}
\mathbf{Succ_c}\, \mathbf{c}_n &= (\lambda nfx.\, f(nfx))\mathbf{c}_n \\
&\longrightarrow_\beta \big(\lambda fx.\, f(nfx)\big)[n := \mathbf{c}_n] = \lambda fx.\, f(\mathbf{c}_n fx) \\
&\longrightarrow_\beta \lambda fx.\, f(f^n(x)) \\
&= \lambda fx.\, f^{n+1}(x) = \mathbf{c}_{n+1}.
\end{aligned}$$

Example 3.17. The function **IsZero** which tests whether a natural number is equal to 0 is defined by the combinator

$$\mathbf{IsZero_c} = \lambda x.\, x(\mathbf{K}\,\mathbf{F})\mathbf{T}.$$

The proof that it works is left as an exercise.

Addition and multiplication are a little more tricky to define.

Proposition 3.5. *(J.B. Rosser) Define* **Add** *and* **Mult** *as the combinators given by*

$$\begin{aligned}
\mathbf{Add} &= \lambda mnfx.\, mf(nfx) \\
\mathbf{Mult} &= \lambda xyz.\, x(yz).
\end{aligned}$$

We have

$$\begin{aligned}
\mathbf{Add}\, \mathbf{c}_m \mathbf{c}_n &\xrightarrow{+}_\beta \mathbf{c}_{m+n} \\
\mathbf{Mult}\, \mathbf{c}_m \mathbf{c}_n &\xrightarrow{+}_\beta \mathbf{c}_{m*n}
\end{aligned}$$

for all $m, n \in \mathbb{N}$.

Proof. We have

$$\begin{aligned}
\mathbf{Add}\, \mathbf{c}_m \mathbf{c}_n &= (\lambda mnfx.\, mf(nfx))\mathbf{c}_m \mathbf{c}_n \\
&\xrightarrow{+}_\beta (\lambda fx.\, \mathbf{c}_m f(\mathbf{c}_n fx)) \\
&\xrightarrow{+}_\beta \lambda fx.\, f^m(f^n(x)) \\
&= \lambda fx.\, f^{m+n}(x) = \mathbf{c}_{m+n}.
\end{aligned}$$

For multiplication we need to prove by induction on m that

$$(\mathbf{c}_n x)^m(y) \xrightarrow{*}_\beta x^{m*n}(y). \qquad (*)$$

If $m = 0$, then both sides are equal to y.

For the induction step, we have

$$\begin{aligned}(\mathbf{c}_n x)^{m+1}(y) &= \mathbf{c}_n x((\mathbf{c}_n x)^m(y)) \\ &\xrightarrow{*}_\beta \mathbf{c}_n x(x^{m*n}(y)) && \text{by induction} \\ &\xrightarrow{*}_\beta x^n(x^{m*n}(y)) \\ &= x^{n+m*n}(y) = x^{(m+1)*n}(y).\end{aligned}$$

We now have

$$\begin{aligned}\mathbf{Mult}\, \mathbf{c}_m\mathbf{c}_n &= (\lambda xyz.\, x(yz))\mathbf{c}_m\mathbf{c}_n \\ &\xrightarrow{+}_\beta \lambda z.\, (\mathbf{c}_m(\mathbf{c}_n z)) \\ &= \lambda z.\, ((\lambda fy.\, f^m(y))(\mathbf{c}_n z)) \\ &\xrightarrow{+}_\beta \lambda zy.\, (\mathbf{c}_n z)^m(y),\end{aligned}$$

and since we proved in $(*)$ that

$$(\mathbf{c}_n z)^m(y) \xrightarrow{*}_\beta z^{m*n}(y),$$

we get

$$\mathbf{Mult}\, \mathbf{c}_m\mathbf{c}_n \xrightarrow{+}_\beta \lambda zy.\, (\mathbf{c}_n z)^m(y) \xrightarrow{+}_\beta \lambda zy.\, z^{m*n}(y) = \mathbf{c}_{m*n},$$

which completes the proof. □

As an exercise the reader should prove that addition and multiplication can also be defined in terms of **Iter** (see Definition 3.11) by

$$\begin{aligned}\mathbf{Add} &= \lambda mn.\, \mathbf{Iter}\, m\, \mathbf{Succ_c}\, n \\ \mathbf{Mult} &= \lambda mn.\, \mathbf{Iter}\, m\, (\mathbf{Add}\, n)\, \mathbf{c}_0.\end{aligned}$$

The above expressions are close matches to the primitive recursive definitions of addition and multiplication. To check that they work, prove that

$$\mathbf{Add}\, \mathbf{c}_m\, \mathbf{c}_n \xrightarrow{+}_\beta (\mathbf{Succ_c})^m(\mathbf{c}_n) \xrightarrow{+}_\beta \mathbf{c}_{m+n}$$

and

$$\mathbf{Mult}\, \mathbf{c}_m\, \mathbf{c}_n \xrightarrow{+}_\beta (\mathbf{Add}\, n)^m(\mathbf{c}_0) \xrightarrow{+}_\beta \mathbf{c}_{m*n}.$$

A version of the exponential function can also be defined.

Example 3.18. A function that plays an important technical role is the predecessor function **Pred** defined such that

$$\begin{aligned}\mathbf{Pred}(0) &= 0 \\ \mathbf{Pred}(n+1) &= n.\end{aligned}$$

It turns out that it is quite tricky to define this function in terms of the Church numerals. Church and his students struggled for a while until Kleene found a solution in his famous 1936 paper. The story goes that Kleene found his solution when he was sittting in the dentist's chair! The trick is to make use of pairs. Kleene's solution is

$$\mathbf{Pred_K} = \lambda n.\, \pi_2(\mathbf{Iter}\, n\, (\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)\, \langle \mathbf{c}_0, \mathbf{c}_0\rangle).$$

In the above expression, **Iter** is applied to the three arguments n, $(\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)$, and $\langle \mathbf{c}_0, \mathbf{c}_0\rangle$. Thus we have

$$\mathbf{Pred_K}\, \mathbf{c}_n \xrightarrow{+}_\beta \pi_2((\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^n \langle \mathbf{c}_0, \mathbf{c}_0\rangle).$$

The reason this works is that we see immediately that

$$(\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^0 \langle \mathbf{c}_0, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_0, \mathbf{c}_0\rangle,$$

and we can prove by induction that

$$(\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^{n+1} \langle \mathbf{c}_0, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{n+1}, \mathbf{c}_n\rangle.$$

Then we have

$$\mathbf{Pred_K}\, \mathbf{c}_0 \xrightarrow{+}_\beta \pi_2(\langle \mathbf{c}_0, \mathbf{c}_0\rangle) = \mathbf{c}_0$$

and

$$\mathbf{Pred_K}\, \mathbf{c}_{n+1} \xrightarrow{+}_\beta \pi_2(\langle \mathbf{c}_{n+1}, \mathbf{c}_n\rangle) = \mathbf{c}_n.$$

For the base case $n = 0$ we get

$$\begin{aligned}(\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)\langle \mathbf{c}_0, \mathbf{c}_0\rangle &\longrightarrow_\beta \langle \mathbf{Succ_c}(\pi_1 \langle \mathbf{c}_0, \mathbf{c}_0\rangle), \pi_1 \langle \mathbf{c}_0, \mathbf{c}_0\rangle\rangle \\ &\xrightarrow{+}_\beta \langle \mathbf{Succ_c}(\mathbf{c}_0), \mathbf{c}_0\rangle \\ &\xrightarrow{+}_\beta \langle \mathbf{c}_1, \mathbf{c}_0\rangle.\end{aligned}$$

For the induction step, we have

$$\begin{aligned}&(\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^{n+2} \langle \mathbf{c}_0, \mathbf{c}_0\rangle \\ &\quad = (\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)\big((\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^{n+1} \langle \mathbf{c}_0, \mathbf{c}_0\rangle\big) \\ &\quad \xrightarrow{+}_\beta (\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)\langle \mathbf{c}_{n+1}, \mathbf{c}_n\rangle \\ &\quad \xrightarrow{+}_\beta \langle \mathbf{Succ_c}(\mathbf{c}_{n+1}), \mathbf{c}_{n+1}\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{n+2}, \mathbf{c}_{n+1}\rangle.\end{aligned}$$

Here is another tricky solution due to J. Velmans (according to H. Barendregt):

$$\mathbf{Pred_c} = \lambda xyz.\, x(\lambda pq.\, q(py))(\mathbf{K}z)\mathbf{I}.$$

We leave it to the reader to verify that it works.

The ability to construct pairs together with the **Iter** combinator allows the definition of a large class of functions, because **Iter** is "type-free" in its second and third arguments so it really allows higher-order primitive recursion.

Example 3.19. The factorial function defined such that

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1)n! \end{aligned}$$

can be defined. First we define h by

$$h = \lambda xn.\, \mathbf{Mult}\, \mathbf{Succ_c} n\, x$$

and then

$$\mathbf{fact} = \lambda n.\, \pi_1(\mathbf{Iter}\, n\, (\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle).$$

The above expression, **Iter** is applied to the arguments n, $(\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)$, and $\langle \mathbf{c}_1, \mathbf{c}_0\rangle$. We have

$$h\, \mathbf{c}_{n!}\, \mathbf{c}_n \xrightarrow{+}_\beta \mathbf{Mult}\, \mathbf{Succ_c}\, \mathbf{c}_n\, \mathbf{c}_{n!} \xrightarrow{+}_\beta \mathbf{Mult}\, \mathbf{c}_{n+1}\, \mathbf{c}_{n!} \xrightarrow{+}_\beta \mathbf{c}_{(n+1)!},$$

and

$$\mathbf{fact}\, \mathbf{c}_n \xrightarrow{+}_\beta \pi_1((\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)^n\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle).$$

This works because

$$(\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)^0\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_1, \mathbf{c}_0\rangle = \langle \mathbf{c}_{0!}, \mathbf{c}_0\rangle,$$

and by induction,

$$(\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)^{n+1}\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{(n+1)!}, \mathbf{c}_{n+1}\rangle.$$

Then we have

$$\begin{aligned} \mathbf{fact}\, \mathbf{c}_n &\xrightarrow{+}_\beta \pi_1((\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)^n\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle) \\ &\xrightarrow{+}_\beta \pi_1(\langle \mathbf{c}_{n!}, \mathbf{c}_n\rangle) \xrightarrow{+}_\beta \mathbf{c}_{n!}. \end{aligned}$$

For the induction step, we have

$$\begin{aligned} &(\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)^{n+2}\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle \\ &\quad = (\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)\Big(\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)^{n+1}\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle\Big) \\ &\quad \xrightarrow{+}_\beta (\lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z), \mathbf{Succ_c}(\pi_2 z)\rangle)\langle \mathbf{c}_{(n+1)!}, \mathbf{c}_{n+1}\rangle \\ &\quad \xrightarrow{+}_\beta \langle h\, \mathbf{c}_{(n+1)!}\, \mathbf{c}_{n+1}, \mathbf{Succ_c}(\mathbf{c}_{n+1})\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{(n+2)!}, \mathbf{c}_{n+2}\rangle. \end{aligned}$$

We leave the details as an exercise.

Barendregt came up with another way of representing the natural numbers that makes things easier.

Definition 3.12. (Barendregt Numerals) The *Barendregt numerals* $\mathbf{b}_n$ are defined as follows:

$$\begin{aligned} \mathbf{b}_0 &= \mathbf{I} = \lambda x.\, x \\ \mathbf{b}_{n+1} &= \langle \mathbf{F}, \mathbf{b}_n \rangle. \end{aligned}$$

The Barendregt numerals are β-normal forms. Barendregt uses the notation $\ulcorner n \urcorner$ instead of $\mathbf{b}_n$ but this notation is also used for the Church numerals by other authors so we prefer using $\mathbf{b}_n$ (which is consistent with the use of $\mathbf{c}_n$ for the Church numerals). The Barendregt numerals are tuples, which makes operating on them simpler than the Church numerals which encode n as the composition f^n.

Proposition 3.6. *The functions* **Succ**, **Pred** *and* **IsZero** *are defined in terms of the Barendregt numerals by the combinators*

$$\begin{aligned} \mathbf{Succ_b} &= \lambda x.\, \langle \mathbf{F}, x \rangle \\ \mathbf{Pred_b} &= \lambda x.\, (x\mathbf{F}) \\ \mathbf{IsZero_b} &= \lambda x.\, (x\mathbf{T}), \end{aligned}$$

and we have

$$\begin{aligned} \mathbf{Succ_b}\, \mathbf{b}_n &\xrightarrow{+}_\beta \mathbf{b}_{n+1} \\ \mathbf{Pred_b}\, \mathbf{b}_0 &\xrightarrow{+}_\beta \mathbf{b}_0 \\ \mathbf{Pred_b}\, \mathbf{b}_{n+1} &\xrightarrow{+}_\beta \mathbf{b}_n \\ \mathbf{IsZero_b}\, \mathbf{b}_0 &\xrightarrow{+}_\beta \mathbf{T} \\ \mathbf{IsZero_b}\, \mathbf{b}_{n+1} &\xrightarrow{+}_\beta \mathbf{F}. \end{aligned}$$

The proof is left as an exercise.

Since there is an obvious bijection between the Church combinators and the Barendregt combinators there should be combinators effecting the translations. Indeed we have the following result.

Proposition 3.7. *The combinator* T *given by*

$$T = \lambda x.\, (x\mathbf{Succ_b})\mathbf{b}_0$$

has the property that

$$T\, \mathbf{c}_n \xrightarrow{+}_\beta \mathbf{b}_n \qquad \textit{for all } n \in \mathbb{N}.$$

Proof. We proceed by induction on n. For the base case

$$\begin{aligned} T\,\mathbf{c}_0 &= (\lambda x.\,(x\mathbf{Succ_b})\mathbf{b}_0)\mathbf{c}_0 \\ &\xrightarrow{+}_{\beta} (\mathbf{c}_0\mathbf{Succ_b})\mathbf{b}_0 && \text{by } (\dagger_1) \\ &\xrightarrow{+}_{\beta} \mathbf{b}_0. \end{aligned}$$

For the induction step,

$$\begin{aligned} T\,\mathbf{c}_n &= (\lambda x.\,(x\mathbf{Succ_b})\mathbf{b}_0)\mathbf{c}_n \\ &\xrightarrow{+}_{\beta} (\mathbf{c}_n\mathbf{Succ_b})\mathbf{b}_0 && \text{by } (\dagger_2) \\ &\xrightarrow{+}_{\beta} \mathbf{Succ_b}^n(\mathbf{b}_0). \end{aligned}$$

Thus we need to prove that

$$\mathbf{Succ_b}^n(\mathbf{b}_0) \xrightarrow{+}_{\beta} \mathbf{b}_n. \tag{$*$}$$

For the base case $n = 0$, the left-hand side reduces to $\mathbf{b}_0$.
For the induction step, we have

$$\begin{aligned} \mathbf{Succ_b}^{n+1}(\mathbf{b}_0) &= \mathbf{Succ_b}(\mathbf{Succ_b}^n(\mathbf{b}_0)) \\ &= \xrightarrow{+}_{\beta} \mathbf{Succ_b}(\mathbf{b}_n) && \text{by induction} \\ &= \xrightarrow{+}_{\beta} \mathbf{b}_{n+1}, \end{aligned}$$

which concludes the proof. □

There is also a combinator defining the inverse map but it is defined recursively and we don't know how to express recursive definitions in the λ-calculus. This is achieved by using fixed-point combinators.

Remark: With some work, it is possible to show that lists and trees can be represented in the λ-calculus.

3.5 Fixed-Point Combinators and Recursively Defined Functions

Fixed-point combinators are the key to the definability of recursive functions in the λ-calculus. We begin with the $\mathbf{Y}$-combinator due to Curry.

Proposition 3.8. *(Curry $\mathbf{Y}$-combinator) If we define the combinator $\mathbf{Y}$ as*

$$\mathbf{Y} = \lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx)),$$

then for any λ-term F we have

$$F(\mathbf{Y}F) \xleftrightarrow{*}_{\beta} \mathbf{Y}F.$$

Proof. Write $W = \lambda x.\, F(xx)$. We have

$$F(\mathbf{Y}F) = F\Big(\big(\lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))\big)F\Big) \longrightarrow_\beta F\big((\lambda x.\, F(xx))(\lambda x.\, F(xx))\big) = F(WW),$$

and

$$\begin{aligned}\mathbf{Y}F = \big(\lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))\big)F &\longrightarrow_\beta (\lambda x.\, F(xx))(\lambda x.\, F(xx)) = (\lambda x.\, F(xx))W \\ &\longrightarrow_\beta F(WW).\end{aligned}$$

Therefore $F(\mathbf{Y}F) \overset{*}{\longleftrightarrow}_\beta \mathbf{Y}F$, as claimed. □

Observe that neither $F(\mathbf{Y}F) \xrightarrow{+}_\beta \mathbf{Y}F$ nor $\mathbf{Y}F \xrightarrow{+}_\beta F(\mathbf{Y}F)$. This is a slight disadvantage of the Curry $\mathbf{Y}$-combinator. Turing came up with another fixed-point combinator that does not have this problem.

Proposition 3.9. *(Turing $\mathbf{\Theta}$-combinator) If we define the combinator $\mathbf{\Theta}$ as*

$$\mathbf{\Theta} = (\lambda xy.\, y(xxy))(\lambda xy.\, y(xxy)),$$

then for any λ-term F we have

$$\mathbf{\Theta}F \xrightarrow{+}_\beta F(\mathbf{\Theta}F).$$

Proof. If we write $A = (\lambda xy.\, y(xxy))$, then $\mathbf{\Theta} = AA$. We have

$$\begin{aligned}\mathbf{\Theta}F &= (AA)F = ((\lambda xy.\, y(xxy))A)F \\ &\longrightarrow_\beta (\lambda y.\, y(AAy))F \\ &\longrightarrow_\beta F(AAF) \\ &= F(\mathbf{\Theta}F),\end{aligned}$$

as claimed. □

Now we show how to use the fixed-point combinators to represent recursively-defined functions in the λ-calculus.

Example 3.20. There is a combinator G such that

$$GX \xrightarrow{+}_\beta X(XG) \quad \text{for all } X.$$

Informally, the idea is to consider the "functional" $F = \lambda gx.\, x(xg)$ and to find a fixed-point of this functional. Pick

$$G = \mathbf{\Theta}\lambda gx.\, x(xg) = \mathbf{\Theta}F.$$

Since by Proposition 3.9 we have $G = \mathbf{\Theta}F \xrightarrow{+}_\beta F(\mathbf{\Theta}F) = FG$, and we also have

$$FG = (\lambda gx.\, x(xg))G \longrightarrow_\beta \lambda x.\, x(xG),$$

so $G \xrightarrow{+}_\beta FG \xrightarrow{+}_\beta \lambda x.\, x(xG)$, which implies

$$GX \xrightarrow{+}_\beta (\lambda x.\, x(xG))X \longrightarrow_\beta X(XG).$$

In general, if we want to define a function G recursively such that

$$GX \xrightarrow{+}_{\beta} M(X, G)$$

where $M(X, G)$ is λ-term containing recursive occurrences of G, we let $F = \lambda gx.\, M(x, g)$ and

$$G = \mathbf{\Theta} F.$$

Then we have

$$G \xrightarrow{+}_{\beta} FG = (\lambda gx.\, M(x, g))G \longrightarrow_{\beta} \lambda x.\, M(x, g)[g := G] = \lambda x.\, M(x, G),$$

so

$$GX \xrightarrow{+}_{\beta} (\lambda x.\, M(x, G))X \longrightarrow_{\beta} M(x, G)[x := X] = M(X, G),$$

as desired.

Example 3.21. Here is how the factorial function can be defined (using the Church numerals). Let

$$F = \lambda gn.\, \mathsf{if}\ \mathbf{IsZero_c}\, n\ \mathsf{then}\ \mathbf{c}_1\ \mathsf{else}\ \mathbf{Mult}\, n\, g(\mathbf{Pred_c}\, n).$$

Then the term $G = \mathbf{\Theta} F$ defines the factorial function. The verification of the above fact is left as an exercise.

As usual with recursive definitions there is no guarantee that the function that we obtain terminates for all input.

Example 3.22. For example, if we consider

$$F = \lambda gn.\, \mathsf{if}\ \mathbf{IsZero_c}\, n\ \mathsf{then}\ \mathbf{c}_1\ \mathsf{else}\ \mathbf{Mult}\, n\, g(\mathbf{Succ_c}\, n),$$

then for $n \geq 1$, the reduction behavior is

$$G\mathbf{c}_n \xrightarrow{+}_{\beta} \mathbf{Mult}\, \mathbf{c}_n\, G\, \mathbf{c}_{n+1},$$

which does not terminate.

We leave it as an exercise to show that the inverse of the function T mapping the Church numerals to the Barendregt numerals is given by the combinator

$$T^{-1} = \mathbf{\Theta}(\lambda fx.\, \mathsf{if}\ \mathbf{IsZero_b}\, x\ \mathsf{then}\ \mathbf{c}_0\ \mathbf{else}\ \mathbf{Succ_c}(f(\mathbf{Pred_b}\, x)).$$

It is remarkable that the λ-calculus allows the implementation of *arbitrary recursion* without a stack, just using λ-terms as the data-structure and β-reduction. This does not mean that this evaluation mechanism is efficient but this is another story (as well as evaluation strategies, which have to do with parameter-passing strategies, call-by-name, call-by-value).

Now we have all the ingredients to show that all the (total) computable functions are definable in the λ-calculus.

3.6 λ-Definability of the Computable Functions

Let us begin by reviewing the definition of the computable functions (recursive functions) (à la Herbrand–Gödel–Kleene). For our purposes it suffices to consider functions (partial or total) $f\colon \mathbb{N}^n \to \mathbb{N}$ as opposed to the more general case of functions $f\colon (\Sigma^*)^n \to \Sigma^*$ defined on strings.

Definition 3.13. The *base functions* are the functions Z, S, P_i^n defined as follows:

(1) The constant *zero function* Z such that

$$Z(n) = 0, \qquad \text{for all } n \in \mathbb{N}.$$

(2) The *successor function* S such that

$$S(n) = n + 1, \qquad \text{for all } n \in \mathbb{N}.$$

(3) For every $n \geq 1$ and every i with $1 \leq i \leq n$, the *projection function* P_i^n such that

$$P_i^n(x_1, \ldots, x_n) = x_i, \qquad x_1, \ldots, x_n \in \mathbb{N}.$$

Next comes (extended) composition.

Definition 3.14. Given any partial or total function $g\colon \mathbb{N}^m \to \mathbb{N}$ ($m \geq 1$) and any m partial or total functions $h_i\colon \mathbb{N}^n \to \mathbb{N}$ ($n \geq 1$), the *composition of* g *and* $h_1, \ldots, h_m$, denoted $g \circ (h_1, \ldots, h_m)$, is the partial or total function function $f\colon \mathbb{N}^n \to \mathbb{N}$ given by

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)), \qquad x_1, \ldots, x_n \in \mathbb{N}.$$

If g or any of the h_i are partial functions, then $f(x_1, \ldots, x_n)$ is defined if and only if *all* $h_i(x_1, \ldots, x_n)$ *are defined and* $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ *is defined.*

Note that even if g "ignores" one of its arguments, say the ith one, $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ is undefined if $h_i(x_1, \ldots, x_n)$ is undefined.

Definition 3.15. Given any partial or total functions $g\colon \mathbb{N}^m \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+2} \to \mathbb{N}$ ($m \geq 1$), the partial or total function function $f\colon \mathbb{N}^{m+1} \to \mathbb{N}$ is defined by *primitive recursion* from g and h if f is given by:

$$\begin{aligned} f(0, x_1, \ldots, x_m) &= g(x_1, \ldots, x_m) \\ f(n + 1, x_1, \ldots, x_m) &= h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) \end{aligned}$$

for all $n, x_1, \ldots, x_m \in \mathbb{N}$. If $m = 0$, then g is some fixed natural number and we have

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

It can be shown that if g and h are total functions, then so if f.

Note that the second clause of the definition of primitive recursion is

$$f(n+1, x_1, \ldots, x_m) = h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) \tag{$*_1$}$$

but in an earlier definition it was

$$f(n+1, x_1, \ldots, x_m) = h(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m), \tag{$*_2$}$$

with the first two arguments of h permuted. Since

$$\begin{aligned} h \circ (P_2^{m+2}, P_1^{m+2}, P_3^{m+2}, \ldots, P_{m+2}^{m+2})&(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m) \\ &= h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) \end{aligned}$$

and

$$\begin{aligned} h \circ (P_2^{m+2}, P_1^{m+2}, P_3^{m+2}, \ldots, P_{m+2}^{m+2})&(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) \\ &= h(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m), \end{aligned}$$

the two definitions are equivalent. In this section we chose version $(*_1)$ because it matches the treatment in Barendregt [3] and will make it easier for the reader to follow Barendregt [3] if they wish.

The last operation is *minimization* (sometimes called minimalization).

Definition 3.16. Given any partial or total function $g \colon \mathbb{N}^{m+1} \to \mathbb{N}$ ($m \geq 0$), the partial or total function function $f \colon \mathbb{N}^m \to \mathbb{N}$ is defined as follows: for all $x_1, \ldots, x_m \in \mathbb{N}$,

$$f(x_1, \ldots, x_m) = \text{the least } n \in \mathbb{N} \text{ such that } g(n, x_1, \ldots, x_m) = 0,$$

and undefined if there is no n such that $g(n, x_1, \ldots, x_m) = 0$. We say that f is *defined by minimization from* g, and we write

$$f(x_1, \ldots, x_m) = \mu x[g(x, x_1, \ldots, x_m) = 0].$$

For short, we write $f = \mu g$.

Even if g is a total function, f may be undefined for some (or all) of its inputs.

Definition 3.17. (Herbrand–Gödel–Kleene) The set of *partial computable* (or *partial recursive*) functions is the smallest set of partial functions (defined on $\mathbb{N}^n$ for some $n \geq 1$) which contains the base functions and is closed under

(1) Composition.

(2) Primitive recursion.

(3) Minimization.

The set of *computable* (or *recursive*) functions is the subset of partial computable functions that are total functions (that is, defined for all input).

We proved earlier the Kleene normal form, which says that *every* partial computable function $f\colon \mathbb{N}^m \to \mathbb{N}$ is computable as

$$f = g \circ \mu h,$$

for some *primitive recursive functions* $g\colon \mathbb{N} \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+1} \to \mathbb{N}$. The significance of this result is that f is built up from *total functions* using composition and primitive recursion, and only a single minimization is needed at the end.

Before stating our main theorem, we need to define what it means for a (numerical) function to be definable in the λ-calculus. This requires some care to handle partial functions.

Since there are combinators for translating Church numerals to Barendregt numerals and vice-versa, it does not matter which numerals we pick. We pick the Church numerals because primitive recursion is definable without using a fixed-point combinator.

Definition 3.18. A function (partial or total) $f\colon \mathbb{N}^n \to \mathbb{N}$ is λ-*definable* if for all $m_1, \ldots,$ $m_n \in \mathbb{N}$, there is a combinator (a closed λ-term) F with the following properties:

(1) The value $f(m_1, \ldots, m_n)$ is defined if and only if $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to a β-normal form (necessarily unique by the Church–Rosser theorem).

(2) If $f(m_1, \ldots, m_n)$ is defined, then

$$F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n} \overset{*}{\longleftrightarrow}_{\beta} \mathbf{c}_{f(m_1,\ldots,m_n)}.$$

In view of the Church–Rosser theorem (Theorem 3.1) and the fact that $\mathbf{c}_{f(m_1,\ldots,m_n)}$ is a β-normal form, we can replace

$$F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n} \overset{*}{\longleftrightarrow}_{\beta} \mathbf{c}_{f(m_1,\ldots,m_n)}$$

by

$$F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n} \overset{*}{\longrightarrow}_{\beta} \mathbf{c}_{f(m_1,\ldots,m_n)}.$$

Note that the termination behavior of f on inputs $m_1, \ldots, m_n$ has to *match* the reduction behavior of $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$, namely $f(m_1, \ldots, m_n)$ is undefined if *no* reduction sequence from $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reaches a β-normal form. Condition (2) ensures that if $f(m_1, \ldots, m_n)$ is defined, then the correct value $\mathbf{c}_{f(m_1,\ldots,m_n)}$ is computed by some reduction sequence from $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$. If we only care about total functions then we require that $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to a β-normal for *all* $m_1, \ldots, m_n$ and (2). A stronger and more elegant version of λ-definabilty that better captures when a function is undefined for some input is considered in Section 3.8.

We have the following remarkable theorems.

Theorem 3.10. *If a total function $f\colon \mathbb{N}^n \to \mathbb{N}$ is λ-definable, then it is (total) computable. If a partial function $f\colon \mathbb{N}^n \to \mathbb{N}$ is λ-definable, then it is partial computable.*

Although Theorem 3.10 is intuitively obvious since computation by β-reduction sequences are "clearly" computable, a detailed proof is long and very tedious. One has to define primitive recursive functions to mimick β-conversion, *etc.* Most books sweep this issue under the rug. Barendregt observes that the "λ-calculus is recursively axiomatized," which implies that the graph of the function beeing defined is recursively enumerable, but no details are provided; see Barendregt [3] (Chapter 6, Theorem 6.3.13). Kleene (1936) provides a detailed and very tedious proof. This is an amazing paper, but very hard to read. If the reader is not content she/he should work out the details over many long lonely evenings.

Theorem 3.11. *(Kleene, 1936) If a (total) function $f\colon \mathbb{N}^n \to \mathbb{N}$ is computable, then it is λ-definable. If a (partial) function $f\colon \mathbb{N}^n \to \mathbb{N}$ is is partial computable, then it is λ-definable.*

Proof. First we assume all functions to be total. There are several steps.

Step 1. The base functions are λ-definable.

We already showed that $\mathbf{Z_c}$ computes Z and that $\mathbf{Succ_c}$ computes S. Observe that $\mathbf{U}_i^n$ given by

$$\mathbf{U}_i^n = \lambda x_1 \cdots x_n.\, x_i$$

computes P_i^n.

Step 2. Closure under composition.

If g is λ-defined by the combinator G and $h_1, \ldots, h_m$ are λ-defined by the combinators $H_1, \ldots, H_m$, then $g \circ (h_1, \ldots, h_m)$ is λ-defined by

$$F = \lambda x_1 \cdots x_n.\, G(H_1 x_1 \cdots x_n) \ldots (H_m x_1 \cdots x_n).$$

Since the functions are total, there is no problem.

Step 3. Closure under primitive recursion.

We could use a fixed-point combinator but the combinator **Iter** and pairing do the job. If f is defined by primitive recursion from g and h, and if G λ-defines g and H λ-defines h, then f is λ-defined by

$$F = \lambda n x_1 \cdots x_m.\, \pi_1\big(\mathbf{Iter}\, n\, \lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, x_1 \cdots x_m,\, \mathbf{Succ_c}(\pi_2 z)\rangle\, \langle G x_1 \cdots x_m, \mathbf{c}_0\rangle\big).$$

The reason F works is that we can prove by induction that

$$\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big)^n \langle G\mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{f(n,n_1,\ldots,n_m)}, \mathbf{c}_n\rangle.$$

For the base case $n = 0$,

$$\begin{aligned}&\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big)^0 \langle G\mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle \\ &\qquad \xrightarrow{+}_\beta \langle G\mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle = \langle \mathbf{c}_{g(n_1,\ldots,n_m)}, \mathbf{c}_0\rangle = \langle \mathbf{c}_{f(0,n_1,\ldots,n_m)}, \mathbf{c}_0\rangle.\end{aligned}$$

For the induction step,

$$\begin{aligned}
&\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big)^{n+1} \langle G\mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle \\
&\qquad = \big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big) \\
&\qquad \Big(\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big)^{n} \langle G\mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle\Big) \\
&\qquad \xrightarrow{+}_{\beta} \big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}(\pi_2 z)\rangle\big) \langle \mathbf{c}_{f(n,n_1,\ldots,n_m)}, \mathbf{c}_n\rangle \\
&\qquad \xrightarrow{+}_{\beta} \langle H\mathbf{c}_{f(n,n_1,\ldots,n_m)}\, \mathbf{c}_n\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\, \mathbf{Succ_c}\, \mathbf{c}_n\rangle \\
&\qquad \xrightarrow{+}_{\beta} \langle \mathbf{c}_{h(f(n,n_1,\ldots,n_m),n,n_1,\ldots,n_m)}, \mathbf{c}_{n+1}\rangle = \langle \mathbf{c}_{f(n+1,n_1,\ldots,n_m)}, \mathbf{c}_{n+1}\rangle.
\end{aligned}$$

Since the functions are total, there is no problem.

We can also show that primitive recursion can be achieved using a fixed-point combinator. Define the combinators J and F by

$$J = \lambda f x x_1 \cdots x_m.\, \mathsf{if}\, \mathbf{IsZero_c} x\, \mathsf{then}\, G x_1 \cdots x_m\, \mathsf{else}\, H(f(\mathbf{Pred_c} x)\, x_1 \cdots x_m)(\mathbf{Pred_c}\, x)\, x_1 \cdots x_m,$$

and

$$F = \mathbf{\Theta} J.$$

Then F λ-defines f, and since the functions are total, there is no problem. This method must be used if we use the Barendregt numerals.

Step 4. Closure under minimization.

Suppose f is total and defined by minimization from g and that g is λ-defined by G.

Define the combinators J and F by

$$J = \lambda f x x_1 \cdots x_m.\, \mathsf{if}\ \mathbf{IsZero_c}\, G x\, x_1 \cdots x_m\ \mathsf{then}\ x\ \mathsf{else}\ f(\mathbf{Succ_c}\, x)\, x_1 \cdots x_m$$

and

$$F = \mathbf{\Theta} J.$$

It is not hard to check that

$$F\, \mathbf{c}_n\, \mathbf{c}_{n_1} \ldots \mathbf{c}_{n_n} \xrightarrow{+}_{\beta} \begin{cases} \mathbf{c}_n & \text{if } g(n, n_1, \ldots, n_m) = 0 \\ F\, \mathbf{c}_{n+1}\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_n} & \text{otherwise,} \end{cases}$$

and we can use this to prove that F λ-defines f. Since we assumed that f is total, some least n will be found. We leave the details as an exercise.

This finishes the proof that every total computable function is λ-definable.

To prove the result for the partial computable functions we appeal to the Kleene normal form: every partial computable function $f\colon \mathbb{N}^m \to \mathbb{N}$ is computable as

$$f = g \circ \mu h,$$

for some *primitive recursive functions* $g\colon \mathbb{N} \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+1} \to \mathbb{N}$. Then our previous proof yields combinators G and H that λ-define g and h. The minimization of h may fail but since

g is a total function of a single argument, $f(n_1, \ldots, n_m)$ is defined iff $g(\mu n[h(n, n_1, \ldots, n_m) = 0])$ is defined so it should be clear that F computes f, but the reader may want to provide a rigorous argument. A detailed proof is given in Hindley and Seldin [22] (Chapter 4, Theorem 4.18). □

Combining Theorem 3.10 and Theorem 3.11 we have established the remarkable result that the set of λ-definable total functions is exactly the set of (total) computable functions, and similarly for partial functions. So the λ-calculus has universal computing power.

Remark: With some work, it is possible to show that lists and trees can be represented in the λ-calculus. Since a Turing machine tape can be viewed as a list, it should be possible (but very tedious) to simulate a Turing machine in the λ-calculus. This simulation should be somewhat analogous to the proof that a Turing machine computes a computable function (defined à la Herbrand–Gödel–Kleene).

Since the λ-calculus has the same power as Turing machines we should expect some undecidabity results analogous to the undecidability of the halting problem or Rice's theorem. We state the following analog of Rice's theorem without proof. It is a corollary of a theorem known as the Scott–Curry theorem.

Theorem 3.12. *(D. Scott) Let $\mathcal{A}$ be any nonempty set of λ-terms not equal to the set of all λ-terms. If $\mathcal{A}$ is closed under β-reduction, then it is not computable (not recursive).*

Theorem 3.12 is proven in Barendregt [3] (Chapter 6, Theorem 6.6.2) and Barendregt [4].

As a corollary of Theorem 3.12 it is undecidable whether a λ-term has a β-normal form, a result originally proved by Church. This is an analog of the undecidability of the halting problem, but it seems more spectacular because the syntax of λ-terms is really very simple. The problem is that β-reduction is very powerful and elusive.

In the pure λ-calculus, some λ-terms have no β-normal form, and worse, it is undecidable whether a λ-term has a β-normal form.

Church realized that by adding *types*, we can restrict when a term M can be applied to a term N, and in particular, "wild"' applications can be avoided.

The idea is to introduce *type-checking rules* and only work with terms that type-check.

As a result, we obtain a system called the *simply-typed lambda calculus*, and by Theorem 3.15, *every* λ-term that type-checks in the simply-typed λ-calculus has a β-normal form.

The idea of introducing types as a mechanism to increase "safe practice" in writing programs has been picked up by programming language designers.

3.7 The Simply-Typed λ-Calculus

The first step is to define simple types. We assume that we have a countable set $\{\mathbf{T}_0, \mathbf{T}_1, \ldots, \mathbf{T}_n, \ldots\}$ of *base types* (or *atomic types*).

For example, the base types may include types such as Nat for the natural numbers, Bool for the booleans, String for strings, Tree for trees, *etc.* In the Curry–Howard isomorphism they correspond to the propositional symbols $\{\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n, \ldots\}$.

Definition 3.19. The *simple types* σ are defined inductively as follows.

(1) If $\mathbf{T}_i$ is a base type, then $\mathbf{T}_i$ is a simple type.

(2) If σ and τ are simple types, then $(\sigma \to \tau)$ is a simple type.

Thus $(\mathbf{T}_1 \to \mathbf{T}_1)$, $(\mathbf{T}_1 \to (\mathbf{T}_2 \to \mathbf{T}_1))$, $((\mathbf{T}_1 \to \mathbf{T}_2) \to \mathbf{T}_1)$, are simple types.

The standard abbreviation for $(\sigma_1 \to (\sigma_2 \to (\cdots \to \sigma_n)))$ is $\sigma_1 \to \sigma_2 \to \cdots \to \sigma_n$.

There is obviously a bijection between propositions and simple types. Every propositional symbol $\mathbf{P}_i$ can be viewed as a base type, and the proposition $(P \Rightarrow Q)$ corresponds to the simple type $(P \to Q)$. The only difference is that the custom is to use $\Rightarrow$ to denote logical implication and $\to$ for simple types. The reason is that intuitively a simple type $(\sigma \to \tau)$ corresponds to a *set of functions* from a domain of type σ to a range of type τ.

The next crucial step is to define simply-typed λ-terms. This is done in two stages. First we define *raw simply-typed λ-terms*. They have a simple inductive definition but they do not necessarily type-check so we define some type-checking rules that turn out to be *the Gentzen-style deduction proof rules annotated with simply-typed λ-terms*. These simply-typed λ-terms are representations of natural deductions.

We have a countable set of variables $\{x_0, x_1, \ldots, x_n \ldots\}$ that correspond to the atomic raw λ-terms. These are also the variables that are used for tagging assumptions when constructing deductions.

Definition 3.20. The *raw simply-typed λ-terms* (for short *raw terms* or *λ-terms*) M are defined inductively as follows.

(1) If x_i is a variable, then x_i is a raw term.

(2) If M and N are raw terms, then (MN) is a raw term called an *application*.

(3) If M is a raw term, σ is a simple type, and x is a variable, then the expression $\lambda x\colon \sigma.\, M$ is a raw term called a *λ-abstraction*.

Matching parentheses may be dropped or added for convenience.

Definition 3.21. In a raw λ-term M, a variable x appearing in an expression $\lambda x\colon \sigma$ is said to be *bound* in M. The other variables in M (if any) are said to be *free* in M. A λ-term M is *closed* if it has no free variables.

Example 3.23. For example, in the term $\lambda x\colon \sigma.\, (yx)$, the variable x is bound and the variable y is free. This term is not closed.

The term $\lambda y\colon \sigma \to \sigma.\, (\lambda x\colon \sigma.\, (yx))$ is closed.

The intuition is that a term of the form $\lambda x\colon \sigma.\, M$ represents a function. How such a function operates will be defined in terms of β-reduction.

Definition 3.22. The *depth* $d(M)$ of a raw λ-term M is defined inductively as follows.

1. If M is a variable x, then $d(x) = 0$.
2. If M is an application $(M_1 M_2)$, then $d(M) = \max\{d(M_1), d(M_2)\} + 1$.
3. If M is a λ-abstraction $(\lambda x\colon \sigma.\, M_1)$, then $d(M) = d(M_1) + 1$.

It is pretty clear that raw λ-terms have representations as (ordered) labeled trees.

Definition 3.23. Given a raw λ-term M, the *tree* $\mathrm{tree}(M)$ representing M is defined inductively as follows.

1. If M is a variable x, then $\mathrm{tree}(M)$ is the one-node tree labeled x.
2. If M is an application $(M_1 M_2)$, then $\mathrm{tree}(M)$ is the tree with a binary root node labeled . and with a left subtree $\mathrm{tree}(M_1)$ and a right subtree $\mathrm{tree}(M_2)$.
3. If M is a λ-abstraction $\lambda x\colon \sigma.\, M_1$, then $\mathrm{tree}(M)$ is the tree with a unary root node labeled $\lambda x\colon \sigma$ and with one subtree $\mathrm{tree}(M_1)$.

Definition 3.23 is illustrated in Figure 3.7.

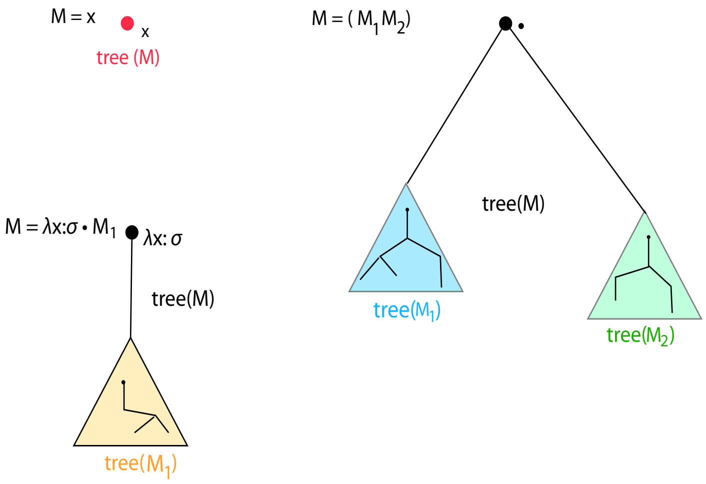


Figure 3.7: The tree $\mathrm{tree}(M)$ associated with a raw λ-term M.

Obviously, the depth $d(M)$ of raw λ-term is the depth of its tree representation $\mathrm{tree}(M)$.

Definition 3.23 could be used to deal with bound variables. For every leaf labeled with a bound variable x, we draw a backpointer to an ancestor of x determined as follows. Given

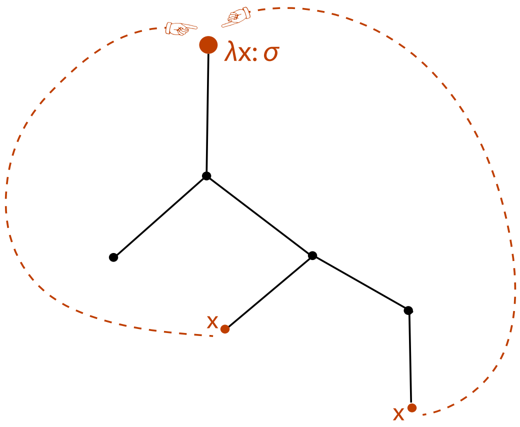


Figure 3.8: Using backpointers to deal with bound variables.

a leaf labeled with a bound variable x, climb up to the closest ancestor labeled $\lambda x\colon \sigma$, and draw a backpointer to this node. Then all bound variables can be erased. See Figure 3.8 for an example.

Definition 3.20 allows the construction of undesirable terms such as (xx) or $(\lambda x\colon \sigma.\,(xx))(\lambda x\colon \sigma.\,(xx))$ because no type-checking is done. Part of the problem is that the variables occurring in a raw term have not been assigned types. This can be done using a context (or type assignment).

Definition 3.24. A *context* (or *type assignment*) is a set of pairs $\Gamma = \{x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n\}$, where the σ_i are simple types and the variables x_i are pairwise distinct.

Once a type assignment has been provided, the type-checking rules are basically the proof rules of natural deduction in Gentzen-style.

Definition 3.25. The fact that a raw term M has type σ given a type assignment Γ that assigns types to all the free variables in M is written as

$$\Gamma \triangleright M\colon \sigma.$$

Such an expression is called a *judgement*. The symbol $\triangleright$ is used instead of the symbol $\rightarrow$ because $\rightarrow$ occurs in simple types.

Here are the typing-checking rules.

Definition 3.26. The *type-checking rules* of the simply-typed λ-calculus $\lambda^{\to}$ are listed below:

$$\Gamma, x\colon \sigma \triangleright x\colon \sigma$$

$$\frac{\Gamma, x\colon \sigma \triangleright M\colon \tau}{\Gamma \triangleright (\lambda x\colon \sigma.\, M)\colon \sigma \rightarrow \tau} \quad (\textit{abstraction})$$

$$\frac{\Gamma \triangleright M\colon \sigma \to \tau \quad \Delta \triangleright N\colon \sigma}{\Gamma \cup \Delta \triangleright (MN)\colon \tau} \quad (application)$$

We write $\vdash \Gamma \triangleright M\colon \sigma$ to express that the judgement $\Gamma \triangleright M\colon \sigma$ is provable. Given a raw simply-typed λ-term M, if there is a type-assigment Γ and a simple type σ such that the judgement $\Gamma \triangleright M\colon \sigma$ is provable, we say that M *type-checks with type* σ.

It can be shown by induction on the depth of raw terms that for a fixed type-assigment Γ, if a raw simply-typed λ-term M type-checks with some simple type σ, then σ is unique.

The correspondence between proofs in natural deduction and simply-typed λ-terms (the Curry/Howard isomorphism) is now clear: the abstraction rule corresponds to implication-introduction, the application rule corresponds to implication elimination, and the blue term is a *representation of the deduction* of the sequents $\Gamma, x\colon \sigma \to \sigma$, $\Gamma \to \sigma \Rightarrow \tau$, and $\Gamma \cup \Delta \to \tau$, with the types $\sigma, \sigma \Rightarrow \tau$ and τ viewed as propositions. Note that proofs correspond to closed λ-terms.

Example 3.24. For example, we have the type-checking proof

$$\cfrac{\cfrac{y\colon ((R \Rightarrow R) \Rightarrow Q) \triangleright y\colon ((R \Rightarrow R) \Rightarrow Q) \qquad \cfrac{z\colon R \triangleright z\colon R}{\triangleright \lambda z\colon R.\, z\colon R \Rightarrow R}}{y\colon ((R \Rightarrow R) \Rightarrow Q) \triangleright y(\lambda z\colon R.\, z)\colon Q}}{\triangleright \lambda y\colon ((R \Rightarrow R) \Rightarrow Q).\, y(\lambda z\colon R.\, z)\colon ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}$$

which shows that the simply-typed λ-term

$$M = \lambda y\colon ((R \Rightarrow R) \Rightarrow Q).\, y(\lambda z\colon R.\, z)$$

represents the proof

$$\cfrac{\cfrac{y\colon ((R \Rightarrow R) \Rightarrow Q) \to ((R \Rightarrow R) \Rightarrow Q) \qquad \cfrac{z\colon R \to R}{\to R \Rightarrow R}}{y\colon ((R \Rightarrow R) \Rightarrow Q) \to Q}}{\to ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q}$$

The proposition $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$ being proven is the type of the λ-term M.

The tree representing the λ-term $M = \lambda y\colon ((R \Rightarrow R) \Rightarrow Q).\, y(\lambda z\colon R.\, z)$ is shown in Figure 3.9.

Furthermore, and this is the deepest aspect of the Curry/Howard isomorphism, proof normalization corresponds to β-reduction in the simply-typed λ-calculus.

The notion of β-reduction is defined in terms of substitutions.

Definition 3.27. A *substitution* φ is a finite set of pairs $\varphi = \{(x_1, N_1), \ldots, (x_n, N_n)\}$, where the x_i are distinct variables and the N_i are raw λ-terms. We write

$$\varphi = [N_1/x_1, \ldots, N_n/x_n] \quad \text{or} \quad \varphi = [x_1 := N_1, \ldots, x_n := N_n].$$

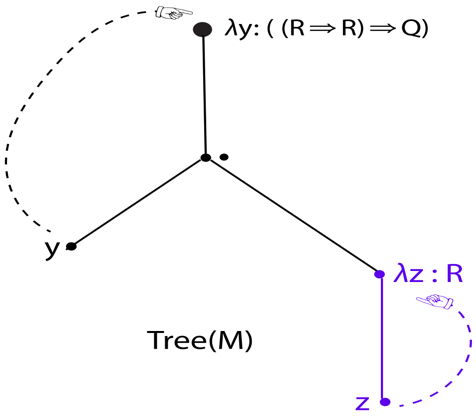


Figure 3.9: The tree representation of the λ-term M.

The second notation indicates more clearly that each term N_i is substituted for the variable x_i and it seems to have been almost universally adopted.

Given a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, for any variable x_i, we denote by φ_{-x_i} the new substitution where the pair (x_i, N_i) is replaced by the pair (x_i, x_i) (that is, the new substitution leaves x_i unchanged).

Definition 3.28. Given any raw λ-term M and any substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, we define the raw λ-term $M[\varphi]$, the result of applying the substitution φ to M, as follows:

(1) If $M = y$, with $y \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = y = M$.

(2) If $M = x_i$ for some $i \in \{1, \ldots, n\}$, then $M[\varphi] = N_i$.

(3) If $M = (PQ)$, then $M[\varphi] = (P[\varphi]Q[\varphi])$.

(4) If $M = \lambda x\colon \sigma.\, N$ and $x \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = \lambda x\colon \sigma.\, N[\varphi]$.

(5) If $M = \lambda x\colon \sigma.\, N$ and $x = x_i$ for some $i \in \{1, \ldots, n\}$, then
$M[\varphi] = \lambda x\colon \sigma.\, N[\varphi]_{-x_i}$.

There is a problem with the present definition of a substitution in Cases (4) and (5), which is that the result of substituting a term N_i containing the free variable x causes this variable to become bound after the substitution. We say that x is *captured*.

To remedy this problem, Church defined α-*conversion*.

Definition 3.29. The idea of α-*conversion* is that in a raw term M, any subterm of the form $\lambda x\colon \sigma.\, P$ can be replaced by the subterm $\lambda z\colon \sigma.\, P[x := z]$ where z is a new variable not occurring at all (free or bound) in M to obtain a new term M'. We write $M \equiv_\alpha M'$ and we view M and M' as equivalent.

Example 3.25. For example, $\lambda x \colon \sigma.\, yx \equiv_\alpha \lambda z \colon \sigma.\, yz$ and

$$\lambda y \colon \sigma \to \sigma.\, (\lambda x \colon \sigma.\, yx) \equiv_\alpha \lambda w \colon \sigma \to \sigma.\, (\lambda z \colon \sigma.\, wz).$$

The variables x and y are just place-holders.

Then given a raw λ-term M and a substitution $\varphi = [x_1 := N_1 \ldots, x_n := N_n]$, before applying φ to M we first apply some α-conversion to rename all bound variables in M obtaining $M' \equiv_\alpha M$ so that they do not occur in any of the N_i, and then safely apply the substitution φ to M' without any capture of variables. We say that the term M' is *safe* for the substitution φ.

The details are a bit tedious and we omit them. We refer the interested reader to Gallier [16] for a comprehensive discussion.

The following result shows that substitutions behave well with respect to type-checking. Given a context $\Gamma = \{x_1 \colon \sigma_1, \ldots, x_n \colon \sigma_n\}$, we let $\Gamma(x_i) = \sigma_i$.

Proposition 3.13. *For any raw λ-term M and any substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, whose domain contains the set of free variables of M, if the judgement $\Gamma \triangleright M \colon \tau$ is provable for some context Γ and some simple type τ, and if there is some context Δ such that for every free variable x_j in M the judgement $\Delta \triangleright N_j \colon \Gamma(x)$ is provable, then there some $M' \equiv_\alpha M$ such that the judgment $\Delta \triangleright M'[\varphi] \colon \tau$ is provable.*

Finally we define β-reduction and β-conversion as follows.

Definition 3.30. The relation $\longrightarrow_\beta$, called *immediate β-reduction*, is the smallest relation satisfying the following properties for all raw λ-terms M, N, P, Q:

$$(\lambda x \colon \sigma.\, M)N \longrightarrow_\beta M[x := N]$$

provided that M is safe for $[x := N]$;

$$\frac{M \longrightarrow_\beta N}{MQ \longrightarrow_\beta NQ} \qquad \frac{M \longrightarrow_\beta N}{PM \longrightarrow_\beta PN}, \qquad \text{for all } P, Q \qquad (\textit{congruence})$$

$$\frac{M \longrightarrow_\beta N}{\lambda x \colon \sigma.\, M \longrightarrow_\beta \lambda x \colon \sigma.\, N}, \qquad \text{for all } \sigma. \qquad (\xi)$$

The transitive closure of $\longrightarrow_\beta$ is denoted by $\xrightarrow{+}_\beta$, the reflexive and transitive closure of $\longrightarrow_\beta$ is denoted by $\xrightarrow{*}_\beta$, and we define *β-conversion*, denoted by $\xleftrightarrow{*}_\beta$, as the smallest equivalence relation $\xleftrightarrow{*}_\beta = (\longrightarrow_\beta \cup \longrightarrow_\beta^{-1})^*$ containing $\longrightarrow_\beta$.

Example 3.26. For example, we have

$$\begin{aligned}
&(\lambda u \colon \sigma.\, (vu))\big((\lambda x \colon \sigma \to \sigma.\, (xy))(\lambda z \colon \sigma.\, z)\big) \longrightarrow_\beta \\
&\qquad (\lambda u \colon \sigma.\, (vu))(\lambda x \colon \sigma \to \sigma.\, (xy))[x := (\lambda z \colon \sigma.\, z)] = (\lambda u \colon \sigma.\, (vu))\big((\lambda z \colon \sigma.\, z)y\big) \\
&\qquad\qquad \longrightarrow_\beta (\lambda u \colon \sigma.\, (vu))z[z := y] = (\lambda u \colon \sigma.\, (vu))y \longrightarrow_\beta (vu)[u := y] = vy.
\end{aligned}$$

In the above, β-reduction steps are applied to the blue subterms.

The following result shows that β-reduction (and β-conversion) behave well with respect to type-checking.

Proposition 3.14. *For any two raw λ-terms M and N, if there is a proof of the judgement $\Gamma \rhd M\colon \sigma$ for some context Γ and some simple type σ, and if $M \xrightarrow{+}_\beta N$ (or $M \xleftrightarrow{*}_\beta N$), then the judgement $\Gamma \rhd N\colon \sigma$ is provable. Thus β-reduction and β-conversion preserve type-checking.*

Definition 3.31. We say that a λ-term M is *β-irreducible* or a *β-normal form* if there is *no* term N such that $M \longrightarrow_\beta N$.

The fundamental result about the simply-typed λ-calculus is this.

Theorem 3.15. *For every raw λ-term M, if M type-checks, which means that there a provable judgement $\Gamma \rhd M\colon \sigma$ for some context Γ and some simple type σ, then the following results hold.*

(1) If $M \xrightarrow{*}_\beta M_1$ and $M \xrightarrow{*}_\beta M_2$, then there is some M_3 such that $M_1 \xrightarrow{*}_\beta M_3$ and $M_2 \xrightarrow{*}_\beta M_3$. We say that $\xrightarrow{*}_\beta$ is confluent.

(2) Every reduction sequence $M \xrightarrow{+}_\beta N$ is finite. We that that the simply-typed λ-calculus is strongly normalizing (for short, SN).

As a consequence of (1) and (2), there is a unique β-irreducible term N (called a β-normal form) such that $M \xrightarrow{*}_\beta N$.

A proof of Theorem 3.15 can be found in Gallier [14]. See also Gallier [16] which contains a thorough discussion of the techniques involved in proving these results.

In Theorem 3.15, the fact that the term M type-checks is crucial. Indeed the term

$$(\lambda x.\,(xx))(\lambda x.\,(xx)),$$

which does not type-check (we omitted the type tags σ of the variable x since they do not play any role), gives rise to an infinite β-reduction sequence!

In summary, the correspondence between proofs in intuitionistic logic and typed λ-terms on one hand and between proof normalization and β-reduction, can be used to translate results about typed λ-terms into results about proofs in intuitionistic logic. These results can be generalized to typed λ-calculi with product types and union types; see Gallier [14].

Using some suitable intuitionistic sequent calculi and Gentzen's cut elimination theorem or some suitable typed λ-calculi and (strong) normalization results about them, it is possible to prove that there is a decision procedure for propositional intuitionistic logic. However, it can also be shown that the time-complexity of any such procedure is very high. As a matter of fact, it was shown by Statman (1979) that deciding whether a proposition is intuitionisticaly provable is P-space complete; see [40] and Section 12.4. Here, we are alluding to *complexity theory*, another active area of computer science, Hopcroft, Motwani, and Ullman [23] and Lewis and Papadimitriou [28].

Readers who wish to learn more about these topics can read the two survey papers Gallier [14] (On the Correspondence Between Proofs and λ-Terms) and Gallier [13] (A Tutorial on Proof Systems and Typed λ-Calculi), both available on the website http://www.cis.upenn.edu/˜jean/gbooks/logic.html and the excellent introduction to proof theory by Troelstra and Schwichtenberg [43].

Anybody who really wants to understand logic should of course take a look at Kleene [24] (the famous "I.M."), but this is not recommended to beginners.

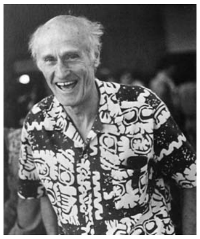


Figure 3.10: Stephen C. Kleene, 1909–1994

3.8 Definability of Functions in Typed Lambda-Calculi

In the pure λ-calculus, some λ-terms have no β-normal form, and worse, it is undecidable whether a λ-term has a β-normal form. In contrast, by Theorem 3.15, *every* raw λ-term that type-checks in the simply-typed λ-calculus has a β-normal form. Thus it is natural to ask whether the natural numbers are definable in the simply-typed λ-calculus because if the answer is positive, then the numerical functions definable in the simply-typed λ-calculus are guaranteed to be total.

This indeed possible. If we pick any base type σ, then we can define typed Church numerals $\mathbf{c}_n$ as terms of type $\mathsf{Nat}_\sigma = (\sigma \to \sigma) \to (\sigma \to \sigma)$, by

$$\mathbf{c}_n = \lambda f\colon (\sigma \to \sigma).\, \lambda x\colon \sigma.\, f^n(x).$$

The notion of λ-definable function is defined just as before. Then we can define **Add** and **Mult** as terms of type $\mathsf{Nat}_\sigma \to (\mathsf{Nat}_\sigma \to \mathsf{Nat}_\sigma)$ essentially as before, but surprise, not much more is definable. Among other things, strong typing of terms restricts the iterator combinator too much. It was shown by Schwichtenberg and Statman that the numerical functions definable in the simply-typed λ-calculus are the extended polynomials; see Statman [41] and Troelstra and Schwichtenberg [43]. The extended polynomials are the smallest class of numerical functions closed under composition containing

1. The constant functions 0 and 1.

2. The projections.

3. Addition and multiplication.

4. The function $\mathbf{IsZero_c}$.

Is there a way to get a larger class of total functions?

There are indeed various ways of doing this. One method is to add the natural numbers and the booleans as data types to the simply-typed λ-calculus, and to also add product types, an iterator combinator, and some new reduction rules. This way we obtain a system equivalent to Gödel's system T. A large class of numerical total functions containing the primitive recursive functions is definable in this system; see Girard–Lafond–Taylor [19]. Although theoretically interesting, this is not a practical system.

Another wilder method is to allow more general types to the simply-typed λ-calculus, the so-called *second-order types* or *polymorphic types*. In addition to base types, we allow *type variables* (often denoted $X, Y, \ldots$) ranging over simple types and new types of the form $\forall X.\, \sigma$.[3]

Example 3.27. The type $\forall X.\,(X \to X)$ is such a new type, and so is

$$\forall X.\,(X \to ((X \to X) \to X)).$$

Actually, the second-order types that we just defined are special cases of the QBF (quantified boolean formulae) arising in complexity theory restricted to implication and universal quantifiers; see Section 12.3. Remarkably, the other connectives $\wedge, \vee, \neg$ and $\exists$ are definable in terms of $\to$ (as a logical connective, $\Rightarrow$) and $\forall$; see Troelstra and Schwichtenberg [43] (Chapter 11).

Remark: The type

$$\mathsf{Nat} = \forall X.\,(X \to ((X \to X) \to X)).$$

can be chosen to represent the type of the natural numbers. The type of the natural numbers can also be chosen to be

$$\forall X.\,((X \to X) \to (X \to X)).$$

This makes essentially no difference but the first choice has some technical advantages.

There is also a new form of *type abstraction*, $\Lambda X.\, M$, and of *type application*, $M\sigma$, where M is a λ-term and σ is a type. There are two new typing rules:

$$\frac{\Gamma \rhd M : \sigma}{\Gamma \rhd (\Lambda X.\, M) \colon \forall X.\, \sigma} \quad \text{(type abstraction)}$$

provided that X does not occur free in any of the types in Γ, and

$$\frac{\Gamma \rhd M : \forall X.\, \sigma}{\Gamma \rhd (M\tau) \colon \sigma[X := \tau]} \quad \text{(type application)}$$

[3] Barendregt and others used Greek letters to denote type variables but we find this confusing.

where τ is any type (and no capture of variable takes place).

From the point of view where types are viewed as propositions and λ-terms are viewed as proofs, type abstraction is an introduction rule and type application is an elimination rule, both for the second-order quantifier $\forall$.

We also have a new reduction rule

$$(\Lambda X.\, M)\sigma \longrightarrow_{\beta\forall} M[X := \sigma]$$

that corresponds to a new form of redundancy in proofs having to do with a $\forall$-elimination immediately following a $\forall$-introduction. Here in the substitution $M[X := \tau]$, all free occurrences of X in M and the types in M are replaced by τ.

Example 3.28. We have

$$\begin{aligned}(\Lambda X.\, \lambda f\colon (X \to X).\, \lambda x\colon X.\, \lambda g\colon \forall Y.\, (Y \to Y).\, gX\, fx)[X := \tau]& \\ = \lambda f\colon (\tau \to \tau).\, \lambda x\colon \tau.\, \lambda g\colon \forall Y.\, (Y \to Y).\, g\tau\, xf.&\end{aligned}$$

For technical details, see Gallier [16].

This new typed λ-calculus is the *second-order polymorphic lambda calculus*. It was invented by Girard (1972) who named it *system F*; see Girard [20, 21], and it is denoted $\boldsymbol{\lambda 2}$ by Barendregt. From the point of view of logic, Girard's system is a proof system for *intuitionistic second-order propositional logic*. We define $\xrightarrow{+}_{\boldsymbol{\lambda 2}}$ and $\xrightarrow{*}_{\boldsymbol{\lambda 2}}$ as the relations

$$\begin{aligned}\xrightarrow{+}_{\boldsymbol{\lambda 2}} &= (\longrightarrow_{\beta} \cup \longrightarrow_{\beta\forall})^{+}\\ \xrightarrow{*}_{\boldsymbol{\lambda 2}} &= (\longrightarrow_{\beta} \cup \longrightarrow_{\beta\forall})^{*}.\end{aligned}$$

A variant of system F was also introduced independently by John Reynolds (1974) but for very different reasons.

The intuition behind terms of type $\forall X.\, \sigma$ is that a term M of type $\forall X.\, \sigma$ is a sort of *generic function* such that for *any* type τ, the function $M\tau$ is a *specialized version* of type $\sigma[X := \tau]$ of M.

For example, M could be the function that appends an element to a list, and for specific types such as the natural numbers Nat, strings String, trees Tree, *etc.*, the functions $M\mathsf{Nat}$, $M\mathsf{String}$, $M\mathsf{Tree}$, are the specialized versions of M to lists of elements having the specific data types $\mathsf{Nat}, \mathsf{String}, \mathsf{Tree}$.

Example 3.29. If σ is any type, we have the closed term

$$\mathbf{A}_{\sigma} = \lambda x\colon \sigma.\, \lambda f\colon (\sigma \to \sigma).\, fx,$$

of type $\sigma \to ((\sigma \to \sigma) \to \sigma)$, such that for every term F of type $\sigma \to \sigma$ and every term a of type σ,

$$\mathbf{A}_{\sigma} aF \xrightarrow{+}_{\boldsymbol{\lambda 2}} Fa.$$

Since $\mathbf{A}_\sigma$ has the same behavior for *all* types σ, it is natural to define the generic function $\mathbf{A}$ given by

$$\mathbf{A} = \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, fx,$$

which has type $\forall X.\, (X \to ((X \to X) \to X))$, and then $\mathbf{A}\sigma$ has the same behavior as $\mathbf{A}_\sigma$. We will see shortly that $\mathbf{A}$ is the Church numeral $\mathbf{c}_1$ in $\boldsymbol{\lambda}\mathbf{2}$.

Remarkably, system F is *strongly normalizing*, which means that *every λ-term typable in system F has a β-normal form*. The proof of this theorem is hard and was one of Girard's accomplishments in his dissertation, Girard [21]. The Church–Rosser property also holds for system F. The proof technique used to prove that system F is strongly normalizing is thoroughly analyzed in Gallier [16].

We stated earlier that deciding whether a simple type σ is provable, that is, whether there is a closed λ-term M that type-checks in the simply-typed λ-calculus such that the judgement $\triangleright M\colon \sigma$ is provable is a hard problem. Indeed Statman proved that this problem is P-space complete; see Statman [40] and Section 12.4.

It is natural so ask whether it is decidable whether given any second-order type σ, there is a closed λ-term M that type-checks in system F such that the judgement $\triangleright M\colon \sigma$ is provable (if σ is viewed as a second-order logical formula, the problem is to decide whether σ is provable). Surprisingly the answer is *no*; this problem (called *inhabitation*) is undecidable. This result was proven by Löb around 1976, see Barendregt [4].

This undecidability result is troubling and at first glance seems paradoxical. Indeed, viewed as a logical formula, a second-order type σ is a QBF (a quantified boolean formula), and if we assign the truth values $\mathbf{F}$ and $\mathbf{T}$ to the boolean variables in it, we can decide whether such a proposition is valid in exponential time and polynomial space (in fact, we will see that later QBF validity is P-space complete). This seems in contradiction with the fact that provability is undecidable.

But the proof system corresponding to system F is an *intuitionistic* proof system, so there are (non-quantifed) propositions that are valid in the truth-value semantics but not provable in intuitionistic propositional logic. The set of second-order propositions provable in intuitionistic second-order logic is a *proper* subset of the set of valid QBF (under the truth-value semantics), and it is *not computable*. So there is no paradox after all.

Going back to the issue of computability of numerical functions, a version of the *Church numerals* can be defined as

$$\mathbf{c}_n = \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f^n(x). \tag{$*_{c1}$}$$

Observe that $\mathbf{c}_n$ has type Nat. Also note that variables x and f now appear in the order x, f in the λ-binder, as opposed to f, x as in Definition 3.10.

Inspired by the definition of $\mathbf{Succ}$ given in Section 3.4, we can define the successor function on the natural numbers as

$$\mathbf{Succ} = \lambda n\colon \mathsf{Nat}.\, \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(nX\, xf).$$

Note how n, which is of type $\mathsf{Nat} = \forall X.\, (X \to ((X \to X) \to X))$, is applied to the type variable X in order to become a term nX of type $X \to ((X \to X) \to X)$, so that $nX\, xf$ has type X, thus $f(nX\, xf)$ also has type X.

For every type σ, every term F of type $\sigma \to \sigma$ and every term a of type σ, we have

$$\begin{aligned}\mathbf{c}_n\sigma\, aF &= \big(\Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f^n(x)\big)\sigma\, aF\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda x\colon \sigma.\, \lambda f\colon (\sigma \to \sigma).\, f^n(x)\big)aF\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} F^n(a);\end{aligned}$$

that is,

$$\mathbf{c}_n\sigma\, aF \xrightarrow{+}_{\boldsymbol{\lambda 2}} F^n(a). \tag{$*_{c2}$}$$

So $\mathbf{c}_n\sigma$ iterates F n times starting with a. As a consequence,

$$\begin{aligned}\mathbf{Succ}\, \mathbf{c}_n &= \big(\lambda n\colon \mathsf{Nat}.\, \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(nX\, xf)\big)\mathbf{c}_n\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(\mathbf{c}_n X\, xf)\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(f^n(x))\\ &= \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f^{n+1}(x) = \mathbf{c}_{n+1}.\end{aligned}$$

We can also define addition of natural numbers as

$$\mathbf{Add} = \lambda m\colon \mathsf{Nat}.\, \lambda n\colon \mathsf{Nat}.\, \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, (mX\, f(nX\, xf))f.$$

Note how m and n, which are of type $\mathsf{Nat} = \forall X.\, (X \to ((X \to X) \to X))$, are applied to the type variable X in order to become terms mX and nX of type $X \to ((X \to X) \to X)$, so that $nX\, xf$ has type X, thus $f(nX\, xf)$ also has type X, and $mX\, f(nX\, xf)$ has type $(X \to X) \to X$, and finally $(mX\, f(nX\, xf))f$ has type X.

Many of the constructions that can be performed in the pure λ-calculus can be mimicked in system F, which explains its expressive power.

For example, for any two second-order types σ and τ, we can define a pairing function $\langle -, - \rangle$ (to be very precise, $\langle -, - \rangle_{\sigma,\tau}$) given by

$$\langle -, - \rangle = \lambda u\colon \sigma.\, \lambda v\colon \tau.\, \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fuv,$$

of type $\sigma \to \big(\tau \to \big(\forall X.\, ((\sigma \to (\tau \to X)) \to X)\big)\big)$. Given any term M of type σ and any term N of type τ, we have

$$\langle -, - \rangle_{\sigma,\tau} MN \xrightarrow{*}_{\boldsymbol{\lambda 2}} \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fMN.$$

Thus we define $\langle M, N \rangle$ as

$$\langle M, N \rangle = \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fMN,$$

and the type

$$\forall X.\, ((\sigma \to (\tau \to X)) \to X)$$

of $\langle M, N\rangle$ is denoted by $\sigma \times \tau$. As a logical formula it is equivalent to $\sigma \wedge \tau$, which means that if we view σ and τ as (second-order) propositions, then

$$\sigma \wedge \tau \equiv \forall X.\, ((\sigma \to (\tau \to X)) \to X)$$

is provable intuitionistically. This is a special case of the result that we mentioned earlier: the connectives $\wedge, \vee, \neg$ and $\exists$ are definable in terms of $\to$ (as a logical connective, $\Rightarrow$) and $\forall$.

Proposition 3.16. *The connectives $\wedge, \vee, \neg, \perp$ and $\exists$ are definable in terms of $\to$ and $\forall$, which means that the following equivalences are provable intuitionistically, where X is not free in σ or τ:*

$$\begin{aligned}
\sigma \wedge \tau &\equiv \forall X.\, \big((\sigma \to (\tau \to X)) \to X\big) \\
\sigma \vee \tau &\equiv \forall X.\, \big((\sigma \to X) \to ((\tau \to X) \to X)\big) \\
\perp &\equiv \forall X.\, X \\
\neg\sigma &\equiv \sigma \to \forall X.\, X \\
\exists Y.\, \sigma &\equiv \forall X.\, \big((\forall Y.\, (\sigma \to X)) \to X\big).
\end{aligned}$$

We leave the proof as an exercise, or see Troelstra and Schwichtenberg [43] (Chapter 11).

Remark: The rule of type application implies that $\perp\to \sigma$ is intuitionistically provable for *all* propositions (types) σ. So in second-order logic there is no difference between minimal and intuitionistic logic.

We also have two projections π_1 and π_2 (to be very precise $\pi_1^{\sigma\times\tau}$ and $\pi_2^{\sigma\times\tau}$) given by

$$\begin{aligned}
\pi_1 &= \lambda g\colon \sigma \times \tau.\, g\sigma(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x) \\
\pi_2 &= \lambda g\colon \sigma \times \tau.\, g\tau(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, y).
\end{aligned}$$

It is easy to check that π_1 has type $(\sigma \times \tau) \to \sigma$ and that π_2 has type $(\sigma \times \tau) \to \tau$. The reader should check that for any M of type σ and any N of type τ we have

$$\pi_1\langle M, N\rangle \xrightarrow{+}_{\boldsymbol{\lambda 2}} M \quad \text{and} \quad \pi_2\langle M, N\rangle \xrightarrow{+}_{\boldsymbol{\lambda 2}} N.$$

Example 3.30. We have

$$\begin{aligned}
\pi_1\langle M, N\rangle &= \big(\lambda g\colon \sigma \times \tau.\, g\sigma(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x)\big)\big(\Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fMN\big) \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fMN\big)\sigma(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x) \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda f\colon \sigma \to (\tau \to \sigma).\, fMN\big)(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x) \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} (\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x)MN \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} (\lambda y\colon \tau.\, M)N \\
&\xrightarrow{+}_{\boldsymbol{\lambda 2}} M.
\end{aligned}$$

The booleans can be defined as

$$\begin{aligned}\mathbf{T} &= \Lambda X.\, \lambda x\colon X.\, \lambda y\colon X.\, x\\ \mathbf{F} &= \Lambda X.\, \lambda x\colon X.\, \lambda y\colon X.\, y,\end{aligned}$$

both of type $\mathsf{Bool} = \forall X.\, (X \to (X \to X))$. We also define $\mathsf{if\ then\ else}$ as

$$\mathsf{if\ then\ else} = \Lambda X.\, \lambda z\colon \mathsf{Bool}.\, zX$$

of type $\forall X.\, \mathsf{Bool} \to (X \to (X \to X))$.

It is easy that for any type σ and any two terms M and N of type σ we have

$$\begin{aligned}(\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma &\xrightarrow{+}_{\boldsymbol{\lambda 2}} M\\ (\mathsf{if}\ \mathbf{F}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma &\xrightarrow{+}_{\boldsymbol{\lambda 2}} N,\end{aligned}$$

where we write $(\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma$ instead of $(\mathsf{if\ then\ else})\ \sigma\, \mathbf{T} MN$ (and similarly for the other term).

Example 3.31. We have

$$\begin{aligned}(\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma &= \big(\Lambda X.\, \lambda z\colon \mathsf{Bool}.\, zX\big)\sigma\, \mathbf{T} MN\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda z\colon \mathsf{Bool}.\, z\sigma\big)\mathbf{T} MN\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\mathbf{T}\sigma\big)MN\\ &= \big((\Lambda X.\, \lambda x\colon X.\, \lambda y\colon X.\, x)\sigma\big)MN\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda x\colon \sigma.\, \lambda y\colon \sigma.\, x\big)MN\\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} M.\end{aligned}$$

Lists, trees, and other inductively data stuctures are also representable in system F; see Girard–Lafond–Taylor [19].

We can also define an iterator **Iter** given by

$$\mathbf{Iter} = \Lambda X.\, \lambda u\colon X.\, \lambda f\colon (X \to X).\, \lambda z\colon \mathsf{Nat}.\, zX\, uf$$

of type $\forall X.\, (X \to ((X \to X) \to (\mathsf{Nat} \to X)))$. The idea is that given f of type $\sigma \to \sigma$ and u of type σ, the term $\mathbf{Iter}\, \sigma\, uf\mathbf{c}_n$ iterates f n times over the input u.

It is easy to show that for any term t of type Nat we have

$$\begin{aligned}\mathbf{Iter}\, \sigma\, uf\mathbf{c}_0 &\xrightarrow{+}_{\boldsymbol{\lambda 2}} u\\ \mathbf{Iter}\, \sigma\, uf(\mathbf{Succ_c}\, t) &\xrightarrow{+}_{\boldsymbol{\lambda 2}} f(\mathbf{Iter}\, \sigma\, uft),\end{aligned}$$

and that

$$\mathbf{Iter}\, \sigma\, uf\mathbf{c}_n \xrightarrow{+}_{\boldsymbol{\lambda 2}} f^n(u).$$

Then mimicking what we did in the pure λ-calculus, we can show that the primitive recursive functions are λ-definable in system F. Actually, higher-order primitive recursion is definable. So, for example, Ackermann's function is definable.

Remarkably, the class of numerical functions definable in system F is a class of (total) computable functions much bigger than the class of primitive recursive functions. This class of functions was characterized by Girard as *the functions that are provably-recursive in a formalization of arithmetic known as intuitionistic second-order arithmetic*; see Girard [21], Troelstra and Schwichtenberg [43] and Girard–Lafond–Taylor [19]. It can also be shown (using a diagonal argument) that there are (total) computable functions not definable in system F.

From a theoretical point of view, every (total) function that we will ever want to compute is definable in system F. However, from a practical point of view, programming in system F is very tedious and usually leads to very inefficient programs. Nevertheless polymorphism is an interesting paradigm which had made its way in certain programming languages.

Type systems even more powerful than system F have been designed, the ultimate system being the *calculus of constructions* due to Huet and Coquand, but these topics are beyond the scope of these notes.

One last comment has to do with the use of the simply-typed λ-calculus as a the core of a programming language. In the early 1970's Dana Scott defined a system named LCF based on the the simply-typed λ-calculus and obtained by adding the natural numbers and the booleans as data types, product types, and a fixed-point operator. Robin Milner then extended LCF, and as a by-product, defined a programming language known as ML, which is the ancestor of most functional programming languages. A masterful and thorough exposition of type theory and its use in programming language design is given in Pierce [33].

We now revisit the problem of defining the partial computable functions.

3.9 Head Normal-Forms and the Partial Computable Functions

One defect of the proof of Theorem 3.11 in the case where a computable function is partial is the use of the Kleene normal form. The difficulty has to do with composition. Given a partial computable function g λ-defined by a closed term G and a partial computable function h λ-defined by a closed term H (for simplicity we assume that both g and h have a single argument), it would be nice if the composition $h \circ g$ was represented by $\lambda x.\, H(Gx)$. This is true if both g and h are total, but false if either g or h is partial as shown by the following example from Barendregt [3] (Chapter 2, §2).

Example 3.32. If g is the function undefined everywhere and h is the constant function 0, then g is λ-defined by $G = \mathbf{K}\boldsymbol{\Omega}$ and h is λ-defined by $H = \mathbf{K}\,\mathbf{c}_0$, with $\boldsymbol{\Omega} = (\lambda x.\,(xx))(\lambda x.\,(xx))$. We have

$$\lambda x.\, H(Gx) = \lambda x.\, \mathbf{K}\,\mathbf{c}_0(\mathbf{K}\boldsymbol{\Omega}x) \xrightarrow{+}_{\beta} \lambda x.\, \mathbf{K}\mathbf{c}_0\boldsymbol{\Omega} \xrightarrow{+}_{\beta} \lambda x.\, \mathbf{c}_0,$$

but $h \circ g = g$ is the function undefined everywhere, and $\lambda x.\, \mathbf{c}_0$ represents the total function h, so $\lambda x.\, H(Gx)$ *does not* λ-define $h \circ g$.

It turns out that the λ-definability of the partial computable functions can be obtained in a more elegant fashion without having recourse to the Kleene normal form by capturing the fact that a function is undefined for some input is a more subtle way. The key notion is the notion of *head normal form*, which is more general than the notion of β-normal form. As a consequence, there a *fewer* λ-terms having *no* head normal form than λ-terms having *no* β-normal form, and we capture a stronger form of divergence.

Recall that a λ-term is either a variable x, or an application (MN), or a λ-abstraction $(\lambda x.\, M)$. We can sharpen this characterization as follows.

Proposition 3.17. *The following properties hold:*

(1) Every application term M is of the form

$$M = (N_1 N_2 \cdots N_{n-1}) N_n, \quad n \geq 2,$$

where N_1 is not an application term.

(2) Every abstraction term M is of the form

$$M = \lambda x_1 \cdots x_n.\, N, \quad n \geq 1,$$

where N is not an abstraction term.

(3) Every λ-term M is of one of the following two forms:

$$M = \lambda x_1 \cdots x_n.\, x M_1 \cdots M_m, \quad m, n \geq 0 \tag{a}$$

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0) M_1 \cdots M_m, \quad m \geq 1, n \geq 0, \tag{b}$$

where x is a variable.

Proof. (1) Suppose that M is an application $M = M_1 M_2$. We proceed by induction on the depth of M_1. For the base case M_1 must be variables and we are done. For the induction step, if M_1 is a λ-abstraction, we are done. If M_1 is an application, then by the induction hypothesis it is of the form

$$M_1 = (N_1 N_2 \cdots N_{n-1}) N_n, \quad n \geq 2,$$

where N_1 is not an application term, and then

$$M = M_1 M_2 = ((N_1 N_2 \cdots N_{n-1}) N_n) M_2 \quad n \geq 2,$$

where N_1 is not an application term.

The proof of (2) is similar.

(3) We proceed by induction on the depth of M. If M is a variable, then we are in Case (a) with $m = n = 0$.

If M is an application, then by (1) it is of the form $M = N_1 N_2 \cdots N_p$ with N_1 not an application term. This means that either N_1 is a variable, in which case we are in Case (a) with $n = 0$, or N_1 is an abstraction, in which case we are in Case (b) also with $n = 0$.

If M is an abstraction $\lambda x.\, N$, then by the induction hypothesis N is of the form (a) or (b), and by adding one more binder λx in front of these expressions we preserve the shape of (a) and (b) by increasing n by 1. □

Example 3.33. The terms, $\mathbf{I}, \mathbf{K}, \mathbf{K}_*, \mathbf{S}$, the Church numerals $\mathbf{c}_n$, if then else, $\langle M, N\rangle$, π_1, π_2, **Iter**, $\mathbf{Succ_c}$, **Add** and **Mult** as in Proposition 3.5, are λ-terms of type (a). However, $\mathbf{Pred_K}$, $\mathbf{\Omega} = (\lambda x.\, (xx))(\lambda x.\, (xx))$, $\mathbf{Y}$ (the Curry $\mathbf{Y}$-combinator), $\mathbf{\Theta}$ (the Turing $\mathbf{\Theta}$-combinator) are of type (b).

Proposition 3.17 motivates the following definition.

Definition 3.32. A λ-term M is a *head normal form* (for short *hnf*) if it is of the form (a), namely

$$M = \lambda x_1 \cdots x_n.\, x M_1 \cdots M_m, \quad m, n \geq 0,$$

where x is a variable called the *head variable*.

A λ-term M *has a head normal form* if there is some head normal form N such that $M \xrightarrow{*}_\beta N$.

In a term M of the form (b),

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0) M_1 \cdots M_m, \quad m \geq 1, n \geq 0,$$

the subterm $(\lambda x.\, M_0) M_1$ is called the *head redex* of M.

Example 3.34. In addition to the terms of type (a) that we listed after Proposition 3.17, the term $\lambda x.\, x\mathbf{\Omega}$ is a head normal form. It is the head normal form of the term $\lambda x.\, (\mathbf{I}x)\mathbf{\Omega}$, which has *no* β-normal form.

Not every term has a head normal form. For example, the term

$$\mathbf{\Omega} = (\lambda x.\, (xx))(\lambda x.\, (xx))$$

has no head normal form. Every β-normal form must be a head normal form, but the converse is false as we saw with

$$M = \lambda x.\, x\mathbf{\Omega},$$

which is a head normal form but has no β-normal form.

Note that a head redex of a term is a leftmost redex, but not conversely, as shown by the term $\lambda x.\, x((\lambda y.\, y)x)$.

A term may have more than one head normal form but here is a way of obtaining a head normal form (if there is one) in a systematic fashion.

Definition 3.33. The relation $\longrightarrow_h$, called *one-step head reduction*, is defined as follows: For any two terms M and N, if M contains a head redex $(\lambda x.\, M_0)M_1$, which means that M is of the form

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0)M_1 \cdots M_m, \quad m \geq 1, n \geq 0,$$

then $M \longrightarrow_h N$ with

$$N = \lambda x_1 \cdots x_n.\, (M_0[x := M_1])M_2 \cdots M_m.$$

We denote by $\xrightarrow{+}_h$ the transitive closure of $\longrightarrow_h$ and by $\xrightarrow{*}_h$ the reflexive and transitive closure of $\longrightarrow_h$.

Given a term M containing a head redex, the *head reduction sequence of* M is the uniquely determined sequence of one-step head reductions

$$M = M_0 \longrightarrow_h M_1 \longrightarrow_h \cdots \longrightarrow_h M_n \longrightarrow_h \cdots .$$

If the head reduction sequence reaches a term M_n which is a head normal form we say that the sequence *terminates*, and otherwise we say that M has an *infinite* head reduction.

The following result is shown in Barendregt [3] (Chapter 8, §3).

Theorem 3.18. *(Wadsworth) A λ-term M has a head normal form if and only if the head reduction sequence terminates.*

In some intuitive sense, a λ-term M that does not have any head normal form has a strong divergence behavior with respect to β-reduction.

Remark: There is a notion more general than the notion of head normal form which comes up in functional languages (for example, Haskell). A λ-term M is a *weak head normal form* if it of one of the two forms

$$\lambda x.\, N \quad \text{or} \quad yN_1 \cdots N_m$$

where y is a variable These are exactly the terms that do not have a redex of the form $(\lambda x.\, M_0)M_1N_1 \cdots N_m$. Every head normal form is a weak head normal form, but there are many more weak head normal forms than there are head normal forms since a term of the form $\lambda x.\, N$ where N is *arbitrary* is a weak head normal form, but not a head normal form unless N is of the form $\lambda x_1 \cdots x_n.\, xM_1 \cdots M_m$, with $m, n \geq 0$.

Reducing to a weak head normal form is a lazy evaluation strategy.

There is also another useful notion which turns out to be equivalent to having a head normal form.

Definition 3.34. A closed λ-term M is *solvable* if there are closed terms $N_1, \ldots, N_n$ such that

$$MN_1 \cdots N_n \xrightarrow{*}_\beta \mathbf{I}.$$

A λ-term M with free variables $x_1, \ldots, x_m$ is *solvable* if the closed term $\lambda x_1 \cdots x_m.\, M$ is solvable. A term is *unsolvable* if it is not solvable.

The following result is shown in Barendregt [3] (Chapter 8, §3).

Theorem 3.19. *(Wadsworth) A λ-term M has a head normal form if and only if is it solvable.*

Actually, the proof that having a head normal form implies solvable is not hard.

We are now ready to revise the notion of λ-definability of numerical functions.

Definition 3.35. A function (partial or total) $f\colon \mathbb{N}^n \to \mathbb{N}$ is *strongly λ-definable* if for all $m_1, \ldots, m_n \in \mathbb{N}$, there is a combinator (a closed λ-term) F with the following properties:

(1) If the value $f(m_1, \ldots, m_n)$ is defined, then $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to the β-normal form $\mathbf{c}_{f(m_1,\ldots,m_n)}$.

(2) If $f(m_1, \ldots, m_n)$ is undefined, then $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ has no head normal form, or equivalently, is unsolvable.

Observe that in Case (2), when the value $f(m_1, \ldots, m_n)$ is undefined, the divergence behavior of $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ is stronger than in Definition 3.18. Not only $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ has no β-normal form, but actually it has *no head normal form*.

The following result is proven in Barendregt [3] (Chapter 8, §4). The proof does not use the Kleene normal form. Instead, it makes clever use of the term **KII**. Another proof is given in Krivine [26] (Chapter II).

Theorem 3.20. *Every partial or total computable function is strongly λ-definable. Conversely, every strongly λ-definable function is partial computable.*

Making sure that a composition $g \circ (h_1, \ldots, h_m)$ is defined for some input $x_1, \ldots, x_n$ iff all the $h_i(x_1, \ldots, x_n)$ and $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ are defined is tricky. The term **KII** comes to the rescue! If g is strongly λ-definable by G and the h_i are strongly λ-definable by H_i, then it can be shown that the combinator F given by

$$F = \lambda x_1 \cdots x_n.\, (H_1 x_1 \cdots x_n \mathbf{KII}) \cdots (H_m x_1 \cdots x_n \mathbf{KII})(G(H_1 x_1 \cdots x_n) \cdots (G(H_m x_1 \cdots x_n))$$

strongly λ-defines F; see Barendregt [3] (Chapter 8, Lemma 8.4.6).

Chapter 4

Universal RAM Programs and Undecidability of the Halting Problem

The goal of this chapter is to prove three of the main results of computability theory:

(1) The undecidability of the halting problem for RAM programs (and Turing machines).

(2) The existence of universal RAM programs.

(3) The existence of the Kleene T-predicate.

All three require the ability to code a RAM program as a natural number. Gödel pioneered the technique of encoding objects such as proofs as natural numbers in his famous paper on the (first) incompleteness theorem (1931). One of the technical issues is to code (pack) a tuple of natural numbers as a single natural number, so that the numbers being packed can be retrieved. Gödel designed a fancy function whose defintion does not involve recursion (Gödel's β function; see Kleene [24] or Shoenfield [39]). For our purposes, a simpler function J due to Cantor packing two natural numbers m and n as a single natural number $J(m, n)$ suffices.

Another technical issue is the fact it is possible to reduce most of computability theory to numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$, and even to functions $f\colon \mathbb{N} \to \mathbb{N}$. Indeed, there are primitive recursive coding and decoding functions $D_k\colon \Sigma^* \to \mathbb{N}$ and $C_k\colon \mathbb{N} \to \Sigma^*$ such that $C_k \circ D_k = \mathrm{id}_{\Sigma^*}$, where $\Sigma = \{a_1, \ldots, a_k\}$. It is simpler to code programs (or Turing machines) taking natural numbers as input.

Unfortunately, these coding techniques are very tedious so we advise the reader not to get bogged down with technical details upon first reading.

4.1 Pairing Functions

Pairing functions are used to encode pairs of integers into single integers, or more generally, finite sequences of integers into single integers. We begin by exhibiting a bijective pairing function $J\colon \mathbb{N}^2 \to \mathbb{N}$. The function J has the graph partially showed below:

$$\begin{tikzcd}
y \\
4 & 10 \arrow[dr] \\
3 & 6 \arrow[dr] & 11 \arrow[dr] \\
2 & 3 \arrow[dr] & 7 \arrow[dr] & 12 \arrow[dr] \\
1 & 1 \arrow[dr] & 4 \arrow[dr] & 8 \arrow[dr] & 13 \arrow[dr] \\
0 & 0 & 2 & 5 & 9 & 14 \\
 & 0 & 1 & 2 & 3 & 4 & x
\end{tikzcd}$$

The function J corresponds to a certain way of enumerating pairs of integers (x, y). Note that the value of $x + y$ is constant along each descending diagonal, and consequently, we have

$$\begin{aligned}
J(x, y) &= 1 + 2 + \cdots + (x + y) + x, \\
&= ((x + y)(x + y + 1) + 2x)/2, \\
&= ((x + y)^2 + 3x + y)/2,
\end{aligned}$$

that is,

$$J(x, y) = ((x + y)^2 + 3x + y)/2.$$

For example, $J(0, 3) = 6,\ J(1, 2) = 7,\ J(2, 2) = 12,\ J(3, 1) = 13,\ J(4, 0) = 14.$

If we can prove can J is a bijection, then we can define $K\colon \mathbb{N} \to \mathbb{N}$ and $L\colon \mathbb{N} \to \mathbb{N}$ as the projection functions onto the axes, that is, the unique functions such that

$$K(J(a, b)) = a \quad \text{and} \quad L(J(a, b)) = b,$$

for all $a, b \in \mathbb{N}$. For example, $K(11) = 1$, and $L(11) = 3$; $K(12) = 2$, and $L(12) = 2$; $K(13) = 3$ and $L(13) = 1$.

Definition 4.1. The *pairing function* $J\colon \mathbb{N}^2 \to \mathbb{N}$ is defined by

$$J(x, y) = ((x + y)^2 + 3x + y)/2 \quad \text{for all } x, y \in \mathbb{N}.$$

The functions $K\colon \mathbb{N} \to \mathbb{N}$ and $L\colon \mathbb{N} \to \mathbb{N}$ are the *projection functions* onto the axes, that is, the unique functions such that

$$K(J(a, b)) = a \quad \text{and} \quad L(J(a, b)) = b,$$

for all $a, b \in \mathbb{N}$.

The functions J, K, L are called *Cantor's pairing functions*. They were used by Cantor to prove that the set $\mathbb{Q}$ of rational numbers is countable.

Clearly, J is primitive recursive, since it is given by a polynomial. In Definition 4.1, we implicitly assumed that J is bijective in order to define K and L.

Neither injectivity nor surjectivity of J are easy to prove.

Theorem 4.1. *The pairing function $J\colon \mathbb{N}^2 \to \mathbb{N}$ defined by*

$$J(x,y) = ((x+y)^2 + 3x + y)/2 \quad \textit{for all } x, y \in \mathbb{N}$$

is a bijection. There are unique functions $K\colon \mathbb{N} \to \mathbb{N}$ and $L\colon \mathbb{N} \to \mathbb{N}$ such that

$$\begin{aligned} K(J(a,b)) &= a \\ L(J(a,b)) &= b \\ J(K(z), L(z)) &= z. \end{aligned}$$

for all $a, b, z \in \mathbb{N}$.

Sketch of proof. We follow Martin Davis [7]. The first step is to prove that for any $z \in \mathbb{N}$, if $J(m,n) = z$, then

$$8z + 1 = (2m + 2n + 1)^2 + 8m. \tag{a}$$

From the above equation we can deduce that

$$2m + 2n + 1 \leq \sqrt{8z+1} < 2m + 2n + 3. \tag{b}$$

If $x \mapsto \lfloor x \rfloor$ is the function from $\mathbb{R}$ to $\mathbb{N}$ (the *floor function*), where $\lfloor x \rfloor$ is the largest integer $\leq x$ (for example, $\lfloor 2.3 \rfloor = 2$, $\lfloor \sqrt{2} \rfloor = 1$), we can prove that

$$\lfloor \sqrt{8z+1} \rfloor + 1 = 2m + 2n + 2 \quad \text{or} \quad \lfloor \sqrt{8z+1} \rfloor + 1 = 2m + 2n + 3,$$

so that

$$\lfloor (\lfloor \sqrt{8z+1} \rfloor + 1)/2 \rfloor = m + n + 1. \tag{c}$$

From Equation (c) we obtain

$$m + n = \lfloor (\lfloor \sqrt{8z+1} \rfloor + 1)/2 \rfloor - 1. \tag{d}$$

Since $J(m,n) = z$ means that

$$2z = (m+n)^2 + 3m + n,$$

that is,

$$3m + n = 2z - (m+n)^2, \tag{e}$$

we deduce from (d) and (e) that m and n are solutions of the system

$$\begin{aligned} m+n &= \lfloor(\lfloor\sqrt{8z+1}\rfloor+1)/2\rfloor-1\\ 3m+n &= 2z-(\lfloor(\lfloor\sqrt{8z+1}\rfloor+1)/2\rfloor-1)^2. \end{aligned}$$

If we let

$$\begin{aligned} Q_1(z) &= \lfloor(\lfloor\sqrt{8z+1}\rfloor+1)/2\rfloor-1\\ Q_2(z) &= 2z-(\lfloor(\lfloor\sqrt{8z+1}\rfloor+1)/2\rfloor-1)^2 = 2z-(Q_1(z))^2, \end{aligned}$$

then we can prove that the number $Q_2(z)-Q_1(z)$ is even and that

$$\begin{aligned} m &= \frac{1}{2}(Q_2(z)-Q_1(z)) = K(z)\\ n &= Q_1(z)-\frac{1}{2}(Q_2(z)-Q_1(z)) = L(z). \end{aligned}$$

Consequently, if $z=J(m,n)$, then $m=K(z)$ and $n=L(z)$ as above, showing that m and n are unique and thus that J is injective. The above also proves that J,K,L satisfy the equations.

$$\begin{aligned} m &= K(J(m,n))\\ n &= L(J(m,n)). \end{aligned}$$

It remains to prove that J is surjective. Let $z\in\mathbb{N}$ be any natural number and let $r\in\mathbb{N}$ be the largest number such that

$$1+2+\cdots+r\leq z.$$

If we let

$$x = z-(1+2+\cdots+r), \tag{f}$$

then $x\leq r$, since otherwise $x\geq r+1$, and then (f) implies that $1+2+\cdots+r+(r+1)\leq z$, contradicting the maximality of r. Let $y=r-x\geq 0$. Then we have

$$\begin{aligned} z &= (1+2+\cdots+r)+x\\ &= (1+2+\cdots+x+y)+x\\ &= \frac{1}{2}(x+y)(x+y+1)+x\\ &= J(x,y). \end{aligned}$$

Therefore J is surjective. But

$$\begin{aligned} x =& K(J(x,y)) = K(z)\\ y =& L(J(x,y)) = L(z), \end{aligned}$$

so

$$J(K(z),L(z)) = z,$$

as claimed. □

Theorem 4.1 yields explicit formulae for K and L. If we define

$$\begin{aligned} Q_1(z) &= \lfloor(\lfloor\sqrt{8z+1}\rfloor + 1)/2\rfloor - 1 \\ Q_2(z) &= 2z - (Q_1(z))^2, \end{aligned}$$

then we have

$$\begin{aligned} K(z) &= \frac{1}{2}(Q_2(z) - Q_1(z)) \\ L(z) &= Q_1(z) - \frac{1}{2}(Q_2(z) - Q_1(z)). \end{aligned}$$

In the above formula, the function $m \mapsto \lfloor\sqrt{m}\rfloor$ yields the largest integer s such that $s^2 \leq m$. These formulae also show that K and L are primitive recursive. An easier way to see this is to observe that since J is a bijection,

$$x \leq J(x, y) \quad \text{and} \quad y \leq J(x, y),$$

we have

$$K(z) = \min(x \leq z)(\exists y \leq z)[J(x, y) = z],$$

and

$$L(z) = \min(y \leq z)(\exists x \leq z)[J(x, y) = z].$$

Therefore, by the results of Section 1.9, K and L are primitive recursive.

Observe that the equations $K(J(a, b)) = a$ and $L(J(a, b)) = b$ assert that J is injective and that the equation $J(K(z), L(z)) = z$ assert that J is surjective, but the problem is that the definition of J does not obviously imply these properties so it is necessary to construct K and L as done in the proof of Theorem 4.1.

The pairing function $J(x, y)$ is also denoted as $\langle x, y\rangle$, and K and L are also denoted as Π_1 and Π_2. The notation $\langle x, y\rangle$ is "intentionally ambiguous," in the sense that it can be interpreted as the actual ordered pair consisting of the two numbers x and y, or as *the number* $\langle x, y\rangle = J(x, y)$ that encodes the pair consisting of the two numbers x and y. The context should make it clear which interpretation is intended. In this chapter and the next, it is the number (code) interpretation.

We can define bijections between $\mathbb{N}^n$ and $\mathbb{N}$ by induction for all $n \geq 1$.

Definition 4.2. The function $\langle -, \ldots, -\rangle_n \colon \mathbb{N}^n \to \mathbb{N}$ called an *extended pairing function* is defined as follows. We let

$$\begin{aligned} \langle z\rangle_1 &= z \\ \langle x_1, x_2\rangle_2 &= \langle x_1, x_2\rangle, \end{aligned}$$

and

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n,$$

for all $z, x_2, \ldots, x_{n+1} \in \mathbb{N}$.

Again we stress that $\langle x_1, \ldots, x_n\rangle_n$ is a *natural number*. For example.

$$\begin{aligned}\langle x_1, x_2, x_3\rangle_3 &= \langle x_1, \langle x_2, x_3\rangle\rangle_2\\ &= \langle x_1, \langle x_2, x_3\rangle\rangle\\ \langle x_1, x_2, x_3, x_4\rangle_4 &= \langle x_1, x_2, \langle x_3, x_4\rangle\rangle_3\\ &= \langle x_1, \langle x_2, \langle x_3, x_4\rangle\rangle\rangle\\ \langle x_1, x_2, x_3, x_4, x_5\rangle_5 &= \langle x_1, x_2, x_3, \langle x_4, x_5\rangle\rangle_4\\ &= \langle x_1, \langle x_2, \langle x_3, \langle x_4, x_5\rangle\rangle\rangle\rangle.\end{aligned}$$

It can be shown by induction on n that

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \langle x_2, \ldots, x_{n+1}\rangle_n\rangle. \tag{$*$}$$

Observe that if $z = \langle x_1, \ldots, x_n\rangle_n$, then $x_1 = \Pi_1(z)$, $x_2 = \Pi_1(\Pi_2(z))$, $x_3 = \Pi_1(\Pi_2(\Pi_2(z)))$, $x_4 = \Pi_1(\Pi_2(\Pi_2(\Pi_2(z))))$, $x_5 = \Pi_2(\Pi_2(\Pi_2(\Pi_2(z))))$.

We can also define a uniform projection function $\Pi\colon \mathbb{N}^3 \to \mathbb{N}$ with the following property: if $z = \langle x_1, \ldots, x_n\rangle_n$, with $n \geq 2$, then

$$\Pi(i, n, z) = x_i \quad \text{for all } i, \text{ where } 1 \leq i \leq n.$$

The idea is to view z as an n-tuple, and $\Pi(i, n, z)$ as the i-th component of that n-tuple, but if z, n and i do not fit this interpretation, the function must be still be defined and we give it a "crazy" value by default using some simple primitive recursive clauses.

Definition 4.3. The uniform projection function $\Pi\colon \mathbb{N}^3 \to \mathbb{N}$ is defined by cases as follows:

$$\begin{aligned}\Pi(i, 0, z) &= 0, \quad \text{for all } i \geq 0,\\ \Pi(i, 1, z) &= z, \quad \text{for all } i \geq 0,\\ \Pi(i, 2, z) &= \Pi_1(z), \quad \text{if } 0 \leq i \leq 1,\\ \Pi(i, 2, z) &= \Pi_2(z), \quad \text{for all } i \geq 2,\end{aligned}$$

and for all $n \geq 2$,

$$\Pi(i, n+1, z) = \begin{cases} \Pi(i, n, z) & \text{if } 0 \leq i < n,\\ \Pi_1(\Pi(n, n, z)) & \text{if } i = n,\\ \Pi_2(\Pi(n, n, z)) & \text{if } i > n.\end{cases}$$

By the results of Section 1.9, this is a legitimate primitive recursive definition. If z is the code $\langle x_1, \ldots, x_{n+1}\rangle_{n+1}$ for the $(n+1)$-tuple $(x_1, \ldots, x_{n+1})$ with $n \geq 2$, then for $0 \leq i < n$, the clause of Definition 4.3 that applies is

$$\Pi(i, n+1, z) = \Pi(i, n, z),$$

and since

$$\langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1} = \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n,$$

we have

$$\begin{aligned}\Pi(i, n+1, \langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1}) &= \Pi(i, n+1, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n)\\ &= \Pi(i, n, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n),\end{aligned}$$

and since $\langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n$ codes an n-tuple, for $i = 1, \ldots, n-1$, the value returned is indeed x_i. If $i = n$, then the clause that applies is

$$\Pi(n, n+1, z) = \Pi_1(\Pi(n, n, z)),$$

so we have

$$\begin{aligned}\Pi(n, n+1, \langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1}) &= \Pi(n, n+1, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n)\\ &= \Pi_1(\Pi(n, n, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n))\\ &= \Pi_1(\langle x_n, x_{n+1}\rangle)\\ &= x_n.\end{aligned}$$

Finally, if $i = n+1$, then the clause that applies is

$$\Pi(n+1, n+1, z) = \Pi_2(\Pi(n, n, z)),$$

so we have

$$\begin{aligned}\Pi(n+1, n+1, \langle x_1, \ldots, x_n, x_{n+1}\rangle_{n+1}) &= \Pi(n+1, n+1, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n)\\ &= \Pi_2(\Pi(n, n, \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1}\rangle\rangle_n))\\ &= \Pi_2(\langle x_n, x_{n+1}\rangle)\\ &= x_{n+1}.\end{aligned}$$

When $i = 0$ or $i > n+1$, we get "bogus" values.

Remark: One might argue that it would have been preferable to order the arguments of Π as (n, i, z) rather than (i, n, z). We use the order (i, n, z) in conformity with Machtey and Young [29].

Some basic properties of Π are given as exercises. In particular, the following properties are easily shown:

(a) $\langle 0, \ldots, 0\rangle_n = 0$, $\langle x, 0\rangle = \langle x, 0, \ldots, 0\rangle_n$;

(b) $\Pi(0, n, z) = \Pi(1, n, z)$ and $\Pi(i, n, z) = \Pi(n, n, z)$, for all $i \geq n$ and all $n, z \in \mathbb{N}$;

(c) $\langle \Pi(1, n, z), \ldots, \Pi(n, n, z)\rangle_n = z$, for all $n \geq 1$ and all $z \in \mathbb{N}$;

(d) $\Pi(i, n, z) \leq z$, for all $i, n, z \in \mathbb{N}$;

(e) There is a primitive recursive function Large, such that,

$$\Pi(i, n+1, \mathrm{Large}(n+1, z)) = z,$$

for $i, n, z \in \mathbb{N}$.

As a first application, we observe that we need only consider partial computable functions (partial recursive functions)[1] of a single argument. Indeed, let $\varphi\colon \mathbb{N}^n \to \mathbb{N}$ be a partial computable function of $n \geq 2$ arguments. Let $\overline{\varphi}\colon \mathbb{N} \to \mathbb{N}$ be the function given by

$$\overline{\varphi}(z) = \varphi(\Pi(1, n, z), \ldots, \Pi(n, n, z)),$$

for all $z \in \mathbb{N}$. Then $\overline{\varphi}$ is a partial computable function of a single argument, and φ can be recovered from $\overline{\varphi}$, since

$$\varphi(x_1, \ldots, x_n) = \overline{\varphi}(\langle x_1, \ldots, x_n \rangle_n).$$

Thus, using $\langle -, \cdots, - \rangle_n$ and Π as coding and decoding functions, we can restrict our attention to functions of a single argument.

From now on, since the context usually makes it clear we abbreviate $\langle x_1, \ldots, x_n \rangle_n$ as $\langle x_1, \ldots, x_n \rangle$.

Pairing functions can also be used to prove that certain functions are primitive recursive, even though their definition is not a legal primitive recursive definition. For example, consider the *Fibonacci function* defined as follows:

$$\begin{aligned} f(0) &= 1, \\ f(1) &= 1, \\ f(n+2) &= f(n+1) + f(n), \end{aligned}$$

for all $n \in \mathbb{N}$. This is not a legal primitive recursive definition, since $f(n+2)$ depends both on $f(n+1)$ and $f(n)$. In a primitive recursive definition, $g(y+1, \overline{x})$ is only allowed to depend upon $g(y, \overline{x})$, where $\overline{x}$ is an abbrevation for $(x_2, \ldots, x_m)$.

Definition 4.4. Given any function $f\colon \mathbb{N}^n \to \mathbb{N}$, the function $\overline{f}\colon \mathbb{N}^{n+1} \to \mathbb{N}$ defined such that

$$\overline{f}(y, \overline{x}) = \langle f(0, \overline{x}), \ldots, f(y, \overline{x}) \rangle_{y+1}$$

is called the *course-of-value function* for f.

The following proposition holds.

Proposition 4.2. *Given any function $f\colon \mathbb{N}^n \to \mathbb{N}$, if f is primitive recursive, then so is $\overline{f}$.*

Proof. First it is necessary to define a function *con* such that if $x = \langle x_1, \ldots, x_m \rangle$ and $y = \langle y_1, \ldots, y_n \rangle$, where $m, n \geq 1$, then

$$con(m, x, y) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle.$$

[1] The term *partial recursive* is now considered old-fashion. Many researchers have switched to the term *partial computable.*

This fact is left as an exercise. Now, if f is primitive recursive, let

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

showing that $\overline{f}$ is primitive recursive. Conversely, if $\overline{f}$ is primitive recursive, then

$$f(y,\overline{x}) = \Pi(y+1,y+1,\overline{f}(y,\overline{x})),$$

and so, f is primitive recursive. □

Remark: Why is it that

$$\overline{f}(y+1,\overline{x}) = \langle \overline{f}(y,\overline{x}), f(y+1,\overline{x})\rangle$$

does not work? Check the definition of $\langle x_1,\ldots,x_n\rangle_n$.

We define *course-of-value recursion* as follows.

Definition 4.5. Given any two functions $g\colon \mathbb{N}^n \to \mathbb{N}$ and $h\colon \mathbb{N}^{n+2} \to \mathbb{N}$, the function $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by *course-of-value recursion* from g and h if

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

The following proposition holds.

Proposition 4.3. *If $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by course-of-value recursion from g and h and g, h are primitive recursive, then f is primitive recursive.*

Proof. We prove that $\overline{f}$ is primitive recursive. Then by Proposition 4.2, f is also primitive recursive. To prove that $\overline{f}$ is primitive recursive, observe that

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

□

When we use Proposition 4.3 to prove that a function is primitive recursive, we rarely bother to construct a formal course-of-value recursion. Instead, we simply indicate how the value of $f(y+1,\overline{x})$ can be obtained in a primitive recursive manner from $f(0,\overline{x})$ through $f(y,\overline{x})$. Thus, an informal use of Proposition 4.3 shows that the Fibonacci function is primitive recursive. A rigorous proof of this fact is left as an exercise.

Next we show that there exist coding and decoding functions between Σ^* and $\{a_1\}^*$, and that partial computable functions over Σ^* can be recoded as partial computable functions over $\{a_1\}^*$. Since $\{a_1\}^*$ is isomorphic to $\mathbb{N}$, this shows that we can restrict out attention to functions defined over $\mathbb{N}$.

4.2 Equivalence of Alphabets

Given an alphabet $\Sigma = \{a_1, \ldots, a_k\}$, strings over Σ can be ordered by viewing strings as numbers in a number system where the digits are $a_1, \ldots, a_k$. In this number system, which is almost the number system with base k, the string a_1 corresponds to zero, and a_k to $k-1$. Hence, we have a kind of shifted number system in base k. The total order on Σ^* induced by this number system is defined so that u precedes v if $|u| < |v|$, and if $|u| = |v|$, then u comes before v in the lexicographic ordering. For example, if $\Sigma = \{a, b, c\}$, a listing of Σ^* in the ordering corresponding to the number system begins with

$$\begin{gathered} a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, \\ aaa, aab, aac, aba, abb, abc, \ldots. \end{gathered}$$

This ordering induces a function from Σ^* to $\mathbb{N}$ which is a bijection. Indeed, if $u = a_{i_1} \cdots a_{i_n}$, this function $f\colon \Sigma^* \to \mathbb{N}$ is given by

$$f(u) = i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n.$$

Since we also want a decoding function, we define the coding function $C_k\colon \Sigma^* \to \Sigma^*$ as follows:

$C_k(\epsilon) = \epsilon$, and if $u = a_{i_1} \cdots a_{i_n}$, then

$$C_k(u) = a_1^{i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n}.$$

The function C_k is primitive recursive, because

$$\begin{aligned} C_k(\epsilon) &= \epsilon, \\ C_k(x a_i) &= C_k(x)^k a_1^i. \end{aligned}$$

The inverse of C_k is a function $D_k\colon \{a_1\}^* \to \Sigma^*$. However, primitive recursive functions are total, and we need to extend D_k to Σ^*. This is easily done by letting

$$D_k(x) = D_k(a_1^{|x|})$$

for all $x \in \Sigma^*$. It remains to define D_k by primitive recursion over $\Sigma^* = \{a_1, \ldots, a_k\}^*$. For this, we introduce three auxiliary functions p, q, r, defined as follows. Let

$$\begin{aligned} p(\epsilon) &= \epsilon, \\ p(x a_i) &= x a_i, \quad \text{if } i \neq k, \\ p(x a_k) &= p(x). \end{aligned}$$

Note that $p(x)$ is the result of deleting consecutive a_k's in the tail of x. Let

$$\begin{aligned} q(\epsilon) &= \epsilon, \\ q(x a_i) &= q(x) a_1. \end{aligned}$$

Note that $q(x) = a_1^{|x|}$. Finally, let

$$\begin{aligned} r(\epsilon) &= a_1, \\ r(xa_i) &= xa_{i+1}, \quad \text{if } i \neq k, \\ r(xa_k) &= xa_k. \end{aligned}$$

The function r is almost the successor function for the ordering. Then the trick is that $D_k(xa_i)$ is the successor of $D_k(x)$ in the ordering so usually $D_k(xa_i) = r(D_k(x))$, except if

$$D_k(x) = ya_j a_k^n$$

with $j \neq k$, since the successor of $ya_j a_k^n$ is $ya_{j+1}a_1^n$. Thus, we have

$$\begin{aligned} D_k(\epsilon) &= \epsilon, \\ D_k(xa_i) &= r(p(D_k(x)))q(D_k(x) - p(D_k(x))), \quad a_i \in \Sigma. \end{aligned}$$

Then both C_k and D_k are primitive recursive, and $D_k \circ C_k = \text{id}$. Here

$$u - v = \begin{cases} \epsilon & \text{if } |u| \leq |v| \\ w & \text{if } u = xw \text{ and } |x| = |v|. \end{cases}$$

In other words, $u - v$ is u with its first $|v|$ letters deleted. We can show that this function can be defined by primitive recursion by first defining $\text{rdiff}(u, v)$ as v with its first $|u|$ letters deleted, and then

$$u - v = \text{rdiff}(v, u).$$

To define rdiff, we use *tail* given by

$$\begin{aligned} tail(\epsilon) &= \epsilon \\ tail(a_i u) &= u, \quad a_i \in \Sigma,\ u \in \Sigma^*. \end{aligned}$$

We proved in Section 1.7 that *tail* is primitive recursive. Then

$$\begin{aligned} \text{rdiff}(\epsilon, v) &= v \\ \text{rdiff}(ua_i, v) &= \text{rdiff}(u, tail(v)), \quad a_i \in \Sigma. \end{aligned}$$

We leave as an exercise to put all these definitions into the proper format of primitive recursion using projections.

Let $\varphi\colon (\Sigma^*)^n \to \Sigma^*$ be a partial function over Σ^*, and let $\varphi^+\colon (\{a_1\}^*)^n \to \{a_1\}^*$ be the function given by

$$\varphi^+(x_1, \ldots, x_n) = C_k(\varphi(D_k(x_1), \ldots, D_k(x_n))).$$

Also, for any partial function $\psi\colon (\{a_1\}^*)^n \to \{a_1\}^*$, let $\psi^\sharp\colon (\Sigma^*)^n \to \Sigma^*$ be the function given by

$$\psi^\sharp(x_1, \ldots, x_n) = D_k(\psi(C_k(x_1), \ldots, C_k(x_n))).$$

We claim that if ψ is a partial computable function over $(\{a_1\}^*)^n$, then $\psi^\sharp$ is partial computable over $(\Sigma^*)^n$, and that if φ is a partial computable function over $(\Sigma^*)^n$, then φ^+ is partial computable over $(\{a_1\}^*)^n$.

The function ψ can be extended to $(\Sigma^*)^n$ by letting

$$\psi(x_1, \ldots, x_n) = \psi(a_1^{|x_1|}, \ldots, a_1^{|x_n|})$$

for all $x_1, \ldots, x_n \in \Sigma^*$, and so, if ψ is partial computable, then so is the extended function, by composition. It follows that if ψ is partial (or primitive) recursive, then so is $\psi^\sharp$.

This seems equally obvious for φ and φ^+, but there is a difficulty. The problem is that φ^+ is defined as a composition of functions over Σ^*. We have to show how φ^+ can be defined directly over $\{a_1\}^*$ without using any additional alphabet symbols. This is done in Machtey and Young [29], see Section 2.2, Lemma 2.2.3.

4.3 Coding of RAM Programs; The Halting Problem

In this section we present a specific encoding of RAM programs which allows us *to treat programs as integers*. This encoding will allow us to prove one of the most important results of computability theory first proven by Turing for Turing machines (1936-1937), the *undecidability of the halting problem* for RAM programs (and Turing machines).

Encoding programs as integers also allows us to have programs that take other programs as input, and we obtain a *universal program*. Universal programs have the property that given two inputs, the first one being the code of a program and the second one an input data, the universal program simulates the actions of the encoded program on the input data. A coding scheme is also called an indexing or a Gödel numbering, in honor to Gödel, who invented this technique.

From results of the previous chapter, without loss of generality, we can restrict out attention to RAM programs computing partial functions of one argument over $\mathbb{N}$. Furthermore, we only need the following kinds of instructions, each instruction being coded as shown below. Since we are considering functions over the natural numbers, which corresponds to a one-letter alphabet, there is only one kind of instruction of the form `add` and `jmp` (`add` increments by 1 the contents of the specified register Rj).

Recall that a conditional jump causes a jump to the closest address Nk above or below iff Rj is nonzero, and if Rj is null, the next instruction is executed. We assume that all lines in a RAM program are numbered. This is always feasible, by labeling unnamed instructions with a new and unused line number.

Definition 4.6. Instructions of a RAM program (operating on $\mathbb{N}$) are coded as follows:

$$\begin{array}{llcll}
Ni & & \mathtt{add} & Rj & \textit{code} = \langle 1, i, j, 0\rangle \\
Ni & & \mathtt{tail} & Rj & \textit{code} = \langle 2, i, j, 0\rangle \\
Ni & & \mathtt{continue} & & \textit{code} = \langle 3, i, 1, 0\rangle \\
Ni & Rj & \mathtt{jmp} & Nka & \textit{code} = \langle 4, i, j, k\rangle \\
Ni & Rj & \mathtt{jmp} & Nkb & \textit{code} = \langle 5, i, j, k\rangle
\end{array}$$

The code of an instruction I is denoted as $\#I$.

To simplify the notation, we introduce the following decoding primitive recursive functions Typ, LNum, Reg, and Jmp, defined as follows:

$$\begin{aligned}
\mathrm{Typ}(x) &= \Pi(1, 4, x), \\
\mathrm{LNum}(x) &= \Pi(2, 4, x), \\
\mathrm{Reg}(x) &= \Pi(3, 4, x), \\
\mathrm{Jmp}(x) &= \Pi(4, 4, x).
\end{aligned}$$

The functions yield the type, line number, register name, and line number jumped to, if any, for an instruction coded by x. Note that we have no need to interpret the values of these functions if x does not code an instruction.

We can define the primitive recursive predicate INST, such that $\mathrm{INST}(x)$ holds iff x codes an instruction. First, we need the connective $\Rightarrow$ (*implies*), defined such that

$$P \Rightarrow Q \quad \text{iff} \quad \neg P \vee Q.$$

Definition 4.7. The predicate $\mathrm{INST}(x)$ is defined primitive recursively as follows:

$$\begin{aligned}
&[1 \leq \mathrm{Typ}(x) \leq 5] \wedge [1 \leq \mathrm{Reg}(x)] \wedge \\
&[\mathrm{Typ}(x) \leq 3 \Rightarrow \mathrm{Jmp}(x) = 0] \wedge \\
&[\mathrm{Typ}(x) = 3 \Rightarrow \mathrm{Reg}(x) = 1].
\end{aligned}$$

The predicate $\mathrm{INST}(x)$ says that if x is the code of an instruction, say $x = \langle c, i, j, k\rangle$, then $1 \leq c \leq 5$, $j \geq 1$, if $c \leq 3$, then $k = 0$, and if $c = 3$ then we also have $j = 1$.

Definition 4.8. Program are coded as follows. If P is a RAM program composed of the n instructions $I_1, \ldots, I_n$, the code of P, denoted as $\#P$, is

$$\#P = \langle n, \#I_1, \ldots, \#I_n\rangle.$$

Recall from Property $(*)$ in Section 4.1 that

$$\langle n, \#I_1, \ldots, \#I_n\rangle = \langle n, \langle \#I_1, \ldots, \#I_n\rangle\rangle.$$

Also recall that

$$\langle x, y\rangle = ((x + y)^2 + 3x + y)/2.$$

Example 4.1. Consider the following program Padd2 computing the function add2: $\mathbb{N} \to \mathbb{N}$ given by

$$\text{add2}(n) = n + 2.$$

Padd2:

$$\begin{array}{llll} I_1: & 1 & \texttt{add} & R1 \\ I_2: & 2 & \texttt{add} & R1 \\ I_3: & 3 & \texttt{continue} & \end{array}$$

We have

$$\begin{aligned} \#I1 &= \langle 1, 1, 1, 0 \rangle_4 = \langle 1, \langle 1, \langle 1, 0 \rangle \rangle \rangle = 37 \\ \#I2 &= \langle 1, 2, 1, 0 \rangle_4 = \langle 1, \langle 2, \langle 1, 0 \rangle \rangle \rangle = 92 \\ \#I3 &= \langle 3, 3, 1, 0 \rangle_4 = \langle 3, \langle 3, \langle 1, 0 \rangle \rangle \rangle = 234 \end{aligned}$$

and

$$\begin{aligned} \#\text{Padd2} &= \langle 3, \#I1, \#I2, \#I3 \rangle_4 = \langle 3, \langle 37, \langle 92, 234 \rangle \rangle \\ &= 1\,018\,748\,519\,973\,070\,618. \end{aligned}$$

The codes get big fast!

We define the primitive recursive functions Ln, Pg, and Line, such that:

$$\begin{aligned} \text{Ln}(x) &= \Pi(1, 2, x), \\ \text{Pg}(x) &= \Pi(2, 2, x), \\ \text{Line}(i, x) &= \Pi(i, \text{Ln}(x), \text{Pg}(x)). \end{aligned}$$

The function Ln yields the length of the program (the number of instructions), Pg yields the sequence of instructions in the program (really, a code for the sequence), and $\text{Line}(i, x)$ yields the code of the ith instruction in the program. Again, if x does not code a program, there is no need to interpret these functions. However, note that by a previous exercise, it happens that

$$\begin{aligned} \text{Line}(0, x) &= \text{Line}(1, x), \quad \text{and} \\ \text{Line}(\text{Ln}(x), x) &= \text{Line}(i, x), \quad \text{for all } i \geq \text{Ln}(x). \end{aligned}$$

The primitive recursive predicate PROG is defined such that $\text{PROG}(x)$ holds iff x codes a program. Thus, $\text{PROG}(x)$ holds if each line codes an instruction, each jump has an instruction to jump to, and the last instruction is a `continue`.

Definition 4.9. The primitive recursive predicate $\mathrm{PROG}(x)$ is given by

$$\begin{aligned}
&\forall i \leq \mathrm{Ln}(x)[i \geq 1 \Rightarrow\\
&[\mathrm{INST}(\mathrm{Line}(i,x)) \wedge \mathrm{Typ}(\mathrm{Line}(\mathrm{Ln}(x),x)) = 3\\
&\wedge [\mathrm{Typ}(\mathrm{Line}(i,x)) = 4 \Rightarrow\\
&\exists j \leq i-1[j \geq 1 \wedge \mathrm{LNum}(\mathrm{Line}(j,x)) = \mathrm{Jmp}(\mathrm{Line}(i,x))]]\wedge\\
&[\mathrm{Typ}(\mathrm{Line}(i,x)) = 5 \Rightarrow\\
&\exists j \leq \mathrm{Ln}(x)[j > i \wedge \mathrm{LNum}(\mathrm{Line}(j,x)) = \mathrm{Jmp}(\mathrm{Line}(i,x))]]]].
\end{aligned}$$

Note that we have used Proposition 1.7 which states that if f is a primitive recursive function and if P is a primitive recursive predicate, then $\exists x \leq f(y)P(x)$ is primitive recursive.

The last instruction $\mathrm{Line}(\mathrm{Ln}(x),x))$ in the program must be a `continue`, which means that $\mathrm{Typ}(\mathrm{Line}(\mathrm{Ln}(x),x)) = 3$. When the ith instruction coded by $\mathrm{Line}(i,x)$ of the program coded by x has its first field $\mathrm{Typ}(\mathrm{Line}(i,x)) = 4$, this instruction is a jump above, and there must be an instruction in line j above instruction in line i, which means that $1 \leq j \leq i-1$, and the line number $\mathrm{LNum}(\mathrm{Line}(j,x))$ of the jth instruction must be equal to the jump address $\mathrm{Jmp}(\mathrm{Line}(i,x))$ of the ith instruction. When $\mathrm{Typ}(\mathrm{Line}(i,x)) = 5$, this instruction is a jump below, and the analysis is similar.

We are now ready to prove a fundamental result in the theory of algorithms. This result points out some of the limitations of the notion of algorithm.

Theorem 4.4. *(Undecidability of the halting problem) There is no RAM program* **Decider** *which halts for all inputs and has the following property when started with input x in register $R1$ and with input i in register $R2$ (the other registers being set to zero):*

(1) **Decider** *halts with output* 1 *iff i codes a program that eventually halts when started on input x (all other registers set to zero).*

(2) **Decider** *halts with output* 0 *in $R1$ iff i codes a program that runs forever when started on input x in $R1$ (all other registers set to zero).*

(3) If i does not code a program, then **Decider** *halts with output* 2 *in $R1$.*

Proof. Assume that **Decider** is such a RAM program, and let Q be the following program with a single input:

$$\text{Program } Q \text{ (code } q) \left\{ \begin{array}{llll} & R2 & \leftarrow & R1 \\ & & P & \\ N1 & & \texttt{continue} & \\ & R1 & \texttt{jmp} & N1a \\ & & \texttt{continue} & \end{array} \right.$$

Let i be the code of some program P. The key point is that *the termination behavior of Q on input i is exactly the opposite of the termination behavior of* **Decider** *on input i and code i*.

(1) If **Decider** says that program P coded by i *halts* on input i, then $R1$ just after the `continue` in line $N1$ contains 1, and Q *loops forever*.

(2) If **Decider** says that program P coded by i *loops forever* on input i, then $R1$ just after `continue` in line $N1$ contains 0, and Q *halts*.

The program Q can be translated into a program using only instructions of type 1, 2, 3, 4, 5, described previously, and let q *be the code of the program* Q.

Let us see what happens if we run the program Q *on input* q *in* $R1$ *(all other registers set to zero).*

Just after execution of the assignment $R2 \leftarrow R1$, the program **Decider** is started with q *in both* $R1$ and $R2$. Since **Decider** is supposed to halt for all inputs, it eventually halts with output 0 or 1 in $R1$. If **Decider** halts with output 1 in $R1$ (which means that Q halts on input q), then Q goes into an infinite loop, while if **Decider** halts with output 0 in $R1$ (which means that Q loops forever on input q), then Q halts. But then, we see that **Decider** says that Q halts when started on input q iff Q loops forever on input q, a contradiction. Therefore, **Decider** cannot exist. □

The argument used in the proof of 4.4 is quite similar in spirit to "Russell's Paradox." If we identify the notion of algorithm with that of a RAM program which halts for all inputs, the above theorem says that there is *no* algorithm for deciding whether a RAM program eventually halts for a given input. We say that the halting problem for RAM programs is *undecidable* (or *unsolvable*).

The above theorem also implies that the halting problem for Turing machines is undecidable. Indeed, if we had an algorithm for solving the halting problem for Turing machines, we could solve the halting problem for RAM programs as follows: first, apply the algorithm for translating a RAM program into an equivalent Turing machine, and then apply the algorithm solving the halting problem for Turing machines.

The argument is typical in computability theory and is called a "reducibility argument."

Our next goal is to define a primitive recursive function that describes the computation of RAM programs.

4.4 Universal RAM Programs

To describe the computation of a RAM program, we need to code not only RAM programs but also the contents of the registers. Assume that we have a RAM program P using n registers $R1, \ldots, Rn$, whose contents are denoted as $r_1, \ldots, r_n$. We can code $r_1, \ldots, r_n$ into a single integer $\langle r_1, \ldots, r_n \rangle$. Conversely, every integer x can be viewed as coding the contents of $R1, \ldots, Rn$, by taking the sequence $\Pi(1, n, x), \ldots, \Pi(n, n, x)$.

Actually, it is not necessary to know n, the number of registers, if we make the following observation:

$$\mathrm{Reg}(\mathrm{Line}(i,x)) \leq \mathrm{Line}(i,x) \leq \mathrm{Pg}(x) < x$$

for all $i, x \in \mathbb{N}$. If x codes a program, then $R1, \ldots, Rx$ certainly include all the registers in the program. Also note that from a previous exercise,

$$\langle r_1, \ldots, r_n, 0, \ldots, 0\rangle = \langle r_1, \ldots, r_n, 0\rangle.$$

We now define the primitive recursive functions Nextline, Nextcont, and Comp, describing the computation of RAM programs. There are a lot of tedious technical details that the reader should skip upon first reading. However, to be rigorous, we must spell out all these details.

Definition 4.10. Let x code a program and let i be such that $1 \leq i \leq \mathrm{Ln}(x)$. The following functions are defined:

(1) $\mathrm{Nextline}(i,x,y)$ is the number of the next instruction to be executed after executing the ith instruction (the current instruction) in the program coded by x, where the contents of the registers is coded by y.

(2) $\mathrm{Nextcont}(i,x,y)$ is the code of the contents of the registers after executing the ith instruction in the program coded by x, where the contents of the registers is coded by y.

(3) $\mathrm{Comp}(x,y,m) = \langle i, z\rangle$, where i and z are defined such that after running the program coded by x for m steps, where the initial contents of the program registers are coded by y, the next instruction to be executed is the ith one, and z is the code of the current contents of the registers.

Proposition 4.5. *The functions* Nextline*,* Nextcont*, and* Comp *are primitive recursive.*

Proof. (1) $\mathrm{Nextline}(i,x,y) = i+1$, unless the ith instruction is a jump and the contents of the register being tested is nonzero:

$$\begin{aligned}
&\mathrm{Nextline}(i,x,y) = \\
&\qquad\qquad \max j \leq \mathrm{Ln}(x)[j < i \wedge \mathrm{LNum}(\mathrm{Line}(j,x)) = \mathrm{Jmp}(\mathrm{Line}(i,x))] \\
&\qquad\qquad \text{if } \mathrm{Typ}(\mathrm{Line}(i,x)) = 4 \wedge \Pi(\mathrm{Reg}(\mathrm{Line}(i,x)),x,y) \neq 0 \\
&\qquad\qquad \min j \leq \mathrm{Ln}(x)[j > i \wedge \mathrm{LNum}(\mathrm{Line}(j,x)) = \mathrm{Jmp}(\mathrm{Line}(i,x))] \\
&\qquad\qquad \text{if } \mathrm{Typ}(\mathrm{Line}(i,x)) = 5 \wedge \Pi(\mathrm{Reg}(\mathrm{Line}(i,x)),x,y) \neq 0 \\
&\qquad\qquad i+1 \text{ otherwise.}
\end{aligned}$$

For example, if the ith instruction of the program coded by x is a jump above, namely $\mathrm{Typ}(\mathrm{Line}(i,x)) = 4$, then the register being tested is $\mathrm{Reg}(\mathrm{Line}(i,x))$, and its contents must be nonzero for a jump to occur, so the contents of this register, which is obtained from the code y of all registers as $\Pi(\mathrm{Reg}(\mathrm{Line}(i,x)),x,y)$ (remember that we may assume that there are x registers, by padding with zeros) must be nonzero.

Note that according to this definition, if the ith line is the final `continue`, then Nextline signals that the program has halted by yielding

$$\mathrm{Nextline}(i, x, y) > \mathrm{Ln}(x).$$

(2) We need two auxiliary functions Add and Sub defined as follows.

$\mathrm{Add}(j, x, y)$ is the number coding the contents of the registers used by the program coded by x after register Rj coded by $\Pi(j, x, y)$ has been increased by 1, and

$\mathrm{Sub}(j, x, y)$ codes the contents of the registers after register Rj has been decremented by 1 (y codes the previous contents of the registers). It is easy to see that

$$\begin{aligned}\mathrm{Sub}(j, x, y) = \min z \leq y[\Pi(j, x, z) = \Pi(j, x, y) - 1 \\ \wedge \forall k \leq x[0 < k \neq j \Rightarrow \Pi(k, x, z) = \Pi(k, x, y)]].\end{aligned}$$

The definition of Add is slightly more tricky. We leave as an exercise to the reader to prove that:

$$\begin{aligned}&\mathrm{Add}(j, x, y) = \min z \leq \mathrm{Large}(x, y + 1) \\ &\qquad [\Pi(j, x, z) = \Pi(j, x, y) + 1 \wedge \forall k \leq x[0 < k \neq j \Rightarrow \Pi(k, x, z) = \Pi(k, x, y)]],\end{aligned}$$

where the function Large is the function defined in an earlier exercise. Then

$$\begin{aligned}&\mathrm{Nextcont}(i, x, y) = \\ &\qquad \mathrm{Add}(\mathrm{Reg}(\mathrm{Line}(i, x), x, y) \quad \text{if} \quad \mathrm{Typ}(\mathrm{Line}(i, x)) = 1 \\ &\qquad \mathrm{Sub}(\mathrm{Reg}(\mathrm{Line}(i, x), x, y) \quad \text{if} \quad \mathrm{Typ}(\mathrm{Line}(i, x)) = 2 \\ &\qquad y \quad \text{if} \quad \mathrm{Typ}(\mathrm{Line}(i, x)) \geq 3.\end{aligned}$$

(3) Recall that $\Pi_1(z) = \Pi(1, 2, z)$ and $\Pi_2(z) = \Pi(2, 2, z)$. The function Comp is defined by primitive recursion as follows:

$$\begin{aligned}\mathrm{Comp}(x, y, 0) &= \langle 1, y\rangle \\ \mathrm{Comp}(x, y, m + 1) &= \langle \mathrm{Nextline}(\Pi_1(\mathrm{Comp}(x, y, m)), x, \Pi_2(\mathrm{Comp}(x, y, m))), \\ &\qquad \mathrm{Nextcont}(\Pi_1(\mathrm{Comp}(x, y, m)), x, \Pi_2(\mathrm{Comp}(x, y, m)))\rangle.\end{aligned}$$

If $\mathrm{Comp}(x, y, m) = \langle i, z\rangle$, then $\Pi_1(\mathrm{Comp}(x, y, m)) = i$ is the number of the next instruction to be executed and $\Pi_2(\mathrm{Comp}(x, y, m)) = z$ codes the current contents of the registers, so

$$\mathrm{Comp}(x, y, m + 1) = \langle \mathrm{Nextline}(i, x, z), \mathrm{Nextcont}(i, x, z)\rangle,$$

as desired. □

We can now reprove that every RAM computable function is partial computable. Indeed, assume that x codes a program P.

We would like to define the partial function End so that for all x, y, where x codes a program and y codes the contents of its registers, $\mathrm{End}(x, y)$ is the number of steps for which the computation runs before halting, if it halts. If the program does not halt, then $\mathrm{End}(x, y)$ is undefined.

If y is the value of the register $R1$ before the program P coded by x is started, recall that the contents of the registers is coded by $\langle y, 0\rangle$. Noticing that 0 and 1 do not code programs, we note that if x codes a program, then $x \geq 2$, and $\Pi_1(z) = \Pi(1, x, z)$ is the contents of $R1$ as coded by z.

Since $\mathrm{Comp}(x, y, m) = \langle i, z\rangle$, we have

$$\Pi_1(\mathrm{Comp}(x, y, m)) = i,$$

where i is the number (index) of the instruction reached after running the program P coded by x with initial values of the registers coded by y for m steps. Thus, P halts if i is the last instruction in P, namely $\mathrm{Ln}(x)$, iff

$$\Pi_1(\mathrm{Comp}(x, y, m)) = \mathrm{Ln}(x).$$

This suggests the following definition.

Definition 4.11. The partial function $\mathrm{End}(x, y)$ is defined by

$$\mathrm{End}(x, y) = \min m[\Pi_1(\mathrm{Comp}(x, y, m)) = \mathrm{Ln}(x)].$$

Note that End is a partial computable function; it can be computed by a RAM program involving *only one while loop* searching for the number of steps m. The function involved in the minimization is *primitive recursive*. However, in general, End is not a total function.

If φ is the partial computable function computed by the program P coded by x, then we claim that

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))).$$

This is because if $m = \mathrm{End}(x, \langle y, 0\rangle)$ is the number of steps after which the program P coded by x halts on input y, then

$$\mathrm{Comp}(x, \langle y, 0\rangle, m)) = \langle \mathrm{Ln}(x), z\rangle,$$

where z is the code of the register contents when the program stops. Consequently

$$\begin{aligned} z &= \Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, m)) \\ z &= \Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle))). \end{aligned}$$

The value of the register $R1$ is $\Pi_1(z)$, that is

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle)))).$$

The above fact is worth recording as the following proposition which is a variant of a result known as the *Kleene normal form*

Proposition 4.6. *(Kleene normal form for RAM programs) If φ is the partial computable function computed by the program P coded by x, then we have*

$$\varphi(y) = \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle))) \quad \textit{for all } y \in \mathbb{N}.$$

Observe that φ is written in the form $\varphi = g \circ \min f$, for some primitive recursive functions f and g. It will be convenient to denote the function φ computed by the RAM program P coded by x as φ_x. We also denote the program P coded by x as P_x.

We can also exhibit a partial computable function which enumerates all the unary partial computable functions. It is a *universal function*.

Abusing the notation slightly, we will write $\varphi(x, y)$ for $\varphi(\langle x, y\rangle)$, viewing φ as a function of two arguments (however, φ is really a function of a single argument). We define the function φ_{univ} as follows:

$$\varphi_{univ}(x, y) = \begin{cases} \Pi_1(\Pi_2(\mathrm{Comp}(x, \langle y, 0\rangle, \mathrm{End}(x, \langle y, 0\rangle))) & \text{if PROG}(x), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function φ_{univ} is a partial computable function with the following property: for every x coding a RAM program P, for every input y,

$$\varphi_{univ}(x, y) = \varphi_x(y),$$

the value of the partial computable function φ_x computed by the RAM program P coded by x. If x does not code a program, then $\varphi_{univ}(x, y)$ is undefined for all y.

By Proposition 1.9, the partial function φ_{univ} is not computable (recursive).[2] Indeed, being an enumerating function for the partial computable functions, it is an enumerating function for the total computable functions, and thus, it cannot be computable. Being a partial function saves us from a contradiction.

The existence of the universal function φ_{univ} is sufficiently important to be recorded in the following proposition.

Proposition 4.7. *(Universal RAM program) For the indexing of RAM programs defined earlier, there is a universal partial computable function φ_{univ} such that, for all $x, y \in \mathbb{N}$, if φ_x is the partial computable function computed by the program P_x coded by x, then*

$$\varphi_x(y) = \varphi_{univ}(\langle x, y\rangle).$$

The program UNIV computing φ_{univ} can be viewed as an *interpreter* for RAM programs. By giving the universal program UNIV the "program" x and the "data" y, we get the result of executing program P_x on input y. We can view the RAM model as a *stored program computer*.

[2] The term *recursive function* is now considered old-fashion. Many researchers have switched to the term *computable function*.

By Theorem 4.4 and Proposition 4.7, the halting problem for the single program UNIV is undecidable. Otherwise, the halting problem for RAM programs would be decidable, a contradiction. It should be noted that the program UNIV can actually be written (with a certain amount of pain).

The existence of the function φ_{univ} leads us to the notion of an *indexing* of the RAM programs.

4.5 Indexing of RAM Programs

We can define a listing of the RAM programs as follows. If x codes a program (that is, if PROG(x) holds) and P is the program that x codes, we call this program P the xth RAM program and denote it as P_x. If x does not code a program, we let P_x be the program that diverges for every input:

```
N1           add      R1
N1   R1      jmp      N1a
N1        continue
```

Therefore, in all cases, P_x stands for the xth RAM program. Thus, we have a listing of RAM programs, $P_0, P_1, P_2, P_3, \ldots$, such that every RAM program (of the restricted type considered here) appears in the list exactly once, except for the "infinite loop" program. For example, the program Padd2 (adding 2 to an integer) appears as

$$P_{1\,018\,748\,519\,973\,070\,618}.$$

In particular, note that φ_{univ} being a partial computable function, it is computed by some RAM program UNIV that has a code *univ* and is the program P_{univ} in the list.

Having an indexing of the RAM programs, we also have an indexing of the partial computable functions.

Definition 4.12. For every integer $x \geq 0$, we let P_x be the RAM program coded by x as defined earlier, and φ_x be the partial computable function computed by P_x.

For example, the function add2 (adding 2 to an integer) appears as

$$\varphi_{1\,018\,748\,519\,973\,070\,618}.$$

Remark: Kleene used the notation $\{x\}$ for the partial computable function coded by x. Due to the potential confusion with singleton sets, we follow Rogers, and use the notation φ_x; see Rogers [37], page 21.

It is important to observe that *different programs* P_x and P_y may compute the *same function*, that is, while $P_x \neq P_y$ for all $x \neq y$, it is possible that $\varphi_x = \varphi_y$. For example, the program P_y coded by y may be the program obtained from the program P_x coded by x obtained by adding and subtracting 1 a million times to a register not in the program P_x. In fact, it is *undecidable* whether $\varphi_x = \varphi_y$.

The object of the next section is to show the existence of Kleene's T-predicate. This will yield another important normal form. In addition, the T-predicate is a basic tool in recursion theory.

4.6 Kleene's T-Predicate

In Section 4.3, we have encoded programs. The idea of this section is to also encode *computations* of RAM programs. Assume that x codes a program, that y is some input (not a code), and that z codes a computation of P_x on input y.

Definition 4.13. The predicate $T(x, y, z)$ is defined as follows:

$T(x, y, z)$ holds iff x codes a RAM program, y is an input, and z codes a halting computation of P_x on input y.

The code z of a computation packs the consecutive "states" of the computation, namely the pairs $\langle i_j, y_j\rangle$, where i_j is the physical location of the next instruction to be executed and each y_j codes the contents of the registers just before execution of this instruction. We will show that T is primitive recursive.

First we need to *encode computations*. We say that z codes a computation of length $n \geq 1$ if

$$z = \langle n + 2, \langle 1, y_0\rangle, \langle i_1, y_1\rangle, \ldots, \langle i_n, y_n\rangle\rangle,$$

where each i_j is the physical location of the next instruction to be executed and each y_j codes the contents of the registers just before execution of the instruction at the location i_j. Also, y_0 codes the initial contents of the registers, that is, $y_0 = \langle y, 0\rangle$, for some input y.

We let $\mathrm{Lz}(z) = \Pi_1(z)$ (not to be confused with $\mathrm{Ln}(x)$).

Note that i_j denotes the physical location of the next instruction to be executed in the sequence of instructions constituting the program coded by x, and not the line number (label) of this instruction. Thus, the first instruction to be executed is in location 1, $1 \leq i_j \leq \mathrm{Ln}(x)$, and $i_{n-1} = \mathrm{Ln}(x)$. Since the last instruction which is executed is the last physical instruction in the program, namely, a `continue`, there is no next instruction to be executed after that, and i_n is irrelevant. Writing the definition of T is a little simpler if we let $i_n = \mathrm{Ln}(x) + 1$.

Definition 4.14. The T-predicate is the primitive recursive predicate defined as follows:

$T(x, y, z)$ iff $\mathrm{PROG}(x)$ and $(\mathrm{Lz}(z) \geq 3)$ and
$\forall j \leq \mathrm{Lz}(z) - 3[0 \leq j \Rightarrow$
$\mathrm{Nextline}(\Pi_1(\Pi(j + 2, \mathrm{Lz}(z), z)), x, \Pi_2(\Pi(j + 2, \mathrm{Lz}(z), z))) = \Pi_1(\Pi(j + 3, \mathrm{Lz}(z), z))$ and
$\mathrm{Nextcont}(\Pi_1(\Pi(j + 2, \mathrm{Lz}(z), z)), x, \Pi_2(\Pi(j + 2, \mathrm{Lz}(z), z))) = \Pi_2(\Pi(j + 3, \mathrm{Lz}(z), z))$ and
$\Pi_1(\Pi(\mathrm{Lz}(z) - 1, \mathrm{Lz}(z), z)) = \mathrm{Ln}(x)$ and
$\Pi_1(\Pi(2, \mathrm{Lz}(z), z)) = 1$ and
$y = \Pi_1(\Pi_2(\Pi(2, \mathrm{Lz}(z), z)))$ and $\Pi_2(\Pi_2(\Pi(2, \mathrm{Lz}(z), z))) = 0]$.

The reader can verify that $T(x, y, z)$ holds iff x codes a RAM program, y is an input, and z codes a halting computation of P_x on input y. For example, since

$$z = \langle n + 2, \langle 1, y_0\rangle, \langle i_1, y_1\rangle, \ldots, \langle i_n, y_n\rangle\rangle,$$

we have $\Pi(j + 2, \mathrm{Lz}(z), z) = \langle i_{j-1}, y_{j-1}\rangle$ and $\Pi(j + 3, \mathrm{Lz}(z), z) = \langle i_j, y_j\rangle$, so $\Pi_1(\Pi(j + 2, \mathrm{Lz}(z), z)) = \Pi_1(\langle i_{j-1}, y_{j-1}\rangle) = i_{j-1}$, $\Pi_2(\Pi(j + 2, \mathrm{Lz}(z), z)) = \Pi_2(\langle i_{j-1}, y_{j-1}\rangle) = y_{j-1}$, and similarly $\Pi_1(\Pi(j+3, \mathrm{Lz}(z), z)) = i_j$, $\Pi_2(\Pi(j+3, \mathrm{Lz}(z), z)) = y_j$, so the T predicate expresses that $\mathrm{Nextline}(i_{j-1}, y_{j-1}) = i_j$ and $\mathrm{Nextcont}(i_{j-1}, y_{j-1}) = y_j$.

In order to extract the output of P_x from z, we define the primitive recursive function Res as follows:

$$\mathrm{Res}(z) = \Pi_1(\Pi_2(\Pi(\mathrm{Lz}(z), \mathrm{Lz}(z), z))).$$

The explanation for this formula is that if $\Pi(\mathrm{Lz}(z), \mathrm{Lz}(z), z) = \langle i_n, y_n\rangle$, then $\Pi_2(\Pi(\mathrm{Lz}(z), \mathrm{Lz}(z), z)) = y_n$, the code of the registers, and since the output is returned in Register $R1$, $\mathrm{Res}(z)$ is the contents of register $R1$ when P_x halts, that is, $\Pi_1(y_{\mathrm{Lz}(z)})$. Using the T-predicate, we get the so-called Kleene normal form.

Theorem 4.8. *(Kleene Normal Form) Using the indexing of the partial computable functions defined earlier, we have*

$$\varphi_x(y) = \mathrm{Res}[\min z(T(x, y, z))],$$

where $T(x, y, z)$ and Res *are primitive recursive.*

Note that the universal function φ_{univ} can be defined as

$$\varphi_{univ}(x, y) = \mathrm{Res}[\min z(T(x, y, z))].$$

There is another important property of the partial computable functions, namely, that composition is effective (computable). We need two auxiliary primitive recursive functions. The function Conprogs creates the code of the program obtained by concatenating the programs P_x and P_y, and for $i \geq 2$, $\mathrm{Cumclr}(i)$ is the code of the program which clears registers $R2, \ldots, Ri$. To get Cumclr, we can use the function $\mathrm{clr}(i)$ such that $\mathrm{clr}(i)$ is the code of the program

```
N1         tail      Ri
N1   Ri    jmp       N1a
N          continue
```

We leave it as an exercise to prove that clr, Conprogs, and Cumclr, are primitive recursive.

Theorem 4.9. *There is a primitive recursive function c such that*

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

Proof. If both x and y code programs, then $\varphi_x \circ \varphi_y$ can be computed as follows: Run P_y, clear all registers but $R1$, then run P_x. Otherwise, let loop be the index of the infinite loop program:

$$c(x,y) = \begin{cases} \text{Conprogs}(y, \text{Conprogs}(\text{Cumclr}(y), x)) & \text{if PROG}(x) \text{ and PROG}(y) \\ \text{loop} & \text{otherwise.} \end{cases}$$

□

4.7 A Non-Computable Function; Busy Beavers

Total functions that *are not computable must grow very fast and thus are very complicated*. Yet, in 1962, Radó published a paper in which he defined two functions Σ and S (involving computations of Turing machines) that are total and not computable.

Consider Turing machines with a tape alphabet $\Gamma = \{1, B\}$ with two symbols (B being the blank). We also assume that these Turing machines have a special final state q_F, which is a blocking state (there are no transitions from q_F). We do not count this state when counting the number of states of such Turing machines. The game is to run such Turing machines with a fixed number of states n starting on a blank tape, with the goal of producing the maximum number of (not necessarily consecutive) ones (1).

Definition 4.15. The function Σ (defined on the positive natural numbers) is defined as the maximum number $\Sigma(n)$ of (not necessarily consecutive) 1's written on the tape after a Turing machine with $n \geq 1$ states started on the blank tape halts. The function S is defined as the maximum number $S(n)$ of moves that can be made by a Turing machine of the above type with n states before it halts, started on the blank tape.[3]

Definition 4.16. A Turing machine with n states that writes the maximum number $\Sigma(n)$ of 1's when started on the blank tape is called a *busy beaver*.

[3] The function S defined here is obviously not the successor function from Definition 1.16.

Busy beavers are hard to find, even for small n. First, it can be shown that the number of distinct Turing machines of the above kind with n states is $(4(n+1))^{2n}$. Second, since it is undecidable whether a Turing machine halts on a given input, it is hard to tell which machines loop or halt after a very long time.

Here is a summary of what is known for $1 \leq n \leq 6$. Observe that the exact value of $\Sigma(6)$ and $S(6)$ is unknown.

n	$\Sigma(n)$	$S(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	4098	$47,176,870$
6	$\geq 95,524,079$	$\geq 8,690,333,381,690,951$
6	$\geq 3.515 \times 10^{18267}$	$\geq 7.412 \times 10^{36534}$

The first entry in the table for $n = 6$ corresponds to a machine due to Heiner Marxen (1999). This record was surpassed by Pavel Kropitz in 2010, which corresponds to the second entry for $n = 6$. The machines achieving the record in 2017 for $n = 4, 5, 6$ are shown below, where the blank is denoted Δ instead of B, and where the special halting state is denoted H:

4-state busy beaver:

	A	B	C	D
Δ	$(1, R, B)$	$(1, L, A)$	$(1, R, H)$	$(1, R, D)$
1	$(1, L, B)$	(Δ, L, C)	$(1, L, D)$	(Δ, R, A)

The above machine output 13 ones in 107 steps. In fact, the output is

$$1\,\Delta\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,.$$

5-state busy beaver:

	A	B	C	D	E
Δ	$(1, R, B)$	$(1, R, C)$	$(1, R, D)$	$(1, L, A)$	$(1, R, H)$
1	$(1, L, C)$	$(1, R, B)$	(Δ, L, E)	$(1, L, D)$	(Δ, L, A)

The above machine output 4098 ones in $47,176,870$ steps. The tape actually contains a total of 12289 symbols, 4098 if which are 1's, and the other the blank Δ.

The fact that this machine is a busy beaver was established in 2024 with the help of the Coq proof assistant. The proof is an enumeration of roughly 180 million machines, which are checked for termination with specialized deciders that recognize specific behaviors. This is quite a "tour de force."

6-state contender (Heiner Marxen):

	A	B	C	D	E	F
Δ	$(1, R, B)$	$(1, L, C)$	(Δ, R, F)	$(1, R, A)$	$(1, L, H)$	(Δ, L, A)
1	$(1, R, A)$	$(1, L, B)$	$(1, L, D)$	(Δ, L, E)	$(1, L, F)$	(Δ, L, C)

The above machine outputs $96,524,079$ ones in $8,690,333,381,690,951$ steps.

6-state best contender (Pavel Kropitz):

	A	B	C	D	E	F
Δ	$(1, R, B)$	$(1, R, C)$	$(1, L, D)$	$(1, R, E)$	$(1, L, A)$	$(1, L, H)$
1	$(1, L, E)$	$(1, R, F)$	(Δ, R, B)	(Δ, L, C)	(Δ, R, D)	$(1, R, C)$

The above machine output at least 3.515×10^{18267} ones!

The reason why it is so hard to compute Σ and S is that they are not computable!

Theorem 4.10. *The functions Σ and S are total functions that are not computable (not recursive).*

Proof sketch. The proof consists in showing that Σ (and similarly for S) eventually outgrows any computable function. More specifically, we claim that for every computable function f, there is some positive integer k_f such that

$$\Sigma(n + k_f) \geq f(n) \quad \text{for all } n \geq 0.$$

We simply have to pick k_f to be the number of states of a Turing machine M_f computing f. Then we can create a Turing machine $M_{n,f}$ that works as follows. Using n of its states, it writes n ones on the tape, and then it simulates M_f with input 1^n. Since the ouput of $M_{n,f}$ started on the blank tape consists of $f(n)$ ones, and since $\Sigma(n + k_f)$ is the maximum number of ones that a turing machine with $n + k_f$ states will ouput when it stops, we must have

$$\Sigma(n + k_f) \geq f(n) \quad \text{for all } n \geq 0.$$

Next observe that $\Sigma(n) < \Sigma(n + 1)$, because we can create a Turing machine with $n + 1$ states which simulates a busy beaver machine with n states, and then writes an extra 1 when the busy beaver stops, by making a transition to the $(n + 1)$th state. It follows immediately that if $m < n$ then $\Sigma(m) < \Sigma(n)$. If Σ was computable, then so would be the function g given by $g(n) = \Sigma(2n)$. By the above, we would have

$$\Sigma(n + k_g) \geq g(n) = \Sigma(2n) \quad \text{for all } n \geq 0,$$

and for $n > k_g$, since $2n > n + k_g$, we would have $\Sigma(n + n_g) < \Sigma(2n)$, contradicting the fact that $\Sigma(n + n_g) \geq \Sigma(2n)$.

Since by definition $S(n)$ is the maximum number of moves that can be made by a Turing machine of the above type with n states before it halts, $S(n) \geq \Sigma(n)$. Then the same reasoning as above shows that S is not a computable function. □

The zoo of computable and non-computable functions is illustrated in Figure 4.1.

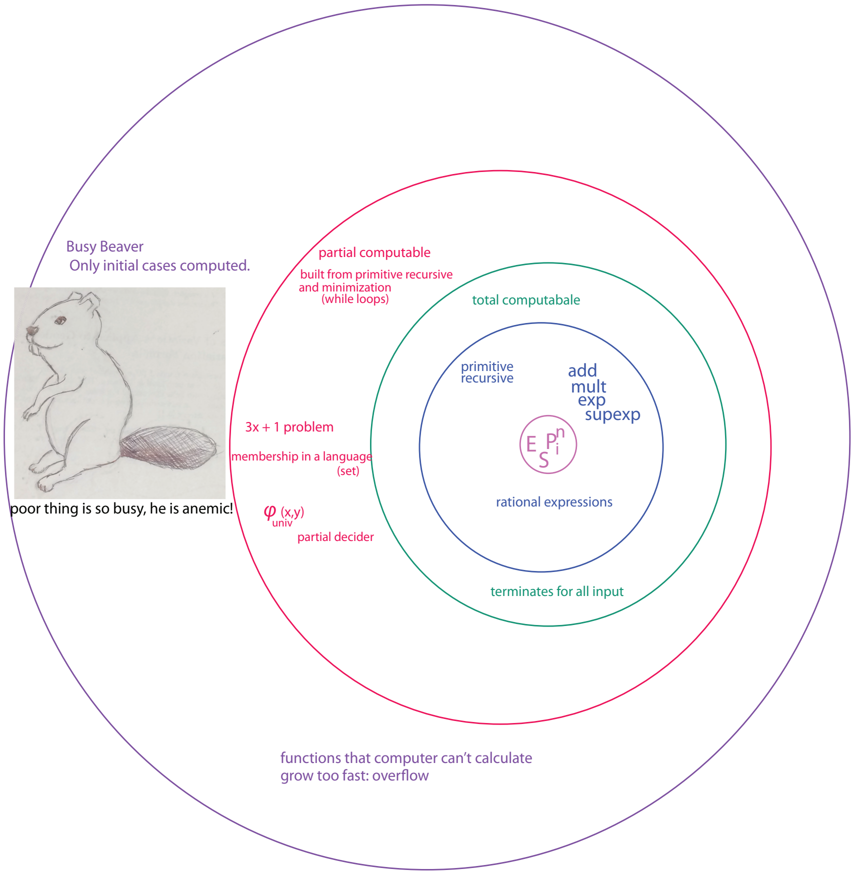


Figure 4.1: Computability Classification of Functions.

Chapter 5

Elementary Recursive Function Theory

5.1 Acceptable Indexings

In Chapter 4, we have exhibited a specific indexing of the partial computable functions by encoding the RAM programs. Using this indexing, we showed the existence of a universal function φ_{univ} and of a computable function c, with the property that for all $x, y \in \mathbb{N}$,

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

It is natural to wonder whether the same results hold if a different coding scheme is used or if a different model of computation is used, for example, Turing machines. In other words, we would like to know if our results depend on a specific coding scheme or not.

Our previous results showing the characterization of the partial computable functions being independent of the specific model used, suggests that it might be possible to pinpoint certain properties of coding schemes which would allow an axiomatic development of recursive function theory. What we are aiming at is to find some simple properties of "nice" coding schemes that allow one to proceed without using explicit coding schemes, as long as the above properties hold.

Remarkably, such properties exist. Furthermore, any two coding schemes having these properties are equivalent in a strong sense (called effectively equivalent), and so, one can pick any such coding scheme without any risk of losing anything else because the wrong coding scheme was chosen. Such coding schemes, also called indexings, or Gödel numberings, or even programming systems, are called *acceptable indexings*.

Definition 5.1. An *indexing* of the partial computable functions is an infinite sequence $\varphi_0, \varphi_1, \ldots,$ of partial computable functions that includes *all* the partial computable functions of one argument (there might be repetitions, this is why we are not using the term enumeration). An indexing is *universal* if it contains the partial computable function φ_{univ}

such that

$$\varphi_{univ}(i,x) = \varphi_i(x) \quad \text{for all } i, x \in \mathbb{N}. \tag{$*_{\text{univ}}$}$$

An indexing is *acceptable* if it is universal and if there is a total computable function c for composition, such that

$$\varphi_{c(i,j)} = \varphi_i \circ \varphi_j \quad \text{for all } i, j \in \mathbb{N}. \tag{$*_{\text{compos}}$}$$

An indexing may fail to be universal because it is not "computable enough," in the sense that it does not yield a function φ_{univ} satisfying ($*_{\text{univ}}$). It may also fail to be acceptable because it is not "computable enough," in the sense that it does not yield a function φ_{univ} satisfying ($*_{\text{compos}}$).

From Chapter 4, we know that the specific indexing of the partial computable functions given for RAM programs is acceptable. Another characterization of acceptable indexings left as an exercise is the following: an indexing $\psi_0, \psi_1, \psi_2, \ldots$ of the partial computable functions is acceptable iff there exists a total computable function f translating the RAM indexing of Section 4.3 into the indexing $\psi_0, \psi_1, \psi_2, \ldots$, that is,

$$\varphi_i = \psi_{f(i)} \quad \text{for all } i \in \mathbb{N}.$$

A very useful property of acceptable indexings is the so-called "s-m-n Theorem". Using the slightly loose notation $\varphi(x_1, \ldots, x_n)$ for $\varphi(\langle x_1, \ldots, x_n\rangle)$, the s-m-n Theorem says the following. Given a function φ considered as having $m + n$ arguments, if we fix the values of the first m arguments and we let the other n arguments vary, we obtain a function ψ of n arguments. Then the index of ψ depends in a computable fashion upon the index of φ and the first m arguments $x_1, \ldots, x_m$. We can "pull" the first m arguments of φ into the index of ψ.

Theorem 5.1. *(The "s-m-n Theorem") For any acceptable indexing $\varphi_0, \varphi_1, \ldots,$ there is a total computable function $s\colon \mathbb{N}^{n+2} \to \mathbb{N}$, such that, for all $i, m, n \geq 1$, for all $x_1, \ldots, x_m$ and all $y_1, \ldots, y_n$, we have*

$$\varphi_{s(i,m,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \varphi_i(x_1, \ldots, x_m, y_1, \ldots, y_n).$$

Proof. First, note that the above identity is really

$$\varphi_{s(i,m,\langle x_1,\ldots,x_m\rangle)}(\langle y_1, \ldots, y_n\rangle) = \varphi_i(\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle).$$

Recall that there is a primitive recursive function Con such that

$$\mathrm{Con}(m, \langle x_1, \ldots, x_m\rangle, \langle y_1, \ldots, y_n\rangle) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle$$

for all $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{N}$. Hence, a computable function s such that

$$\varphi_{s(i,m,x)}(y) = \varphi_i(\mathrm{Con}(m, x, y))$$

will do. We define some auxiliary primitive recursive functions as follows:

$$P(y) = \langle 0, y\rangle \quad \text{and} \quad Q(\langle x, y\rangle) = \langle x+1, y\rangle.$$

Since we have an indexing of the partial computable functions, there are indices p and q such that $P = \varphi_p$ and $Q = \varphi_q$. Let R be defined such that

$$\begin{aligned} R(0) &= p,\\ R(x+1) &= c(q, R(x)), \end{aligned}$$

where c is the computable function for composition given by the indexing. We prove by induction of x that

$$\varphi_{R(x)}(y) = \langle x, y\rangle \quad \text{for all } x, y \in \mathbb{N}.$$

For this we use the existence of the universal function φ_{univ}.

For the base case $x = 0$, we have

$$\begin{aligned} \varphi_{R(0)}(y) &= \varphi_{univ}(\langle R(0), y\rangle)\\ &= \varphi_{univ}(\langle p, y\rangle)\\ &= \varphi_p(y) = P(y) = \langle 0, y\rangle. \end{aligned}$$

For the induction step, we have

$$\begin{aligned} \varphi_{R(x+1)}(y) &= \varphi_{univ}(\langle R(x+1), y\rangle)\\ &= \varphi_{univ}(\langle c(q, R(x)), y\rangle)\\ &= \varphi_{c(q,R(x))}(y)\\ &= (\varphi_q \circ \varphi_{R(x)})(y)\\ &= \varphi_q(\langle x, y\rangle) = Q(\langle x, y\rangle) = \langle x+1, y\rangle. \end{aligned}$$

Also, recall that $\langle x, y, z\rangle = \langle x, \langle y, z\rangle\rangle$, by definition of pairing. Then we have

$$\varphi_{R(x)} \circ \varphi_{R(y)}(z) = \varphi_{R(x)}(\langle y, z\rangle) = \langle x, y, z\rangle.$$

Finally, let k be an index for the function Con, that is, let

$$\varphi_k(\langle m, x, y\rangle) = \mathrm{Con}(m, x, y).$$

Define s by

$$s(i, m, x) = c(i, c(k, c(R(m), R(x)))).$$

Then we have

$$\varphi_{s(i,m,x)}(y) = \varphi_i \circ \varphi_k \circ \varphi_{R(m)} \circ \varphi_{R(x)}(y) = \varphi_i(\mathrm{Con}(m, x, y)),$$

as desired. Notice that if the composition function c is primitive recursive, then s is also primitive recursive. In particular, for the specific indexing of the RAM programs given in Section 4.3, the function s is primitive recursive. □

In practice, when using the s-m-n Theorem we usually denote the function $s(i, m, x)$ simply as $s(x)$.

As a first application of the s-m-n Theorem, we show that any two acceptable indexings are effectively inter-translatable, that is, computably inter-translatable.

Theorem 5.2. *Let $\varphi_0, \varphi_1, \ldots,$ be a universal indexing, and let $\psi_0, \psi_1, \ldots,$ be any indexing with a total computable s-1-1 function, that is, a function s such that*

$$\psi_{s(i,1,x)}(y) = \psi_i(x, y)$$

for all $i, x, y \in \mathbb{N}$. Then there is a total computable function t such that $\varphi_i = \psi_{t(i)}$.

Proof. Let φ_{univ} be a universal partial computable function for the indexing $\varphi_0, \varphi_1, \ldots$. Since $\psi_0, \psi_1, \ldots,$ is also an indexing φ_{univ} occurs somewhere in the second list, and thus, there is some k such that $\varphi_{univ} = \psi_k$. Then we have

$$\psi_{s(k,1,i)}(x) = \psi_k(i, x) = \varphi_{univ}(i, x) = \varphi_i(x),$$

for all $i, x \in \mathbb{N}$. Therefore, we can take the function t to be the function defined such that

$$t(i) = s(k, 1, i)$$

for all $i \in \mathbb{N}$. □

Using Theorem 5.2, if we have two acceptable indexings $\varphi_0, \varphi_1, \ldots,$ and $\psi_0, \psi_1, \ldots$, there exist total computable functions t and u such that

$$\varphi_i = \psi_{t(i)} \quad \text{and} \quad \psi_i = \varphi_{u(i)}$$

for all $i \in \mathbb{N}$.

Also note that if the composition function c is primitive recursive, then any s-m-n function is primitive recursive, and the translation functions are primitive recursive. Actually, a stronger result can be shown. It can be shown that for any two acceptable indexings, there exist total computable *injective* and *surjective* translation functions. In other words, any two acceptable indexings are recursively isomorphic (Roger's isomorphism theorem); see Machtey and Young [29]. Next we turn to algorithmically unsolvable, or *undecidable*, problems.

5.2 Undecidable Problems

We saw in Section 4.3 that the halting problem for RAM programs is undecidable. In this section, we take a slightly more general approach to study the undecidability of problems, and give some tools for resolving decidability questions.

First, we prove again the undecidability of the halting problem, but this time, for *any* indexing of the partial computable functions.

Theorem 5.3. *(Halting Problem, Abstract Version) Let $\psi_0, \psi_1, \ldots,$ be any indexing of the partial computable functions. Then the function f defined such that*

$$f(x, y) = \begin{cases} 1 & \textit{if } \psi_x(y) \textit{ is defined,} \\ 0 & \textit{if } \psi_x(y) \textit{ is undefined,} \end{cases}$$

is not computable.

Proof. Assume that f is computable, and let g be the function defined such that

$$g(x) = f(x, x)$$

for all $x \in \mathbb{N}$. Then g is also computable. Let θ be the function defined such that

$$\theta(x) = \begin{cases} 0 & \text{if } g(x) = 0, \\ \text{undefined} & \text{if } g(x) = 1. \end{cases}$$

We claim that θ is not even partial computable. Observe that θ is such that

$$\theta(x) = \begin{cases} 0 & \text{if } \psi_x(x) \text{ is undefined,} \\ \text{undefined} & \text{if } \psi_x(x) \text{ is defined.} \end{cases}$$

If θ was partial computable, it would occur in the list as some ψ_i, and we would have

$$\theta(i) = \psi_i(i) = 0 \quad \text{iff} \quad \psi_i(i) \text{ is undefined,}$$

a contradiction. Therefore, f and g can't be computable. □

Observe that the proof of Theorem 5.3 *does not* use the fact that the indexing is universal or acceptable, and thus, the theorem holds for *any indexing* of the partial computable functions.

Given any set, X, for any subset, $A \subseteq X$, of X, recall that the *characteristic function, C_A (or χ_A), of A* is the function, $C_A \colon X \to \{0, 1\}$, defined so that, for all $x \in X$,

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

The function g defined in the proof of Theorem 5.3 is the characteristic function of an important set denoted as K.

Definition 5.2. Given any indexing (ψ_i) of the partial computable functions, the set K is defined by

$$K = \{x \mid \psi_x(x) \text{ is defined}\}.$$

The set K is an abstract version of the halting problem. It is example of a set which is *not* computable (or not recursive). Since this fact is quite important, we give the following definition:

Definition 5.3. A subset A of Σ^* (or a subset A of $\mathbb{N}$) is *computable*, or *recursive*,[1] or *decidable* iff its characteristic function, C_A, is a total computable function.

Using Definition 5.3, Theorem 5.3 can be restated as follows.

Proposition 5.4. *For any indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions (over Σ^* or $\mathbb{N}$), the set $K = \{x \mid \varphi_x(x) \text{ is defined}\}$ is not computable (not recursive).*

Computable (recursive) sets allow us to define the concept of a decidable (or undecidable) problem. The idea is to generalize the situation described in Section 4.3 and Section 4.6, where a set of objects, the RAM programs, is encoded into a set of natural numbers, using a coding scheme. For example, we would like to discuss the notion of computability of sets of trees or sets of graphs.

Definition 5.4. Let C be a countable set of objects, and let P be a property of objects in C. We view P as the set

$$\{a \in C \mid P(a)\}.$$

A *coding-scheme* is an injective function $\#\colon C \to \mathbb{N}$ that assigns a unique code to each object in C. The property P is *decidable (relative to #)* iff the set $\{\#(a) \mid a \in C \text{ and } P(a)\}$ is computable (recursive). The property P is *undecidable (relative to #)* iff the set $\{\#(a) \mid a \in C \text{ and } P(a)\}$ is not computable (not recursive).

Observe that the decidability of a property P of objects in C depends upon the coding scheme #. Thus, if we are cheating in using a non-effective (*i.e* not computable by a computer program) coding scheme, we may declare that a property is decidable even though it is not decidable in some reasonable coding scheme. Consequently, we require a coding scheme # to be *effective* in the following sense. Given any object $a \in C$, we can effectively (*i.e.* algorithmically) determine its code $\#(a)$. Conversely, given any integer $n \in \mathbb{N}$, we should be able to tell effectively if n is the code of some object in C, and if so, to find this object. In practice, it is always possible to describe the objects in C as strings over some (possibly complex) alphabet Σ (sets of trees, graphs, etc). In such cases, the coding schemes are computable functions from Σ^* to $\mathbb{N} = \{a_1\}^*$.

For example, let $C = \mathbb{N} \times \mathbb{N}$, where the property P is the equality of the partial functions φ_x and φ_y. We can use the pairing function $\langle -, - \rangle$ as a coding function, and the problem is formally encoded as the computability (recursiveness) of the set

$$\{\langle x, y \rangle \mid x, y \in \mathbb{N},\, \varphi_x = \varphi_y\}.$$

In most cases, we don't even bother to describe the coding scheme explicitly, knowing that such a description is routine, although perhaps tedious.

We now show that most properties about programs (except the trivial ones) are undecidable.

[1] Since 1996, the term *recursive* has been considered old-fashioned by many researchers, and the term *computable* has been used instead.

5.3 Reducibility and Rice's Theorem

First, we show that it is undecidable whether a RAM program halts for every input. In other words, it is undecidable whether a procedure is an algorithm. We actually prove a more general fact.

Proposition 5.5. *For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, the set*

$$\mathrm{TOTAL} = \{x \mid \varphi_x \textit{ is a total function}\}$$

is not computable (not recursive).

Proof. The proof uses a technique known as reducibility. We try to reduce a set A known to be *noncomputable* (*nonrecursive*) to TOTAL via a computable function $f\colon A \to \mathrm{TOTAL}$, so that

$$x \in A \quad \text{iff} \quad f(x) \in \mathrm{TOTAL}.$$

If TOTAL were computable (recursive), its characteristic function g would be computable, and thus, the function $g \circ f$ would be computable, a contradiction, since A is assumed to be noncomputable (nonrecursive). In the present case, we pick $A = K$. To find the computable function $f\colon K \to \mathrm{TOTAL}$, we use the s-m-n Theorem. Let θ be the function defined below: for all $x, y \in \mathbb{N}$,

$$\theta(x, y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$

Note that θ does not depend on y. The function θ is partial computable. Indeed, we have

$$\theta(x, y) = \varphi_x(x) = \varphi_{univ}(x, x).$$

Thus, θ has some index j, so that $\theta = \varphi_j$, and by the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x, y) = \theta(x, y).$$

Let f be the computable function defined such that

$$f(x) = s(j, 1, x)$$

for all $x \in \mathbb{N}$. Then we have

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K \end{cases}$$

for all $y \in \mathbb{N}$. Thus, observe that $\varphi_{f(x)}$ is a total function iff $x \in K$, that is,

$$x \in K \quad \text{iff} \quad f(x) \in \mathrm{TOTAL},$$

where f is computable. As we explained earlier, this shows that TOTAL is not computable (not recursive). □

The above argument can be generalized to yield a result known as Rice's theorem. Let $\varphi_0, \varphi_1, \ldots$ be any indexing of the partial computable functions, and let C be any set of partial computable functions. We define the set P_C as

$$P_C = \{x \in \mathbb{N} \mid \varphi_x \in C\}.$$

We can view C as a property of some of the partial computable functions. For example

$$C = \{\text{all total computable functions}\}.$$

Observe that if $\varphi_i \in C$ for some partial computable function φ_i, equivalently $i \in P_C$, *then $j \in P_C$ for all $j \in \mathbb{N}$ such that $\varphi_j = \varphi_i$.* In other words, if P_C contains the code i of some program P_i computing a partial computable function $\varphi_i \in C$, then P_C contains the code of *every* program computing φ_i. Steve Cook calls such a set P_C a *function index set*. Note that P_C is always infinite, unless $P_C = \emptyset$.

Definition 5.5. We say that a set C of partial computable functions (over $\mathbb{N}$) is *nontrivial* if C is neither empty nor the set of all partial computable functions. Equivalently C is nontrivial iff $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$. We also say that C is *trivial* if $P_C = \emptyset$ or $P_C = \mathbb{N}$.

Theorem 5.6. *(Rice's Theorem, 1953) For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, for any set C of partial computable functions, the set*

$$P_C = \{x \in \mathbb{N} \mid \varphi_x \in C\}$$

is not computable (not recursive) unless C is trivial.

Proof. Assume that C is nontrivial. A set is computable (recursive) iff its complement is computable (recursive) (the proof is trivial). Hence, we may assume that the totally undefined function is not in C, and since $C \neq \emptyset$, let ψ be some other function in C. We produce a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K, \end{cases}$$

for all $y \in \mathbb{N}$. We get f by using the s-m-n Theorem. Let $\psi = \varphi_i$, and define θ as follows:

$$\theta(x, y) = \varphi_{univ}(i, y) + (\varphi_{univ}(x, x) \dot{-} \varphi_{univ}(x, x)),$$

where $\dot{-}$ is the primitive recursive function monus for truncated subtraction; see Section 1.7. Recall that $\varphi_{univ}(x, x) \dot{-} \varphi_{univ}(x, x)$ is defined iff $\varphi_{univ}(x, x)$ is defined iff $x \in K$, and so

$$\theta(x, y) = \varphi_{univ}(i, y) = \varphi_i(y) = \psi(y) \quad \text{iff} \quad x \in K$$

and $\theta(x, y)$ is undefined otherwise. Clearly θ is partial computable, and we let $\theta = \varphi_j$. By the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x, y) = \theta(x, y)$$

for all $x, y \in \mathbb{N}$. Letting f be the computable function such that

$$f(x) = s(j, 1, x),$$

by definition of θ, we get

$$\varphi_{f(x)}(y) = \theta(x, y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$

Thus, f is the desired reduction function. Now we have

$$x \in K \quad \text{iff} \quad f(x) \in P_C,$$

and thus, the characteristic function C_K of K is equal to $C_P \circ f$, where C_P is the characteristic function of P_C. Therefore, P_C is not computable (not recursive), since otherwise, K would be computable, a contradiction. □

Rice's theorem shows that all nontrivial properties of the input/output behavior of programs are undecidable!

It is important to understand that Rice's theorem says that *the set P_C of indices of all partial computable functions equal to some function in a given set C of partial computable functions is not computable* if C is nontrivial, *not that the set C is not computable* if C is nontrivial. The second statement does not make any sense because our machinery only applies to sets of natural numbers (or sets of strings). For example, the set $C = \{\varphi_{i_0}\}$ consisting of a single partial computable function is nontrivial, and being finite, under the second wrong interpretation it would be computable. But we need to consider the set

$$P_C = \{n \in \mathbb{N} \mid \varphi_n = \varphi_{i_0}\}$$

of indices of all partial computable functions φ_n that are equal to φ_{i_0}, and by Rice's theorem, this set is not computable. In other words, it is undecidable whether an arbitrary partial computable function is equal to some fixed partial computable function.

The scenario to apply Rice's theorem to a class C of partial functions is to show that *some partial computable function belongs to C* (C is not empty), *and that some partial computable function does not belong to C* (C is not all the partial computable functions). This demonstrates that C is nontrivial.

In particular, the following properties are undecidable.

Proposition 5.7. *The following properties of partial computable functions are undecidable.*

(a) A partial computable function is a constant function.

(b) Given any integer $y \in \mathbb{N}$, is y in the range of some partial computable function.

(c) Two partial computable functions φ_x and φ_y are identical. More precisely, the set $\{\langle x, y\rangle \mid \varphi_x = \varphi_y\}$ is not computable.

(d) A partial computable function φ_x is equal to a given partial computable function φ_a.

(e) A partial computable function yields output z on input y, for any given $y, z \in \mathbb{N}$.

(f) A partial computable function diverges for some input.

(g) A partial computable function diverges for all input.

The above proposition is left as an easy exercise. For example, in (a), we need to exhibit a constant (partial) computable function, such as $zero(n) = 0$, and a nonconstant (partial) computable function, such as the identity function (or $succ(n) = n + 1$).

A property may be undecidable although it is partially decidable. By partially decidable, we mean that there exists a computable function g that enumerates the set $P_C = \{x \mid \varphi_x \in C\}$. This means that there is a computable function g whose range is P_C. We say that P_C is *listable*, or *computably enumerable*, or *recursively enumerable*. Indeed, g provides a recursive enumeration of P_C, with possible repetitions. Listable sets are the object of the next section.

5.4 Listable (Recursively Enumerable) Sets

In this section and the next our focus is on subsets of $\mathbb{N}$ rather than on numerical functions. Consider the set

$$A = \{k \in \mathbb{N} \mid \varphi_k(a) \text{ is defined}\},$$

where $a \in \mathbb{N}$ is any fixed natural number. By Rice's theorem, A is not computable (not recursive); check this. We claim that A is the range of a computable function g. For this, we use the T-predicate introduced in Definition 4.13. Recall that the predicate $T(i, y, z)$ is defined as follows:

$T(i, y, z)$ holds iff i codes a RAM program, y is an input, and z codes a halting computation of program P_i on input y.

We produce a function which is actually primitive recursive. First, note that A is nonempty (why?), and let x_0 be any index in A. We define g by primitive recursion as follows:

$$\begin{aligned} g(0) &= x_0, \\ g(x+1) &= \begin{cases} \Pi_1(x) & \text{if } T(\Pi_1(x), a, \Pi_2(x)), \\ x_0 & \text{otherwise.} \end{cases} \end{aligned}$$

Since this type of argument is new, it is helpful to explain informally what g does. For every input x, the function g tries finitely many steps of a computation on input a for some partial computable function φ_i computed by the RAM program P_i. Since we need to consider all pairs (i, z) but we only have one variable x at our disposal, we use the trick of packing i and z into $x = \langle i, z \rangle$. Then the index i of the partial function is given by $i = \Pi_1(x)$ and the guess for the code of the computation is given by $z = \Pi_2(x)$. Since Π_1 and Π_2 are projection functions, when x ranges over $\mathbb{N}$, both $i = \Pi_1(x)$ and $z = \Pi_2(x)$ also range over $\mathbb{N}$. Thus *every* partial function φ_i and *every* code for a computation z will be tried, and whenever $\varphi_i(a)$ is defined, which means that there is a correct guess for the code z of the halting computation of P_i on input a, $T(\Pi_1(x), a, \Pi_2(x)) = T(i, a, z)$ is true, and $g(x + 1)$ returns i.

Such a process is called a *dovetailing* computation. This type of argument will be used over and over again.

Definition 5.6. A subset X of $\mathbb{N}$ is *listable*, or *computably enumerable*, or *recursively enumerable*[2] iff either $X = \emptyset$, or X is the range of some total computable function (total recursive function). Similarly, a subset X of Σ^* is *listable* or *computably enumerable*, or *recursively enumerable* iff either $X = \emptyset$, or X is the range of some total computable function (total recursive function).

We will often abbreviate computably enumerable as *c.e*, (and recursively enumerable as *r.e.*). A computably enumerable set is sometimes called a *partially decidable* or *semidecidable* set.

Remark: It should be noted that the definition of a *listable set* (*c.e set* or *r.e. set*) given in Definition 5.6 is *different* from an earlier definition given in terms of acceptance by a Turing machine and it is by no means obvious that these two definitions are equivalent. This equivalence will be proven in Proposition 5.9 ((1) $\Longleftrightarrow$ (4)).

The following proposition relates computable sets and listable sets (recursive sets and recursively enumerable sets).

Proposition 5.8. *A set A is computable (recursive) iff both A and its complement $\overline{A}$ are listable (computably enumerable, recursively enumerable).*

Proof. Assume that A is computable. Then it is trivial that its complement is also computable. Hence, we only have to show that a computable set is listable. The empty set is listable by definition. Otherwise, let $y \in A$ be any element. Then the function f defined such that

$$f(x) = \begin{cases} x & \text{iff } C_A(x) = 1, \\ y & \text{iff } C_A(x) = 0, \end{cases}$$

for all $x \in \mathbb{N}$ is computable and has range A.

Conversely, assume that both A and $\overline{A}$ are listable. If either A or $\overline{A}$ is empty, then A is computable. Otherwise, let $A = f(\mathbb{N})$ and $\overline{A} = g(\mathbb{N})$, for some computable functions f and g. We define the function C_A as follows:

$$C_A(x) = \begin{cases} 1 & \text{if } f(\min y[f(y) = x \ \vee \ g(y) = x]) = x, \\ 0 & \text{otherwise.} \end{cases}$$

[2] Since 1996, the term *recursively enumerable* has been considered old-fashioned by many researchers, and the terms *listable* and *computably enumerable* have been used instead.

The function C_A lists A and $\overline{A}$ in parallel, waiting to see whether x turns up in A or in $\overline{A}$. Note that x must eventually turn up either in A or in $\overline{A}$, so that C_A is a total computable function. □

Our next goal is to show that the listable (recursively enumerable) sets can be given several equivalent definitions.

Proposition 5.9. *For any subset A of $\mathbb{N}$, the following properties are equivalent:*

(1) A is empty or A is the range of a primitive recursive function (Rosser, 1936).

(2) A is listable (computably enumerable, recursively enumerable).

(3) A is the range of a partial computable function.

(4) A is the domain of a partial computable function.

Proof. The implication (1) ⇒ (2) is trivial, since A is listable iff either it is empty or it is the range of a (total) computable function.

To prove the implication (2) ⇒ (3), it suffices to observe that the empty set is the range of the totally undefined function (computed by an infinite loop program), and that a computable function is a partial computable function.

The implication (3) ⇒ (4) is shown as follows. Assume that A is the range of φ_i. Define the function f such that

$$f(x) = \min k[T(i, \Pi_1(k), \Pi_2(k)) \ \wedge \ \mathrm{Res}(\Pi_2(k)) = x]$$

for all $x \in \mathbb{N}$. Since $A = \varphi_i(\mathbb{N})$, we have $x \in A$ iff there is some input $y \in \mathbb{N}$ and some computation coded by z such that the RAM program P_i on input y has a halting computation coded by z and produces the output x. Using the T-predicate, this is equivalent to $T(i, y, z)$ and $\mathrm{Res}(z) = x$. Since we need to search over all pairs (y, z), we pack y and z as $k = \langle y, z\rangle$ so that $y = \Pi_1(k)$ and $z = \Pi_2(k)$, and we search over all $k \in \mathbb{N}$. If the search succeeds, which means that $T(i, y, z)$ and $\mathrm{Res}(z) = x$, we set $f(x) = k = \langle y, z\rangle$, so that f is a function whose domain in the range of φ_i (namely A). Note that the value $f(x)$ is irrelevant, but it is convenient to pick k. Clearly, f is partial computable and has domain A.

The implication (4) ⇒ (1) is shown as follows. The only nontrivial case is when A is nonempty. Assume that A is the domain of φ_i. Since $A \neq \emptyset$, there is some $a \in \mathbb{N}$ such that $a \in A$, which means that for some input y the RAM program P_i has a halting computation coded by z on input a, so if we pack y and z as $k = \langle y, z\rangle$, the quantity

$$\min k[T(i, \Pi_1(k), \Pi_2(k))] = \min\langle y, z\rangle[T(i, y, z)]$$

is defined. We can pick a to be

$$a = \Pi_1(\min k[T(i, \Pi_1(k), \Pi_2(k))]).$$

We define the primitive recursive function f as follows:

$$f(0) = a,$$
$$f(x+1) = \begin{cases} \Pi_1(x) & \text{if } T(i, \Pi_1(x), \Pi_2(x)), \\ a & \text{if } \neg T(i, \Pi_1(x), \Pi_2(x)). \end{cases}$$

Some $y \in \mathbb{N}$ is in the domain of φ_i (namely A) iff the RAM program P_i has a halting computation coded by z on input y iff $T(i, y, z)$ is true. If we pack y and z as $x = \langle y, z \rangle$, then $T(i, y, z) = T(i, \Pi_1(x), \Pi_2(x))$, so if we search over all $x = \langle y, z \rangle$ we search over all y and all z. Whenever $T(i, y, z) = T(i, \Pi_1(x), \Pi_2(x))$ holds, we set $f(x + 1) = y$ since $y \in A$, and if $T(i, y, z) = T(i, \Pi_1(x), \Pi_2(x))$ is false, we return the default value $a \in A$. Our search will find all y such that $T(i, y, z) = T(i, \Pi_1(x), \Pi_2(x))$ holds for some z, which means that all $y \in A$ will be in the range of f. By construction, f only has values in A. Clearly, f is primitive recursive. □

More intuitive proofs of the implications (3) ⇒ (4) and (4) ⇒ (1) can be given. Assume that $A \neq \emptyset$ and that $A = \mathit{range}(g)$, where g is a partial computable function. Assume that g is computed by a RAM program P. To compute $f(x)$, we start computing the sequence

$$g(0), g(1), \ldots$$

looking for x. If x turns up as say $g(n)$, then we output n. Otherwise the computation diverges. Hence, the domain of f is the range of g.

Assume now that A is the domain of some partial computable function g, and that g is computed by some Turing machine M. Since the case where $A = \emptyset$ is trivial, we may assume that $A \neq \emptyset$, and let $n_0 \in A$ be some chosen element in A. We construct another Turing machine performing the following steps: On input n,

(0) Do one step of the computation of $g(0)$

. . .

(n) Do $n + 1$ steps of the computation of $g(0)$

Do n steps of the computation of $g(1)$

. . .

Do 2 steps of the computation of $g(n - 1)$

Do 1 step of the computation of $g(n)$

During this process, whenever the computation of $g(m)$ halts for some $m \leq n$, we output m. Otherwise, we output n_0.

In this fashion, we will enumerate the domain of g, and since we have constructed a Turing machine that halts for every input, we have a total computable function.

The following proposition can easily be shown using the proof technique of Proposition 5.9.

Proposition 5.10. *The following facts hold.*

(1) There is a computable function h such that

$$range(\varphi_x) = dom(\varphi_{h(x)}) \quad \text{for all } x \in \mathbb{N}.$$

(2) There is a computable function k such that

$$dom(\varphi_x) = range(\varphi_{k(x)})$$

and $\varphi_{k(x)}$ is total computable, for all $x \in \mathbb{N}$ such that $dom(\varphi_x) \neq \emptyset$.

The proof of Proposition 5.10 is left as an exercise.

Using Proposition 5.9, we can prove that K is a listable set. Indeed, we have $K = dom(f)$, where

$$f(x) = \varphi_{univ}(x, x) \quad \text{for all } x \in \mathbb{N}.$$

The set

$$K_0 = \{\langle x, y\rangle \mid \varphi_x(y) \text{ is defined}\}$$

is also a listable set, since $K_0 = dom(g)$, where

$$g(z) = \varphi_{univ}(\Pi_1(z), \Pi_2(z)),$$

which is partial computable. It worth recording these facts in the following proposition.

Proposition 5.11. *The sets K and K_0 are listable (c.e., r.e.) sets that are not computable sets (not recursive).*

We can now prove that there are sets that are not listable (not c.e., not r.e.).

Proposition 5.12. *For any indexing of the partial computable functions, the complement $\overline{K}$ of the set*

$$K = \{x \in \mathbb{N} \mid \varphi_x(x) \textit{ is defined}\}$$

is not listable (not computably enumerable, not recursively enumerable).

Proof. If $\overline{K}$ was listable, since K is also listable, by Proposition 5.8, the set K would be computable, a contradiction. □

The sets $\overline{K}$ and $\overline{K_0}$ are examples of sets that are not listable (not c.e., not r.e.). This shows that the listable (c.e., r.e.) sets are not closed under complementation. However, we leave it as an exercise to prove that the listable (c.e., r.e.) sets are closed under union and intersection.

We will prove later on that TOTAL is not listable (not c.e., not r.e.). This is rather unpleasant. Indeed, this means that there is no way of effectively listing all algorithms (all total computable functions). Hence, in a certain sense, the concept of partial computable function (procedure) is more natural than the concept of a (total) computable function (algorithm).

The next two propositions give other characterizations of the listable (c.e., r.e. sets) and of the computable sets (recursive sets). The proofs are left as an exercise.

Proposition 5.13. *The following facts hold.*

(1) A set A is listable (c.e., r.e.) iff either it is finite or it is the range of an injective computable function.

(2) A set A is listable (c.e., r.e.) if either it is empty or it is the range of a monotonic partial computable function.

(3) A set A is listable (c.e., r.e.) iff there is a Turing machine M such that, for all $x \in \mathbb{N}$, M halts on x iff $x \in A$.

Proposition 5.14. *A set A is computable (recursive) iff either it is finite or it is the range of a strictly increasing computable function.*

Another important result relating the concept of partial computable function and that of a listable (c.e., r.e.) set is given below.

Theorem 5.15. *For every unary partial function f, the following properties are equivalent:*

(1) f is partial computable.

(2) The set

$$\{\langle x, f(x)\rangle \mid x \in dom(f)\}$$

is listable (c.e., r.e.).

Proof. Let $g(x) = \langle x, f(x)\rangle$. Clearly, g is partial computable, and

$$range(g) = \{\langle x, f(x)\rangle \mid x \in dom(f)\}.$$

Conversely, assume that

$$range(g) = \{\langle x, f(x)\rangle \mid x \in dom(f)\}$$

for some computable function g. Then we have

$$f(x) = \Pi_2(g(\min y[\Pi_1(g(y)) = x])) \quad \text{for all } x \in \mathbb{N},$$

so that f is partial computable. □

Using our indexing of the partial computable functions and Proposition 5.9, we obtain an indexing of the listable (c.e., r.e.) sets.

Definition 5.7. For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial computable functions, we define the enumeration $W_0, W_1, \ldots$ of the listable (c.e., r.e.) sets by setting

$$W_x = dom(\varphi_x).$$

We now describe a technique for showing that certain sets are listable (c.e., r.e.) but not computable (not recursive), or complements of listable (c.e., r.e.) sets that are not computable (not recursive), or not listable (not c.e., not r.e.), or neither listable (not c.e., not r.e.) nor the complement of a listable (c.e., r.e.) set. This technique is known as *reducibility*.

5.5 Reducibility and Complete Sets

We already used the notion of reducibility in the proof of Proposition 5.5 to show that TOTAL is not computable (not recursive).

Definition 5.8. Let A and B be subsets of $\mathbb{N}$ (or Σ^*). We say that the set A is *many-one reducible* to the set B if there is a *total computable* function (or *total recursive* function) $f\colon \mathbb{N} \to \mathbb{N}$ (or $f\colon \Sigma^* \to \Sigma^*$) such that

$$x \in A \quad \text{iff} \quad f(x) \in B \quad \text{for all } x \in \mathbb{N}.$$

We write $A \leq B$, and for short, we say that A is *reducible* to B. Sometimes, the notation $A \leq_m B$ is used to stress that this is a many-to-one reduction (that is, f is not necessarily injective).

Intuitively, deciding membership in B is as hard as deciding membership in A. This is because any method for deciding membership in B can be converted to a method for deciding membership in A by first applying f to the number (or string) to be tested.

Remark: Besides many-to-one reducibility, there is a also a notion of *one-one reducibility* defined as follows: the set A is *one-one reducible* to the set B if there is a *total* **injective** *computable* function $f\colon \mathbb{N} \to \mathbb{N}$ such that

$$x \in A \quad \text{iff} \quad f(x) \in B \quad \text{for all } x \in \mathbb{N}.$$

We write $A \leq_1 B$. Obviously $A \leq_1 B$ implies $A \leq_m B$ so one-one reducibiity is a stronger notion. We do not need one-one reducibility for our purposes so we will not discuss it. We refer the interested reader to Rogers [37] (especially Chapter 7) for more on reducibility.

The following simple proposition is left as an exercise to the reader.

Proposition 5.16. *Let A, B, C be subsets of $\mathbb{N}$ (or Σ^*). The following properties hold:*

(1) If $A \leq B$ and $B \leq C$, then $A \leq C$.

(2) If $A \leq B$ then $\overline{A} \leq \overline{B}$.

(3) If $A \leq B$ and B is listable (c.e., r.e.), then A is listable (c.e., r.e.).

(4) If $A \leq B$ and A is not listable (not c.e., not r.e.), then B is not listable (not c.e., not r.e.).

(5) If $A \leq B$ and B is computable, then A is computable.

(6) If $A \leq B$ and A is not computable, then B is not computable.

Part (4) of Proposition 5.16 is often useful for proving that some set B is not listable. It suffices to reduce some set known to be nonlistable to B, for example $\overline{K}$. Similarly, Part (6) of Proposition 5.16 is often useful for proving that some set B is not computable. It suffices to reduce some set known to be noncomputable to B, for example K.

Observe that $A \leq B$ implies that $\overline{A} \leq \overline{B}$, but *not* that $\overline{B} \leq \overline{A}$.

Part (3) of Proposition 5.16 may be useful for proving that some set A is listable. It suffices to reduce A to some set known to be listable, for example K. Similarly, Part (5) of Proposition 5.16 may be useful for proving that some set A is computable. It suffices to reduce A to some set known to be computable. In practice, it is often easier to prove directly that A is computable by showing that both A and $\overline{A}$ are listable.

Another important concept is the concept of a complete set.

Definition 5.9. A listable (c.e., r.e.) set A is *complete w.r.t. many-one reducibility* iff every listable (c.e., r.e.) set B is reducible to A, *i.e.*, $B \leq A$.

For simplicity, we will often say *complete* for *complete w.r.t. many-one reducibility*. Intuitively, a complete listable (c.e., r.e.) set is a "hardest" listable (c.e., r.e.) set as far as membership is concerned.

Theorem 5.17. *The following properties hold:*

(1) If A is complete, B is listable (c.e, r.e.), and $A \leq B$, then B is complete.

(2) K_0 is complete.

(3) K_0 is reducible to K. Consequently, K is also complete.

Proof. (1) This is left as a simple exercise.

(2) Let W_x be any listable set (recall Definition 5.7). Then

$$y \in W_x \quad \text{iff} \quad \langle x, y \rangle \in K_0,$$

and the reduction function is the computable function f such that

$$f(y) = \langle x, y \rangle \quad \text{for all } y \in \mathbb{N}.$$

(3) We use the s-m-n Theorem. First, we leave it as an exercise to prove that there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \varphi_{\Pi_1(x)}(\Pi_2(x)) \text{ is defined,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

for all $x, y \in \mathbb{N}$. Then for every $z \in \mathbb{N}$,

$$z \in K_0 \quad \text{iff} \quad \varphi_{\Pi_1(z)}(\Pi_2(z)) \text{ is defined,}$$

iff $\varphi_{f(z)}(y) = 1$ for all $y \in \mathbb{N}$. However,

$$\varphi_{f(z)}(y) = 1 \quad \text{iff} \quad \varphi_{f(z)}(f(z)) = 1,$$

since $\varphi_{f(z)}$ is a constant function. This means that

$$z \in K_0 \quad \text{iff} \quad f(z) \in K,$$

and f is the desired function. □

As a corollary of Theorem 5.17, the set K is also complete.

Definition 5.10. Two sets A and B have the same *degree of unsolvability* or are *equivalent* iff $A \leq B$ and $B \leq A$.

Since K and K_0 are both complete, they have the same degree of unsolvability in the set of listable sets.

We will now investigate the reducibility and equivalence of various sets.

Recall that

$$\text{TOTAL} = \{x \in \mathbb{N} \mid \varphi_x \text{ is total}\}.$$

We define EMPTY and FINITE, as follows:

$$\text{EMPTY} = \{x \in \mathbb{N} \mid \varphi_x \text{ is undefined for all input}\},$$
$$\text{FINITE} = \{x \in \mathbb{N} \mid \varphi_x \text{ is defined only for finitely many input}\}.$$

Obviously, EMPTY $\subset$ FINITE, and since

$$\mathrm{FINITE} = \{x \in \mathbb{N} \mid \varphi_x \text{ has a finite domain}\},$$

we have

$$\overline{\mathrm{FINITE}} = \{x \in \mathbb{N} \mid \varphi_x \text{ has an infinite domain}\},$$

and thus, TOTAL $\subset \overline{\mathrm{FINITE}}$. Since

$$\mathrm{EMPTY} = \{x \in \mathbb{N} \mid \varphi_x \text{ is undefined for all input}\}$$

we have

$$\overline{\mathrm{EMPTY}} = \{x \in \mathbb{N} \mid \varphi_x \text{ is defined for some input}\},$$

we have $\overline{\mathrm{FINITE}} \subseteq \overline{\mathrm{EMPTY}}$.

Proposition 5.18. *We have* $K_0 \leq \overline{\mathrm{EMPTY}}$.

The proof of Proposition 5.18 follows from the proof of Theorem 5.17. We also have the following proposition.

Proposition 5.19. *The following properties hold:*

(1) EMPTY *is not listable (not c.e., not r.e.).*

(2) $\overline{\mathrm{EMPTY}}$ *is listable (c.e., r.e.).*

(3) $\overline{K}$ *and* EMPTY *are equivalent.*

(4) $\overline{\mathrm{EMPTY}}$ *is complete.*

Proof. We prove (1) and (3), leaving (2) and (4) as an exercise (Actually, (2) and (4) follow easily from (3)). First, we show that $\overline{K} \leq$ EMPTY. By the s-m-n Theorem, there exists a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } \varphi_x(x) \text{ is defined,} \\ \text{undefined} & \text{if } \varphi_x(x) \text{ is undefined,} \end{cases}$$

for all $x, y \in \mathbb{N}$. Note that for all $x \in \mathbb{N}$,

$$x \in \overline{K} \quad \text{iff} \quad f(x) \in \mathrm{EMPTY},$$

and thus, $\overline{K} \leq$ EMPTY. Since $\overline{K}$ is not listable, EMPTY is not listable.

We now prove (3). By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = \min z[T(x, \Pi_1(z), \Pi_2(z))], \quad \text{for all } x, y \in \mathbb{N}.$$

Note that

$$x \in \mathrm{EMPTY} \quad \text{iff} \quad g(x) \in \overline{K} \quad \text{for all } x \in \mathbb{N}.$$

Therefore, EMPTY $\leq \overline{K}$, and since we just showed that $\overline{K} \leq$ EMPTY, the sets $\overline{K}$ and EMPTY are equivalent. □

Proposition 5.20. *The following properties hold:*

(1) TOTAL *and* $\overline{\text{TOTAL}}$ *are not listable (not c.e., not r.e.).*

(2) FINITE *and* $\overline{\text{FINITE}}$ *are not listable (not c.e, not r.e.).*

Proof. Checking the proof of Theorem 5.17, we note that $K_0 \leq$ TOTAL and $K_0 \leq \overline{\text{FINITE}}$. Hence, we get $\overline{K_0} \leq \overline{\text{TOTAL}}$ and $\overline{K_0} \leq$ FINITE, and neither $\overline{\text{TOTAL}}$ nor FINITE is listable. If TOTAL was listable, then there would be a computable function f such that TOTAL $= range(f)$. Define g as follows:

$$g(x) = \varphi_{f(x)}(x) + 1 = \varphi_{univ}(f(x), x) + 1$$

for all $x \in \mathbb{N}$. Since f is total and $\varphi_{f(x)}$ is total for all $x \in \mathbb{N}$, the function g is total computable. Let e be an index such that

$$g = \varphi_{f(e)}.$$

Since g is total, $g(e)$ is defined. Then we have

$$g(e) = \varphi_{f(e)}(e) + 1 = g(e) + 1,$$

a contradiction. Hence, TOTAL is not listable. Finally, we show that TOTAL $\leq \overline{\text{FINITE}}$. This also shows that $\overline{\text{FINITE}}$ is not listable. By the s-m-n Theorem, there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \forall z \leq y(\varphi_x(z)\downarrow), \\ \text{undefined} & \text{otherwise,} \end{cases}$$

for all $x, y \in \mathbb{N}$. It is easily seen that

$$x \in \text{TOTAL} \quad \text{iff} \quad f(x) \in \overline{\text{FINITE}} \quad \text{for all } x \in \mathbb{N}.$$ □

From Proposition 5.20, we have TOTAL $\leq \overline{\text{FINITE}}$. It turns out that $\overline{\text{FINITE}} \leq$ TOTAL, and TOTAL and $\overline{\text{FINITE}}$ are equivalent.

Proposition 5.21. *The sets* TOTAL *and* $\overline{\text{FINITE}}$ *are equivalent.*

Proof. We show that $\overline{\text{FINITE}} \leq$ TOTAL. By the s-m-n Theorem, there is a computable function f such that

$$\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \exists z \geq y(\varphi_x(z)\downarrow), \\ \text{undefined} & \text{if } \forall z \geq y(\varphi_x(z)\uparrow), \end{cases}$$

for all $x, y \in \mathbb{N}$. It is easily seen that

$$x \in \overline{\text{FINITE}} \quad \text{iff} \quad f(x) \in \text{TOTAL} \quad \text{for all } x \in \mathbb{N}.$$ □

More advanced topics such that the recursion theorem, the extended Rice Theorem, and creative and productive sets will be discussed in Chapter 6.

Chapter 6

Recursion Theory; More Advanced Topics

This chapter is devoted to three advanced topics of recursion theory:

(1) The recursion theorem.

(2) The extended Rice theorem.

(3) Creative and productive sets and their use in proving a strong version of Gödel's first incompleteness theorem.

The recursion theorem is a deep result and an important technical tool in recursion theory.

The extended Rice theorem gives a characterization of the sets of partial computable functions that are listable in terms of extensions of partial computable functions with finite domains.

Productive and creative sets arise when dealing with truth and provability in arithmetic. The "royal road" to Gödel's first incompleteness theorem is to first prove that for any proof system for arithmetic that only proves true statements (and is rich enough), the set of true sentences of arithmetic is productive. Productive sets are *not* listable in a strong sense, so we deduce that it is impossible to axiomatize the set of true sentences of arithmetic in a computable manner. The set of provable sentences of arithmetic is creative, which implies that it is impossible to decide whether a sentence of arithmetic is provable. This also implies that there are true sentences F such that neither F nor $\neg F$ are provable.

6.1 The Recursion Theorem

The recursion theorem, due to Kleene, is a fundamental result in recursion theory. Let f be a total computable function. Then it turns out that there is some n such that

$$\varphi_n = \varphi_{f(n)}.$$

To understand why such a mysterious result is interesting, consider the recursive definition of the factorial function $fact(n) = n!$ given by

$$\begin{aligned} fact(0) &= 1 \\ fact(n+1) &= (n+1)fact(n). \end{aligned}$$

The trick is to define the partial computable computable function g (defined on $\mathbb{N}^2$) given by

$$\begin{aligned} g(m,0) &= 1 \\ g(m,n+1) &= (n+1)\varphi_m(n) \end{aligned}$$

for all $m, n \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function f such that

$$g(m,n) = \varphi_{f(m)}(n) \quad \text{for all } m, n \in \mathbb{N}.$$

Then the equations above become

$$\begin{aligned} \varphi_{f(m)}(0) &= 1 \\ \varphi_{f(m)}(n+1) &= (n+1)\varphi_m(n). \end{aligned}$$

Since f is (total) recursive, there is some m_0 such that $\varphi_{m_0} = \varphi_{f(m_0)}$, and for m_0 we get

$$\begin{aligned} \varphi_{m_0}(0) &= 1 \\ \varphi_{m_0}(n+1) &= (n+1)\varphi_{m_0}(n), \end{aligned}$$

so the partial recursive function φ_{m_0} *satisfies the recursive definition of factorial*, which means that it is a *fixed point* of the recursive equations defining factorial. Since factorial is a total function, $\varphi_{m_0} = fact$, that is, factorial is a total computable function.

More generally, if a function h (over $\mathbb{N}^k$) is defined in terms of recursive equations of the form

$$h(z_1, \ldots, z_k) = t(h(y_1, \ldots, y_k))$$

where $y_1, \ldots, y_k, z_1, \ldots, z_k$ are expressions in some variables $x_1, \ldots, x_k$ ranging over $\mathbb{N}$ and where t is an expression containing recursive occurrences of h, if we can show that the equations

$$g(m, z_1, \ldots, z_k) = t(\varphi_m(y_1, \ldots, y_k))$$

define a partial computable function g, then we can use the above trick to put them in the form

$$\varphi_{f(m)}(z_1, \ldots, z_k) = t(\varphi_m(y_1, \ldots, y_k)).$$

for some computable function f. Such a formalism is decribed in detail in Chapter XI of Kleene I.M [24]. By the recursion theorem, there is some m_0 such that $\varphi_{m_0} = \varphi_{f(m_0)}$, so φ_{m_0} satisfies the recursive equations

$$\varphi_{m_0}(z_1, \ldots, z_k) = t(\varphi_{m_0}(y_1, \ldots, y_k)),$$

and φ_{m_0} is a fixed point of these recursive equations. If we can show that φ_{m_0} is total, then we found *the* fixed point of this set of recursive equations and $h = \varphi_{m_0}$ is a total computable function. If φ_{m_0} is a partial function, it is still a fixed point. However in general there is more than one fixed point and we don't which one φ_{m_0} is (it could be the partial function undefined everywhere).

Theorem 6.1. *(Recursion Theorem, Version 1) Let $\varphi_0, \varphi_1, \ldots$ be any acceptable indexing of the partial computable functions. For every total computable function f, there is some n such that*

$$\varphi_n = \varphi_{f(n)}.$$

Proof. Consider the function θ defined such that

$$\theta(x, y) = \varphi_{univ}(\varphi_{univ}(x, x), y) \quad \text{for all } x, y \in \mathbb{N}.$$

The function θ is partial computable, and there is some index j such that $\varphi_j = \theta$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = \theta(x, y).$$

Consider the function $f \circ g$. Since it is computable, there is some index m such that $\varphi_m = f \circ g$. Let

$$n = g(m).$$

Since φ_m is total, $\varphi_m(m)$ is defined, and we have

$$\begin{aligned} \varphi_n(y) &= \varphi_{g(m)}(y) = \theta(m, y) = \varphi_{univ}(\varphi_{univ}(m, m), y) = \varphi_{\varphi_{univ}(m,m)}(y) \\ &= \varphi_{\varphi_m(m)}(y) = \varphi_{f\circ g(m)}(y) = \varphi_{f(g(m))}(y) = \varphi_{f(n)}(y), \end{aligned}$$

for all $y \in \mathbb{N}$. Therefore, $\varphi_n = \varphi_{f(n)}$, as desired. □

The recursion theorem can be strengthened as follows.

Theorem 6.2. *(Recursion Theorem, Version 2) Let $\varphi_0, \varphi_1, \ldots$ be any acceptable indexing of the partial computable functions. There is a total computable function h such that for all $x \in \mathbb{N}$, if φ_x is total, then*

$$\varphi_{\varphi_x(h(x))} = \varphi_{h(x)}.$$

Proof. The computable function g obtained in the proof of Theorem 6.1 satisfies the condition

$$\varphi_{g(x)} = \varphi_{\varphi_x(x)},$$

and it has some index i such that $\varphi_i = g$. Recall that c is a computable composition function such that

$$\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.$$

It is easily verified that the function h defined such that

$$h(x) = g(c(x, i)) \quad \text{for all } x \in \mathbb{N}$$

does the job. □

A third version of the recursion Theorem is given below.

Theorem 6.3. *(Recursion Theorem, Version 3) For all $n \geq 1$, there is a total computable function h of $n+1$ arguments, such that for all $x \in \mathbb{N}$, if φ_x is a total computable function of $n+1$ arguments, then*

$$\varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)} = \varphi_{h(x,x_1,\ldots,x_n)},$$

for all $x_1, \ldots, x_n \in \mathbb{N}$.

Proof. Let θ be the function defined such that

$$\theta(x, x_1, \ldots, x_n, y) = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}(y) = \varphi_{univ}(\varphi_{univ}(x, x, x_1, \ldots, x_n), y)$$

for all $x, x_1, \ldots, x_n, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x,x_1,\ldots,x_n)} = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}.$$

It is easily shown that there is a computable function c such that

$$\varphi_{c(i,j)}(x, x_1, \ldots, x_n) = \varphi_i(\varphi_j(x, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

for any two partial computable functions φ_i and φ_j (viewed as functions of $n+1$ arguments) and all $x, x_1, \ldots, x_n \in \mathbb{N}$. Let $\varphi_i = g$, and define h such that

$$h(x, x_1, \ldots, x_n) = g(c(x, i), x_1, \ldots, x_n),$$

for all $x, x_1, \ldots, x_n \in \mathbb{N}$. We have

$$\varphi_{h(x,x_1,\ldots,x_n)} = \varphi_{g(c(x,i),x_1,\ldots,x_n)} = \varphi_{\varphi_{c(x,i)}(c(x,i),x_1,\ldots,x_n)},$$

and using the fact that $\varphi_i = g$,

$$\begin{aligned}\varphi_{\varphi_{c(x,i)}(c(x,i),x_1,\ldots,x_n)} &= \varphi_{\varphi_x(\varphi_i(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)},\\ &= \varphi_{\varphi_x(g(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)},\\ &= \varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)}.\end{aligned}$$ □

As a first application of the recursion theorem, we can show that there is an index n such that φ_n is the constant function with output n. Loosely speaking, φ_n prints its own name. Let f be the computable function such that

$$f(x, y) = x$$

for all $x, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function g such that

$$\varphi_{g(x)}(y) = f(x, y) = x$$

for all $x, y \in \mathbb{N}$. By the Theorem 6.1, there is some n such that

$$\varphi_{g(n)} = \varphi_n,$$

the constant function with value n.

As a second application, we get a very short proof of Rice's theorem. Let C be such that $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$, and let $j \in P_C$ and $k \in \mathbb{N} - P_C$. Define the function f as follows:

$$f(x) = \begin{cases} j & \text{if } x \notin P_C, \\ k & \text{if } x \in P_C, \end{cases}$$

If P_C is computable, then f is computable. By the recursion theorem (Theorem 6.1), there is some n such that

$$\varphi_{f(n)} = \varphi_n.$$

But then we have

$$n \in P_C \quad \text{iff} \quad f(n) \notin P_C$$

by definition of f, and thus,

$$\varphi_{f(n)} \neq \varphi_n,$$

a contradiction. Hence, P_C is not computable.

As a third application, we prove the following proposition.

Proposition 6.4. *Let C be a set of partial computable functions and let*

$$A = \{x \in \mathbb{N} \mid \varphi_x \in C\}.$$

The set A is not reducible to its complement $\overline{A}$.

Proof. Assume that $A \leq \overline{A}$. Then there is a computable function f such that

$$x \in A \quad \text{iff} \quad f(x) \in \overline{A}$$

for all $x \in \mathbb{N}$. By the recursion theorem, there is some n such that

$$\varphi_{f(n)} = \varphi_n.$$

But then,

$$\varphi_n \in C \quad \text{iff} \quad n \in A \quad \text{iff} \quad f(n) \in \overline{A} \quad \text{iff} \quad \varphi_{f(n)} \in \overline{C},$$

contradicting the fact that

$$\varphi_{f(n)} = \varphi_n.$$

□

The recursion theorem can also be used to show that functions defined by recursive definitions other than primitive recursion are partial computable, as we discussed a the beginning of this section. This is the case for the function known as *Ackermann's function* discussed in Section 1.10 and defined recursively as follows:

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

It can be shown that this function is not primitive recursive. Intuitively, it outgrows all primitive recursive functions. However, f is computable, but this is not so obvious. We can use the recursion theorem to prove that f is computable. Using the technique described at the beginning of this section consider the following definition by cases:

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

Clearly, g is partial computable. By the s-m-n Theorem, there is a computable function h such that

$$\varphi_{h(n)}(x, y) = g(n, x, y).$$

The equations defining g yield

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

By the recursion theorem, there is an m such that

$$\varphi_{h(m)} = \varphi_m.$$

Therefore, the partial computable function $\varphi_m(x, y)$ satisfies the equations

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

defining Ackermann's function. We showed in Section 1.10 that $\varphi_m(x, y)$ is a total function, and thus, $f = \varphi_m$ and Ackermann's function is a total computable function.

Hence, the recursion theorem justifies the use of certain recursive definitions. However, note that there are some recursive definitions that are only satisfied by the completely undefined function.

In the next section, we prove the extended Rice theorem.

6.2 Extended Rice Theorem

The extended Rice theorem characterizes the sets of partial computable functions C such that P_C is listable (c.e., r.e.). First, we need to discuss a way of indexing the partial computable functions that have a finite domain. Using the uniform projection function Π (see Definition 4.3), we define the primitive recursive function F such that

$$F(x, y) = \Pi(y + 1, \Pi_1(x) + 1, \Pi_2(x)).$$

We also define the sequence of partial functions $P_0, P_1, \ldots$ as follows:

$$P_x(y) = \begin{cases} F(x, y) - 1 & \text{if } 0 < F(x, y) \text{ and } y < \Pi_1(x) + 1, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Proposition 6.5. *Every P_x is a partial computable function with finite domain, and every partial computable function with finite domain is equal to some P_x.*

The proof is left as an exercise. The easy part of the extended Rice theorem is the following lemma. Recall that given any two partial functions $f\colon A \to B$ and $g\colon A \to B$, we say that g *extends* f iff $f \subseteq g$, which means that $g(x)$ is defined whenever $f(x)$ is defined, and if so, $g(x) = f(x)$.

Proposition 6.6. *Let C be a set of partial computable functions. If there is a listable (c.e., r.e.) set A such that $\varphi_x \in C$ iff there is some $y \in A$ such that φ_x extends P_y, then $P_C = \{x \mid \varphi_x \in C\}$ is listable (c.e., r.e.).*

Proof. Proposition 6.6 can be restated as

$$P_C = \{x \mid \exists y \in A,\ P_y \subseteq \varphi_x\}$$

is listable. If A is empty, so is P_C, and P_C is listable. Otherwise, let f be a computable function such that

$$A = range(f).$$

Let ψ be the following partial computable function:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } P_{f(\Pi_2(z))} \subseteq \varphi_{\Pi_1(z)}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

It is clear that

$$P_C = range(\psi).$$

To see that ψ is partial computable, write $\psi(z)$ as follows:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } \forall w \leq \Pi_1(f(\Pi_2(z))) \\ & [F(f(\Pi_2(z)), w) > 0 \Rightarrow \varphi_{\Pi_1(z)}(w) = F(f(\Pi_2(z)), w) - 1], \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This completes the proof. □

To establish the converse of Proposition 6.6, we need two propositions.

Proposition 6.7. *If P_C is listable (c.e., r.e.) and $\varphi \in C$, then there is some $P_y \subseteq \varphi$ such that $P_y \in C$.*

Proof. Assume that P_C is listable and that $\varphi \in C$. By an s-m-n construction, there is a computable function g such that

$$\varphi_{g(x)}(y) = \begin{cases} \varphi(y) & \text{if } \forall z \leq y[\neg T(x, x, z)], \\ \text{undefined} & \text{if } \exists z \leq y[T(x, x, z)], \end{cases}$$

for all $x, y \in \mathbb{N}$. Observe that if $x \in K$, then $\varphi_{g(x)}$ is a finite subfunction of φ, and if $x \in \overline{K}$, then $\varphi_{g(x)} = \varphi$. Assume that no finite subfunction of φ is in C. Then

$$x \in \overline{K} \quad \text{iff} \quad g(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$. Since P_C is listable, $\overline{K}$ would also be listable, a contradiction. □

As a corollary of Proposition 6.7, we note that TOTAL is not listable.

Proposition 6.8. *If P_C is listable (c.e., r.e.), $\varphi \in C$, and $\varphi \subseteq \psi$, where ψ is a partial computable function, then $\psi \in C$.*

Proof. Assume that P_C is listable. We claim that there is a computable function h such that

$$\varphi_{h(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \varphi(y) & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y \in \mathbb{N}$. Assume that $\psi \notin C$. Then

$$x \in \overline{K} \quad \text{iff} \quad h(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$, a contradiction, since P_C is listable. Therefore, $\psi \in C$. To find the function h we proceed as follows: Let $\varphi = \varphi_j$ and define Θ such that

$$\Theta(x, y, z) = \begin{cases} \varphi(y) & \text{if } T(j, y, z) \wedge \neg T(x, y, w), \text{ for } 0 \leq w < z \\ \psi(y) & \text{if } T(x, x, z) \wedge \neg T(j, y, w), \text{ for } 0 \leq w < z \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Observe that if $x = y = j$, then $\Theta(j, j, z)$ is multiply defined, but since ψ extends φ, we get the same value $\psi(y) = \varphi(y)$, so Θ is a well defined partial function. Clearly, for all $(m, n) \in \mathbb{N}^2$, there is at most one $z \in \mathbb{N}$ so that $\Theta(x, y, z)$ is defined, so the function σ defined by

$$\sigma(x, y) = \begin{cases} z & \text{if } (x, y, z) \in \text{dom}(\Theta) \\ \text{undefined} & \text{otherwise} \end{cases}$$

is a partial computable function. Finally, let

$$\theta(x, y) = \Theta(x, y, \sigma(x, y)),$$

a partial computable function. It is easy to check that

$$\theta(x, y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \varphi(y) & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y \in \mathbb{N}$. By the s-m-n Theorem, there is a computable function h such that

$$\varphi_{h(x)}(y) = \theta(x, y)$$

for all $x, y \in \mathbb{N}$. □

Observe that Proposition 6.8 yields a new proof that $\overline{\text{TOTAL}}$ is not listable (not c.e., not r.e.). Finally we can prove the extended Rice theorem.

Theorem 6.9. *(Extended Rice Theorem) The set P_C is listable (c.e., r.e.) iff there is a listable (c.e., r.e) set A such that*

$$\varphi_x \in C \quad \textit{iff} \quad \exists y \in A\, (P_y \subseteq \varphi_x).$$

Proof. Let $P_C = \mathit{dom}(\varphi_i)$. Using the s-m-n Theorem, there is a computable function k such that

$$\varphi_{k(y)} = P_y \quad \text{for all } y \in \mathbb{N}.$$

Define the listable set A such that

$$A = \mathit{dom}(\varphi_i \circ k).$$

Then

$$y \in A \quad \text{iff} \quad \varphi_i(k(y))\downarrow \quad \text{iff} \quad P_y \in C.$$

Next, using Proposition 6.7 and Proposition 6.8, it is easy to see that

$$\varphi_x \in C \quad \text{iff} \quad \exists y \in A\, (P_y \subseteq \varphi_x).$$

Indeed, if $\varphi_x \in C$, by Proposition 6.7, there is a finite subfunction $P_y \subseteq \varphi_x$ such that $P_y \in C$, but

$$P_y \in C \quad \text{iff} \quad y \in A,$$

as desired. On the other hand, if

$$P_y \subseteq \varphi_x$$

for some $y \in A$, then

$$P_y \in C,$$

and by Proposition 6.8, since φ_x extends P_y, we get

$$\varphi_x \in C.$$

□

6.3 Creative and Productive Sets; Incompleteness in Arithmetic

In this section, we discuss some special sets that have important applications in logic: creative and productive sets. These notions were introduced by Post and Dekker (1944, 1955). The concepts to be described are illustrated by the following situation. Assume that

$$W_x \subseteq \overline{K}$$

for some $x \in \mathbb{N}$ (recall that W_x was introduced in Definition 5.7). We claim that

$$x \in \overline{K} - W_x.$$

Indeed, if $x \in W_x$, then $\varphi_x(x)$ is defined, and by definition of K, we get $x \notin \overline{K}$, a contradiction. Therefore, $\varphi_x(x)$ must be undefined, that is,

$$x \in \overline{K} - W_x.$$

The above situation can be generalized as follows.

Definition 6.1. A set $A \subseteq \mathbb{N}$ is *productive* iff there is a total computable function f such that for every listable set W_x,

$$\text{if} \quad W_x \subseteq A \quad \text{then} \quad f(x) \in A - W_x \quad \text{for all } x \in \mathbb{N}.$$

The function f is called the *productive function of* A. A set A is *creative* if it is listable (c.e., r.e.) and if its complement $\overline{A}$ is productive.

As we just showed, K is creative and $\overline{K}$ is productive. It is also easy to see that TOTAL is productive. But TOTAL is worse than $\overline{K}$, because by Proposition 5.20, $\overline{\text{TOTAL}}$ is not listable.

The following facts are immediate consequences of the definition.

(1) A productive set is not listable (not c.e., not r.e.), since $A \neq W_x$ for all listable sets W_x (the image of the productive function f is a subset of $A - W_x$, which can't be empty since f is total).

(2) A creative set is not computable (not recursive).

Productiveness is a technical way of saying that a nonlistable set A is not listable in a rather strong and constructive sense. Indeed, there is a computable function f such that no matter how we attempt to approximate A with a listable set $W_x \subseteq A$, then $f(x)$ is an element in A not in W_x.

Remark: In Rogers [37] (Chapter 7, Section 3), the definition of a productive set only requires the productive function f to be *partial computable*. However, it is proven in Theorem XI of Rogers that this weaker requirement is equivalent to the stronger requirement of Definition 6.1.

Creative and productive sets arise in logic. The set of theorems of a logical theory is often creative. For example, the set of theorems in Peano's arithmetic is creative, and the set of true sentences of Peano's arithmetic is productive. This yields incompleteness results. We will return to this topic at the end of this section.

Proposition 6.10. *If a set A is productive, then it has an infinite listable (c.e., r.e.) subset.*

Proof. We first give an informal proof. Let f be the computable productive function of A. We define a computable function g as follows: Let x_0 be an index for the empty set, and let

$$g(0) = f(x_0).$$

Assuming that

$$\{g(0), g(1), \ldots, g(y)\}$$

is known, let x_{y+1} be an index for this finite set, and let

$$g(y+1) = f(x_{y+1}).$$

Since $W_{x_{y+1}} \subseteq A$, we have $f(x_{y+1}) \in A$.

For the formal proof, following Rogers [37] (Chapter 7, Section 7, Theorem X), we use the following facts whose proof is left as an exercise:

(1) There is a computable function u such that

$$W_{u(x,y)} = W_x \cup W_y.$$

(2) There is a computable function t such that

$$W_{t(x)} = \{x\}.$$

Letting x_0 be an index for the empty set, we define the function h as follows:

$$\begin{aligned} h(0) &= x_0, \\ h(y+1) &= u(t(f(y)), h(y)). \end{aligned}$$

We define g such that

$$g = f \circ h.$$

It is easily seen that g does the job. □

Another important property of productive sets is the following.

Proposition 6.11. *If a set A is productive, then $\overline{K} \leq A$.*

Proof. Let f be a productive function for A. Using the s-m-n Theorem, we can find a computable function h such that

$$W_{h(y,x)} = \begin{cases} \{f(y)\} & \text{if } x \in K, \\ \emptyset & \text{if } x \in \overline{K}. \end{cases}$$

The above can be restated as follows:

$$\varphi_{h(y,x)}(z) = \begin{cases} 1 & \text{if } x \in K \text{ and } z = f(y), \\ \text{undefined} & \text{if } x \in \overline{K}, \end{cases}$$

for all $x, y, z \in \mathbb{N}$. By the third version of the recursion theorem (Theorem 6.3), there is a computable function g such that

$$W_{g(x)} = W_{h(g(x),x)} \quad \text{for all } x \in \mathbb{N}.$$

Let

$$k = f \circ g.$$

We claim that

$$x \in \overline{K} \quad \text{iff} \quad k(x) \in A \quad \text{for all } x \in \mathbb{N}.$$

Subtituting $g(x)$ for y in the equation for $W_{h(y,x)}$ and using the fact that $W_{g(x)} = W_{h(g(x),x)}$ and $k(x) = f(g(x))$, we get

$$W_{g(x)} = \begin{cases} \{f(g(x))\} = \{k(x)\} & \text{if } x \in K, \\ \emptyset & \text{if } x \in \overline{K}. \end{cases}$$

Because f is a productive function for A, if $x \in \overline{K}$, then $W_{g(x)} = \emptyset \subseteq A$, so $k(x) = f(g(x)) \in A$. Conversely, assume that $k(x) = f(g(x)) \in A$. If $x \in K$, then $W_{g(x)} = \{f(g(x))\}$, so $W_{g(x)} \subseteq A$, and since f is a productive function for A, we have $f(g(x)) \in A - W_{g(x)} = A - \{f(g(x))\}$, a contradiction. Therefore, $x \notin \overline{K}$ and the reduction is achieved. Thus, $\overline{K} \leq A$. □

Using Part (1) of Proposition 6.12 stated next we obtain the converse of Proposition 6.11. Thus a set A is productive iff $\overline{K} \leq A$. This fact is recorded in the next proposition.

The following results can also be shown.

Proposition 6.12. *The following facts hold.*

(1) If A is productive and $A \leq B$, then B is productive.

(2) A is creative iff A is complete.

(3) A is creative iff A is equivalent to K.

(4) A is productive iff $\overline{K} \leq A$.

Part (1) is easy to prove; see Rogers [37] (Chapter 7, Theorem V(b)). Part (2) is proven in Rogers [37] (Chapter 11, Corollary V). Part (3) follows from Part (2) since K is complete. Part (4) follows from Proposition 6.11 and Part (1).

We conclude with a discussion of the significance of the notions of productive and creative sets to logic. A more detailed discussion can be found in Rogers [37] (Chapter 7, Section 8). In Section **??** we discussed Peano arithmetic and the reader is invited to review it. It is convenient to add a countable set of constants $0, 1, 2, \ldots$, denoting the natural numbers to the language of arithmetic, and the new axioms

$$S^n(0) = n, \quad n \in \mathbb{N}.$$

By a now fairly routine process (using a pairing function and an extended pairing function), it is possible to assign a Gödel number $\#(A)$ to every first-order sentence A in the language of arithmetic; see Enderton [11] (Chapter III) or Kleene I.M. [24] (Chapter X). With some labor, it is possible to construct a formula F_x with one free variable x having the following property:

$$\begin{aligned} &n \in K \ \text{ iff } \ (F_n \text{ is true in } \mathbb{N}) \\ &n \notin K \ \text{ iff } \ (F_n \text{ is false in } \mathbb{N}) \text{ iff } (\neg F_x \text{ is true in } \mathbb{N}). \end{aligned}$$

One should not underestimate the technical difficulty of this task. One of Gödel's most original steps in proving his first incompleteness theorem was to define a variant of the formula F_x. Later on, simpler proofs were given, but they are still very technical. The brave reader should attempt to solve Exercises 7.64 and 7.65 in Rogers [37].

Observe that the sentences F_n are special kinds of sentences of arithmetic but of couse there are many more sentences of arithmetic. The following "basic lemma" from Rogers [37] (Chapter 7, Section 8) is easily shown.

Proposition 6.13. *For any two subsets S and T of $\mathbb{N}$, if T is listable and if $S \cap T$ is productive, then S is productive. In particular, if T is computable and if $S \cap T$ is productive, then S is productive.*

With a slight abuse of notation, we say that a set T is sentences of arithmetic is computable (*resp.* listable) iff the set of Gödel numbers $\#(A)$ of sentences A in T is computable (*resp.* listable). Then the following remarkable (historically shocking) facts hold.

Theorem 6.14. *(Unaxiomatizability of arithmetic) The following facts hold.*

(1) The set of sentences of arithmetic true in $\mathbb{N}$ is a productive set. Consequently, the set of true sentences is not listable.

(2) The set of sentences of arithmetic false in $\mathbb{N}$ *is a productive set. Consequently, the set of false sentences is not listable.*

Proof sketch. (1) It is easy to show that the set $\{\neg F_x \mid x \in \mathbb{N}\}$ is computable. Since

$$\{n \in \mathbb{N} \mid \neg F_n \text{ is true in } \mathbb{N}\} = \overline{K}$$

is productive and

$$\begin{aligned}\{A \mid A \text{ is true in } \mathbb{N}\} \cap \{\neg F_x \mid x \in \mathbb{N}\} &= \{\neg F_x \mid \neg F_x \text{ is true in } \mathbb{N}\} \\ &= \{\neg F_x \mid x \in \overline{K}\},\end{aligned}$$

by Proposition 6.13, the set $\{A \mid A \text{ is true in } \mathbb{N}\}$ is also productive.

(2) It is also easy to show that the set $\{F_x \mid x \in \mathbb{N}\}$ is computable. Since

$$\{n \in \mathbb{N} \mid F_n \text{ is false in } \mathbb{N}\} = \overline{K}$$

is productive and

$$\begin{aligned}\{A \mid A \text{ is false in } \mathbb{N}\} \cap \{F_x \mid x \in \mathbb{N}\} &= \{F_x \mid F_x \text{ is false in } \mathbb{N}\} \\ &= \{F_x \mid x \in \overline{K}\},\end{aligned}$$

by Proposition 6.13, the set $\{A \mid A \text{ is false in } \mathbb{N}\}$ is also productive. □

Definition 6.2. A proof system for arithmetic is *axiomatizable* if the set of provable sentences is listable.

Since the set of provable sentences of an axiomatizable proof system is listable, Theorem 6.14 annihilates any hope of finding an axiomatization of arithmetic. Theorem 6.14 also shows that it is impossible to decide effectively (algorithmically) whether a sentence of arithmetic is true. In fact the set of true sentences of arithmetic is *not even listable.*

If we consider proof systems for arithmetic, such as Peano arithmetic, then creative sets show up.

Definition 6.3. A proof system for arithmetic is *sound* if every provable sentence is true (in $\mathbb{N}$). A proof system is *consistent* if there is no sentence A such that both A and $\neg A$ are provable.

Clearly, a sound proof system is consistent.

Assume that a proof system for arithmetic is sound and strong enough so that the formula F_x with the free variable x introduced just before Proposition 6.13 has the following properties:

$$\begin{aligned}n \in K &\text{ iff } (F_n \text{ is provable}) \\ n \notin K &\text{ iff } (F_n \text{ is not provable}).\end{aligned}$$

Peano arithmetic is such a proof system. Then we have the following theorem.

Theorem 6.15. *(Undecidability of provability in arithmetic) Consider any axiomatizable proof system for arithmetic satisfying the hypotheses stated before the statement of the theorem. The following facts hold.*

(1) The set of unprovable sentences of arithmetic is a productive set. Consequently, the set of unprovable sentences is not listable.

(2) The set of provable sentences of arithmetic is a creative set. Consequently, the set of provable sentences is not computable.

Proof sketch. (1) It is easy to show that the set $\{F_x \mid x \in \mathbb{N}\}$ is computable. Since

$$\{n \in \mathbb{N} \mid F_n \text{ is not provable}\} = \overline{K}$$

is productive and

$$\begin{aligned}\{A \mid A \text{ is not provable}\} \cap \{F_x \mid x \in \mathbb{N}\} &= \{F_x \mid F_x \text{ is not provable}\}\\ &= \{F_x \mid x \in \overline{K}\},\end{aligned}$$

by Proposition 6.13, the set $\{A \mid A \text{ is not provable}\}$ is also productive.

(2) Since our proof system is axiomatizable, the set of provable sentences is listable, and by (1), its complement is productive, so the set of provable sentences is creative. □

As a corollary of Theorem 6.15, there is no algorithm to decide whether a sentence of arithmetic is provable or not. But things are worse. Because the set of unprovable sentences of arithmetic is productive, there is a recursive function f, which for any attempt to find a listable subset W of the nonprovable sentences of arithmetic, produces another nonprovable sentence not in W.

Theorem 6.15 also implies Gödel's first incompleteness theorem. Indeed, it is immediately seen that the set $\{F_x \mid \neg F_x \text{ is provable}\}$ is listable (because $\{\neg F_x \mid x \in \mathbb{N}\}$ is computable and $\{A \mid A \text{ is provable}\}$ is listable). But since our proof system is assumed to be sound, $\neg F_x$ provable implies that F_x is *not* provable, so by

$$n \notin K \;\text{ iff }\; (F_n \text{ is not provable}),$$

we have

$$\{x \in \mathbb{N} \mid \neg F_x \text{ is provable}\} \subseteq \{x \in \mathbb{N} \mid F_x \text{ is not provable}\} = \overline{K}.$$

Since $\overline{K}$ is productive and $\{x \in \mathbb{N} \mid \neg F_x \text{ is provable}\}$ is listable, we have

$$W_y = \{x \in \mathbb{N} \mid \neg F_x \text{ is provable}\}$$

for some y, and if f is the productive function associated with $\overline{K}$, then for $x_0 = f(y)$ we have

$$F_{x_0} \in \{F_x \mid F_x \text{ is not provable}\} - \{\neg F_x \mid \neg F_x \text{ is provable}\},$$

that is, *both* F_{x_0} *and* $\neg F_{x_0}$ are *not* provable. Furthermore, since

$$n \notin K \text{ iff } (F_n \text{ is not provable})$$

and

$$n \notin K \text{ iff } (F_n \text{ is false in } \mathbb{N})$$

we see that F_{x_0} is false in $\mathbb{N}$, and so $\neg F_{x_0}$ is true in $\mathbb{N}$. In summary, we proved the following result.

Theorem 6.16. *(Incompleteness in arithmetic (Gödel 1931)) Consider any axiomatizable proof system for arithmetic satisfying the hypotheses stated earlier. Then there exists a sentence F of arithmetic ($F = \neg F_{x_0}$) such that neither F nor $\neg F$ are provable. Furthermore, F is true in $\mathbb{N}$.*

Theorem 6.15 holds under the weaker assumption that the proof system is *consistent* (as opposed to sound), and that there is a formula G with one free variable x such that

$$n \in K \text{ iff } (G_n \text{ is provable}).$$

The formula G is due to Rosser. The incompleteness theorem (Theorem 6.16) also holds under the weaker assumption of consistency. See also Kleene [25] (Chapter 5, Theorem VIII and Corollary 1).

To summarize informally the above negative results:

1. No (effective) axiomatization of mathematics can exactly capture all true statements of arithmetic.

2. From any (effective) axiomatization which yields only true statements of arithmetic, a new true statement can be found *not provable* in that axiomatization.

Fact (2) is what inspired Post to use the term *creative* for the type of sets arising in Definition 6.1. Indeed, one has to be creative to capture truth in arithmetic.

Another (relatively painless) way to prove incompleteness results in arithmetic is to use Diophantine definability; see Section 7.8.

Chapter 7

Listable Sets and Diophantine Sets; Hilbert's Tenth Problem

7.1 Diophantine Equations and Hilbert's Tenth Problem

There is a deep and a priori unexpected connection between the theory of computable and listable sets and the solutions of polynomial equations involving polynomials in several variables with integer coefficients. These are polynomials in $n \geq 1$ variables $x_1, \ldots, x_n$ which are finite sums of *monomials* of the form

$$a x_1^{k_1} \cdots x_n^{k_n},$$

where $k_1, \ldots, k_n \in \mathbb{N}$ are nonnegative integers, and $a \in \mathbb{Z}$ is an integer (possibly negative). The natural number $k_1 + \cdots + k_n$ is called the *degree* of the monomial $a x_1^{k_1} \cdots x_n^{k_n}$.

For example, if $n = 3$, then

1. $5,\ -7$, are monomials of degree 0.

2. $3x_1,\ -2x_2$, are monomials of degree 1.

3. $x_1x_2,\ 2x_1^2,\ 3x_1x_3,\ -5x_2^2$, are monomials of degree 2.

4. $x_1x_2x_3,\ x_1^2x_3,\ -x_2^3$, are monomials of degree 3.

5. $x_1^4,\ -x_1^2x_3^2,\ x_1x_2^2x_3$, are monomials of degree 4.

It is convenient to introduce multi-indices, where an *n-dimensional multi-index* is an n-tuple $\alpha = (k_1, \ldots, k_n)$ with $n \geq 1$ and $k_i \in \mathbb{N}$. Let $|\alpha| = k_1 + \cdots + k_n$. Then we can write

$$x^\alpha = x_1^{k_1} \cdots x_n^{k_n}.$$

For example, for $n = 3$,

$$x^{(1,2,1)} = x_1x_2^2x_3,\ x^{(0,2,2)} = x_2^2x_3^2.$$

Definition 7.1. A *polynomial* $P(x_1, \ldots, x_n)$ in the variables $x_1, \ldots, x_n$ with integer coefficients is a finite sum of monomials of the form

$$P(x_1, \ldots, x_n) = \sum_{\alpha} a_\alpha x^\alpha,$$

where the α's are n-dimensional multi-indices, and with $a_\alpha \in \mathbb{Z}$. The maximum of the degrees $|\alpha|$ of the monomials $a_\alpha x^\alpha$ is called the *total degree* of the polynomial $P(x_1, \ldots, x_n)$. The set of all such polynomials is denoted by $\mathbb{Z}[x_1, \ldots, x_n]$.

Sometimes, we write P instead of $P(x_1, \ldots, x_n)$. We also use variables x, y, z *etc.* instead of $x_1, x_2, x_3, \ldots$.

For example, $2x - 3y - 1$ is a polynomial of total degree 1, $x^2 + y^2 - z^2$ is a polynomial of total degree 2, and $x^3 + y^3 + z^3 - 29$ is a polynomial of total degree 3, and $2x^4 + xyz - 1$ is a polynomial of total degree 4.

Mathematicians have been interested for a long time in the problem of solving equations of the form

$$P(x_1, \ldots, x_n) = 0,$$

with $P \in \mathbb{Z}[x_1, \ldots, x_n]$, seeking only *integer solutions* for $x_1, \ldots, x_n$. What this means is that we try to find n-tuples of integers $(a_1, \ldots, a_n) \in \mathbb{Z}^n$ such that when we assign the value a_i to the variable x_i for $i = 1, \ldots, n$ in the polynomial $P(x_1, \ldots, x_n)$ and evaluate $P(a_1, \ldots, a_n)$ we obtain $P(a_1, \ldots, a_n) = 0$.

Diophantus of Alexandria, a Greek mathematician of the 3rd century, was one of the first to investigate such equations. For this reason, seeking integer solutions of polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$ is referred to as *solving Diophantine equations*.

This problem is not as simple as it looks. The equation

$$2x - 3y - 1 = 0$$

obviously has the solution $x = 2, y = 1$, and more generally $x = -1 + 3a$, $y = -1 + 2a$, for any integer $a \in \mathbb{Z}$.

The equation

$$x^2 + y^2 - z^2 = 0$$

has the solution $x = 3$, $y = 4$, $z = 5$, since $3^2 + 4^2 = 9 + 16 = 25 = 5^2$. More generally, the reader should check that

$$x = t^2 - 1,\ y = 2t,\ z = t^2 + 1$$

is a solution for all $t \in \mathbb{Z}$.

Even solving quadratic Diophantine equations can be harder than it looks. For example, it can be shown that the smallest positive solution to the equation

$$x^2 - 73y^2 - 1 = 0$$

is

$$x = 2,281,249, \quad y = 267,000.$$

See Niven, Zuckermann and Montgomery [31], Section 7.8. The above equation is a special case of what is known as *Pell's equation*, $x^2 - d^2y^2 = 1$. It plays a crucial role in the negative solution of Hilbert's tenth problem (see below).

The equation

$$x^3 + y^3 + z^3 - 29 = 0$$

has the solution $x = 3$, $y = 1$, $z = 1$.

What about the equation

$$x^3 + y^3 + z^3 - 30 = 0?$$

Amazingly, the only known integer solution is

$$(x, y, z) = (-283059965, -2218888517, 2220422932),$$

discovered in 1999 by E. Pine, K. Yarbrough, W. Tarrant, and M. Beck, following an approach suggested by N. Elkies.

And what about solutions of the equation

$$x^3 + y^3 + z^3 - 33 = 0?$$

Until 2019 it was still an open problem but Andrew Booker found the following amazing solution:

$$(8,866,128,975,287,528)^3 + (-8,778,405,442,862,239)^3 + (-2,736,111,468,807,040)^3 = 33.$$

In 1900, at the International Congress of Mathematicians held in Paris, the famous mathematician David Hilbert presented a list of ten open mathematical problems. Soon after, Hilbert published a list of 23 problems. The tenth problem is this:

Hilbert's tenth problem (H10)

Find an algorithm that solves the following problem:

Given as input a polynomial $P \in \mathbb{Z}[x_1, \ldots, x_n]$ with integer coefficients, return YES or NO, according to whether there exist integers $a_1, \ldots, a_n \in \mathbb{Z}$ so that $P(a_1, \ldots, a_n) = 0$; that is, the Diophantine equation $P(x_1, \ldots, x_n) = 0$ has a solution.

It is important to note that at the time Hilbert proposed his tenth problem, a rigorous mathematical definition of the notion of algorithm did not exist. In fact, the machinery needed to even define the notion of algorithm did not exist. It is only around 1930 that precise definitions of the notion of computability due to Turing, Church, and Kleene were formulated, and soon after shown to be all equivalent.

So to be precise, the above statement of Hilbert's tenth should say: find a RAM program (or equivalently a Turing machine) that solves the following problem: ...

In 1970, the following somewhat surprising resolution of Hilbert's tenth problem was reached:

Theorem (Davis-Putnam-Robinson-Matiyasevich)

Hilbert's tenth problem is undecidable; that is, there is no algorithm for solving Hilbert's tenth problem.

In 1962, Davis, Putnam and Robinson had shown that if a fact known as *Julia Robinson hypothesis* could be proven, then Hilbert's tenth problem would be undecidable. At the time, the Julia Robinson hypothesis seemed implausible to many, so it was a surprise when in 1970 Matiyasevich found a set satisfying the Julia Robinson hypothesis, thus completing the proof of the undecidability of Hilbert's tenth problem. It is also a bit startling that Matiyasevich's set involves the Fibonacci numbers.

A detailed account of the history of the proof of the undecidability of Hilbert's tenth problem can be found in Martin Davis' classical paper Davis [8].

Even though Hilbert's tenth problem turned out to have a negative solution, the knowledge gained in developing the methods to prove this result is very significant. What was revealed is that polynomials have considerable expressive powers. This is what we discuss in the next section.

7.2 Diophantine Sets and Listable Sets

We begin by showing that if we can prove that the version of Hilbert's tenth problem *with solutions restricted to belong to* $\mathbb{N}$ is undecidable, then Hilbert's tenth problem (with solutions in $\mathbb{Z}$ is undecidable).

Proposition 7.1. *If we had an algorithm for solving Hilbert's tenth problem (with solutions in* $\mathbb{Z}$*), then we would have an algorithm for solving Hilbert's tenth problem with solutions restricted to belong to* $\mathbb{N}$ *(that is, nonnegative integers).*

Proof. The above statement is not at all obvious, although its proof is short with the help of some number theory. Indeed, by a theorem of Lagrange (Lagrange's four square theorem), *every* natural number m can be represented as the sum of four squares,

$$m = a_0^2 + a_1^2 + a_2^2 + a_3^2, \quad a_0, a_1, a_2, a_3 \in \mathbb{Z}.$$

For a proof, see Niven, Zuckermann and Montgomery [31] (Section 6.4, Theorem 6.26) and Davenport [6] (Chapter V, Section 4). Davenport's proof is more elementary.

We reduce Hilbert's tenth problem restricted to solutions in $\mathbb{N}$ to Hilbert's tenth problem (with solutions in $\mathbb{Z}$). Given a Diophantine equation $P(x_1, \ldots, x_n) = 0$, we can form the polynomial

$$Q = P(u_1^2 + v_1^2 + y_1^2 + z_1^2, \ldots, u_n^2 + v_n^2 + y_n^2 + z_n^2)$$

in the $4n$ variables u_i, v_i, y_i, z_i $(1 \leq i \leq n)$ obtained by replacing x_i by $u_i^2 + v_i^2 + y_i^2 + z_i^2$ for $i = 1, \ldots, n$. If $Q = 0$ has a solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$, then if we set $a_i = p_i^2 + q_i^2 + r_i^2 + s_i^2$, obviously $P(a_1, \ldots, a_n) = 0$ with $a_i \in \mathbb{N}$. Conversely, if $P(a_1, \ldots, a_n) = 0$ with $a_i \in \mathbb{N}$, then by Lagrange's theorem there exist some $p_i, q_i, r_i, s_i \in \mathbb{Z}$ (in fact $\mathbb{N}$) such that $a_i = p_i^2 + q_i^2 + r_i^2 + s_i^2$ for $i = 1, \ldots, n$, and the equation $Q = 0$ has the solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$. Therefore $Q = 0$ has a solution $(p_1, q_1, r_1, s_1, \ldots, p_n, q_n, r_n, s_n,)$ with $p_i, q_i, r_i, s_i \in \mathbb{Z}$ iff $P = 0$ has a solution $(a_1, \ldots, a_n)$ with $a_i \in \mathbb{N}$. If we had an algorithm to decide whether Q has a solution with its components in $\mathbb{Z}$, then we would have an algorithm to decide whether $P = 0$ has a solution with its components in $\mathbb{N}$. □

As consequence, the contrapositive of Proposition 7.1 shows that if the version of Hilbert's tenth problem restricted to solutions in $\mathbb{N}$ is undecidable, so is Hilbert's original problem (with solutions in $\mathbb{Z}$).

In fact, the Davis-Putnam-Robinson-Matiyasevich theorem establishes the undecidability of the version of Hilbert's tenth problem restricted to solutions in $\mathbb{N}$. *From now on, we restrict our attention to this version of Hilbert's tenth problem.*

A key idea is to use Diophantine equations with parameters to *define* sets of numbers.

Example 7.1. For example, consider the polynomial

$$P_1(a, y, z) = (y + 2)(z + 2) - a.$$

For $a \in \mathbb{N}$ fixed, the equation $(y + 2)(z + 2) - a = 0$, equivalently

$$a = (y + 2)(z + 2),$$

has a solution for some $y, z \in \mathbb{N}$ iff a is composite. The variables a, y, z do not play the same role. When we try to solve the equation $(y + 2)(z + 2) - a = 0$, we assume that a is *fixed* and we look for values of y and z that solve the equation. To distinguish between the roles of a and y, z we call y and z parameters. If no solution exists for y, z, then we reject a, that is, we do not include it in the set that we are trying to define. Otherwise we include a in the set that we are defining, namely the set of composites.

Example 7.2. If we now consider the polynomial

$$P_2(a, y, z) = y(2z + 3) - a,$$

for $a \in \mathbb{N}$ fixed, the equation $y(2z + 3) - a = 0$, equivalently

$$a = y(2z + 3),$$

has a solution for some $y, z \in \mathbb{N}$ iff a is not a power of 2. Thus the equation of this example, where y and z are parameters defines the natural numbers that are not a power of 2.

Example 7.3. For a slightly more complicated example, consider the polynomial

$$P_3(a, y) = 3y + 1 - a^2,$$

where y is the parameter. We leave it as an exercise to show that the natural numbers a for which there is some $y \in \mathbb{N}$ such that $3y + 1 - a^2 = 0$, equivalently

$$(a - 1)(a + 1) = 3y,$$

are of the form $a = 3k + 1$ or $a = 3k + 2$, for any $k \in \mathbb{N}$.

In the first case, if we let S_1 be the set of composite natural numbers, then we can write

$$S_1 = \{a \in \mathbb{N} \mid (\exists y, z)((y + 2)(z + 2) - a = 0)\},$$

where it is understood that the existentially quantified variables y, z take their values in $\mathbb{N}$.

In the second case, if we let S_2 be the set of natural numbers that are not powers of 2, then we can write

$$S_2 = \{a \in \mathbb{N} \mid (\exists y, z)(y(2z + 3) - a = 0)\}.$$

In the third case, if we let S_3 be the set of natural numbers that are congruent to 1 or 2 modulo 3, then we can write

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)(3y + 1 - a^2 = 0)\}.$$

A more explicit Diophantine definition for S_3 is

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)((a - 3y - 1)(a - 3y - 2) = 0)\}.$$

The natural generalization is as follows.

Definition 7.2. A set $S \subseteq \mathbb{N}$ of natural numbers is *Diophantine* (or *Diophantine definable*) if there is a polynomial $P(x, y_1, \ldots, y_n) \in \mathbb{Z}[x, y_1, \ldots, y_n]$, with $n \geq 0$[1] such that

$$S = \{a \in \mathbb{N} \mid (\exists y_1, \ldots, y_n)(P(a, y_1, \ldots, y_n) = 0)\},$$

where it is understood that the existentially quantified variables $y_1, \ldots, y_n$ (the *parameters*) take their values in $\mathbb{N}$. Thus $a \in S$ iff there exist some natural numbers $(b_1, \ldots, b_n) \in \mathbb{N}^n$ such that $P(a, b_1, \ldots, b_n) = 0$. More generally, a relation $R \subseteq \mathbb{N}^m$ is *Diophantine* ($m \geq 2$) if there is a polynomial $P(x_1, \ldots, x_m, y_1, \ldots, y_n) \in \mathbb{Z}[x_1, \ldots, x_m, y_1, \ldots, y_n]$, with $n \geq 0$, such that

$$R = \{(a_1, \ldots, a_m) \in \mathbb{N}^m \mid (\exists y_1, \ldots, y_n)(P(a_1, \ldots, a_m, y_1, \ldots, y_n) = 0)\},$$

where it is understood that the existentially quantified variables $y_1, \ldots, y_n$ (*parameters*) take their values in $\mathbb{N}$. Thus $(a_1, \ldots a_m) \in R$ iff there exist some natural numbers $(b_1, \ldots, b_n) \in \mathbb{N}^n$ such that $P(a_1, \ldots, a_m, b_1, \ldots, b_n) = 0$.

[1] We have to allow $n = 0$. Otherwise singleton sets would not be Diophantine.

It is important to note that the simpler definition in which $n = 0$ (there are no parameters) yields a notion which is far too restrictive. Indeed, given a polynomial $P(x)$ of a single variable x, there are only finitely many $a \in \mathbb{N}$ such that $P(a) = 0$. Thus we only obtain finite sets. Similarly, given a polynomial $P(x_1, \ldots, x_n)$ with $n \geq 2$, for any $a_1, \ldots, a_{n-1} \in \mathbb{N}$, there there are only finitely many $a_n \in \mathbb{N}$ such that $P(a_1, \ldots, a_n) = 0$. Again, this class of relations is too restrictive.

The definition of Diophantine definability has the following interpretation as a computational mechanism for defining a set $S \subseteq \mathbb{N}$ in terms of acceptance or rejection. Given $a \in \mathbb{N}$, we can view the search for natural numbers $(b_1, \ldots, b_m) \in \mathbb{N}^m$ such that $P(a, b_1, \ldots, b_m) = 0$ as a computation. If a solution $(b_1, \ldots, b_m)$ is found (making $P(a, b_1, \ldots, b_m) = 0$), then a is accepted, and by definition $a \in S$. If either it can be established that the equation $P(a, y_1, \ldots, y_m) = 0$ has no solution (for $y_1, \ldots, y_m$) or if the search goes on forever, then a is rejected and $a \notin S$. The undecidability of Hilbert's tenth implies that we can't decide if the second alternative arises. Mathematically it is appealing that we obtain a model of computability with universal power that does not require any machine model for its definition.

In Definition 7.2, to define when a set $S \subseteq \mathbb{N}$ is Diophantine we used the variables $y_1, \ldots, y_n$ to denote the parameters occurring in the polynomial $P(x, y_1, \ldots, y_n)$. We did this because in generalizing this notion to m-ary relations it is natural to replace the single variable x by $x_1, \ldots, x_m$, so the use of the variables $y_1, \ldots, y_n$ prevents a clash with the variables $x_1, \ldots, x_m$. However, when we define a set S to be Diophantine we often use the variables $x_1, \ldots, x_n$ instead of $y_1, \ldots, y_n$ since there is very little risk of confusing the variable x with the variables $x_1, \ldots, x_m$.

Example 7.4. The strict order relation $a_1 < a_2$ is defined as follows:

$$a_1 < a_2 \quad \text{iff} \quad (\exists y)(a_1 + 1 + y - a_2 = 0),$$

and the divisibility relation $a_1 \mid a_2$ (a_1 divides a_2) is defined as follows:

$$a_1 \mid a_2 \quad \text{iff} \quad (\exists z)(a_1 z - a_2 = 0).$$

Example 7.5. What about the ternary relation $R \subseteq \mathbb{N}^3$ given by

$$(a_1, a_2, a_3) \in R \quad \text{if} \quad a_1 \mid a_2 \quad \text{and} \quad a_1 < a_3?$$

At first glance it is not obvious how to "convert" a conjunction of Diophantine definitions into a single Diophantine definition, but we can do this using the following squaring trick: given any $n \geq 2$ Diophantine equations in the variables $x_1, \ldots, x_m$,

$$P_1 = 0,\ P_2 = 0,\ \ldots,\ P_n = 0, \tag{$*$}$$

observe that $(*)$ has a solution $(a_1, \ldots, a_m)$, which means that $P_i(a_1, \ldots, a_m) = 0$ for $i = 1, \ldots, n$, iff the single equation

$$P_1^2 + P_2^2 + \cdots + P_n^2 = 0 \tag{$**$}$$

also has the solution $(a_1, \ldots, a_m)$, namely

$$(P_1^2 + P_2^2 + \cdots + P_n^2)(a_1, \ldots a_m) = P_1(a_1, \ldots a_m)^2 + \cdots + P_n(a_1, \ldots a_m)^2 = 0.$$

This is because, since the $P_1(a_1, \ldots, a_m)^2$ for $i = 1 \ldots, n$, are all nonnegative, their sum is equal to zero iff they are *all equal to zero*, that is $P_i(a_1, \ldots, a_m)^2 = 0$ for $i = 1 \ldots, n$, which is equivalent to $P_i(a_1, \ldots, a_m) = 0$ for $i = 1 \ldots, n$.

As a consequence, the set $S \subseteq \mathbb{N}$ defined by n polynomials $P_1, \ldots, P_n$ in $\mathbb{Z}[x, y_1, \ldots, y_p]$ as

$$\{a \in \mathbb{N} \mid (\exists y_1, \ldots, y_p)(P_1(a, y_1, \ldots, y_p) = 0, \ldots, P_n(a, y_1, \ldots, y_p) = 0)\}$$

is actually the Diophantine set defined by

$$\{a \in \mathbb{N} \mid (\exists y_1, \ldots, y_p)(P_1(a, y_1, \ldots, y_p)^2 + \cdots + P_n(a, y_1, \ldots, y_p)^2 = 0)\}.$$

This method also applies to relations $R \subseteq \mathbb{N}^m$ with $m \geq 2$, where we use polynomials $P_1(x_1, \ldots, x_m, y_1, \ldots, y_p), \ldots, P_n(x_1, \ldots, x_m, y_1, \ldots, y_p)$ in $\mathbb{Z}[x_1, \ldots, x_m, y_1, \ldots, y_p]$.

Using this trick, we see that

$$(a_1, a_2, a_3) \in R \quad \text{iff} \quad (\exists u, v)((a_1 u - a_2)^2 + (a_1 + 1 + v - a_3)^2 = 0).$$

We can use the above technique to show that the Diophantine sets are closed under intersection.

Since $(P_1 P_2)(a_1, \ldots, a_m) = 0$ iff $P_1(a_1, \ldots, a_m) = 0$ or $P_2(a_1, \ldots, a_m) = 0$, using this fact it is easily shown that the Diophantine sets are closed under union. However, they are *not* closed under complementation. This is not easy to show directly but it is an immediate consequence of Theorem 7.8 which asserts that the family of Diophantine sets and the family of listable sets coincide.

We can also define the notion of Diophantine function.

7.3 Diophantine Funtions

Definition 7.3. A partial function $f \colon \mathbb{N}^n \to \mathbb{N}$ is *Diophantine* iff its graph $\{(a_1, \ldots, a_n, a_{n+1}) \subseteq \mathbb{N}^{n+1} \mid a_{n+1} = f(a_1, \ldots, a_n)\}$ is Diophantine. This means that there is a polynomial $P(x_1, \ldots, x_{n+1}, y_1, \ldots, y_p) \in \mathbb{Z}[x_1, \ldots, x_{n+1}, y_1, \ldots, y_p]$, with $p \geq 0$, such that $a_{n+1} = f(a_1, \ldots, a_n)$ iff there exist some natural numbers $(b_1, \ldots, b_p) \in \mathbb{N}^p$ such that $P(a_1, \ldots, a_{n+1}, b_1, \ldots, b_p) = 0$. A function $f \colon \mathbb{N}^n \to \mathbb{N}$ is *Diophantine* iff it is Diophantine as a partial function and if it is total, that is, for all $(a_1, \ldots, a_n) \in \mathbb{N}^n$, if $a_{n+1} = f(a_1, \ldots, a_n)$, then the equation $P(a_1, \ldots, a_{n+1}, y_1, \ldots, y_p) = 0$ has a solution (in the variables $y_1, \ldots, y_p$)).

Example 7.6. The pairing function J and the projection functions K, L due to Cantor introduced in Section 4.1 are Diophantine, since

$$\begin{aligned} z = J(x,y) &\quad \text{iff} \quad (x+y)(x+y+1)+2x-2z=0 \\ x = K(z) &\quad \text{iff} \quad (\exists y)((x+y)(x+y+1)+2x-2z=0) \\ y = L(z) &\quad \text{iff} \quad (\exists x)((x+y)(x+y+1)+2x-2z=0). \end{aligned}$$

The definition of J uses no parameter but the definitions of K and L use one parameter.

How extensive is the family of Diophantine sets? The remarkable fact proven by Davis-Putnam-Robinson-Matiyasevich is that they coincide with the listable sets (the recursively enumerable sets). This is a highly nontrivial result. Actually, the crucial point is that a total function is Diophantine iff it is computable. Then this result can be used to prove that a set is Diophantine iff it is listable.

The proof that a total function is Diophantine iff it is computable uses a bit of arithmetic that we now review.

7.4 GCD's, Bezout Identity, Chinese Remainder Theorem

Recall the notion of divisibility from Example 7.4.

Definition 7.4. Given any two integers $m, n \in \mathbb{Z}$, we say that m *divides* n, often written $m \mid n$, if there is some $q \in \mathbb{Z}$ such that $n = mq$. In this case, we call n a *multiple* of m. If $m \neq 0$, the integer q such that $n = mq$ is unique and it is called the *quotient* and it is denoted by n/m.

Observe that if 0 divides n, namely $n = 0q$ for some q, then $n = 0$. So *only* 0 *is divisible by* 0. On the other hand, since $0 = 0q$ for all $q \in \mathbb{Z}$, 0 is divisible by all integers. So even though 0 is divisible by 0, *the quotient* $0/0$ *is undefined* since $0 = 0q$ for all $q \in \mathbb{Z}$. We usually avoid division by 0.

Definition 7.5. Given any two integers $m, n \in \mathbb{Z}$, the *greatest nonnegative common divisor* (for short *gcd*) of m and n is the unique natural number $d \in \mathbb{N}$ such that:

(i) The number d divides both m and n.

(ii) For any $h \in \mathbb{Z}$, if h divides m and n, then h divides d.

The gcd of m and n is denoted as $gcd(m, n)$.

The reader should check that $gcd(0,0) = 0$, $gcd(a, 0) = |a|$ if $a \neq 0$, and $gcd(0, b) = |b|$ if $b \neq 0$.

Example 7.7. Since $15 = 3 \times 5$ and $21 = 3 \times 7$, we see that $gcd(15, 21) = 3$.

Since $657 = 9 \times 73$ and $963 = 9 \times 107$, we see that 9 is a divisor of 657 and 963. Since 73 and 107 are prime (check this fact), 9 is the gcd of 657 and 963.

The following result gives a useful characterization of the gcd in terms of a linear equation.

Proposition 7.2. *(Bezout Identity) For any two integers $m, n \in \mathbb{Z}$, there is a unique natural number $d \in \mathbb{N}$ and some integers $a, b \in \mathbb{Z}$, such that d divides both m and n and*

$$am + bn = d.$$

We have $d = 0$ iff $m = 0$ and $n = 0$. Furthermore, d is the nonnegative gcd of m and n.

Proof. If $d = 0$, since d divides both m and m, we must have $m = n = 0$, and a, b can be chosen arbitrarily. Conversely, if $m = n = 0$, then for any $a, b \in \mathbb{Z}$, we have $d = a0 + b0 = 0$.

Let us now assume that $m \neq 0$ or $n \neq 0$. Consider the set of integers

$$\mathfrak{J} = \{hm + kn \mid h, k \in \mathbb{Z}\}.$$

For $h = 1$ and $k = 0$ we have $m \in \mathfrak{J}$, and for $h = 0$ and $k = 1$ we have $n \in \mathfrak{J}$. Since either $m \neq 0$ or $n \neq 0$, we see that $\mathfrak{J}$ contains some positive natural number (if $m > 0$ we are done, else if $m < 0$ then $(-1)m \in \mathfrak{J}$, with a similar reasoning with $n \neq 0$). Since $\mathfrak{J}$ contains some positive natural number, it contains a smallest one, say d.

We claim that

$$\mathfrak{J} = d\mathbb{Z} = \{dk \mid k \in \mathbb{Z}\}. \tag{$\dagger_B$}$$

Since $d \in \mathfrak{J}$, by definition of $\mathfrak{J}$, we have $d\mathbb{Z} \subseteq \mathfrak{J}$.

Conversely pick any $s \in \mathfrak{J}$. If we divide s by d, we obtain

$$s = dq + r,$$

for some $q \in \mathbb{Z}$ and some r such that $0 \leq r < d$. If $r > 0$, since $s \in \mathfrak{J}$ and $d \in \mathfrak{J}$, they can be expressed as $s = h_1m + k_1n$ and $d = h_2m + k_2n$ for some $h_1, h_2, k_1, k_2 \in \mathbb{Z}$. Then we have

$$r = s - dq = h_1m + k_1n - (h_2m + k_2n)q = (h_1 - h_2q)m + (k_1 - k_2q)n,$$

which shows that $r \in \mathfrak{J}$. But then we have $r \in \mathfrak{J}$ with $r > 0$ and $r < d$, contradicting the fact that d is the smallest positive integer in $\mathfrak{J}$. Therefore $r = 0$, and we proved that $s \in d\mathbb{Z}$. Consequently, $(\dagger_B)$ holds. Since $m, n \in \mathfrak{J} = d\mathbb{Z}$, we see that d divides both m and n. Since $d \in \mathfrak{J}$, there exist $a, b \in \mathbb{Z}$ such that

$$am + bn = d.$$

By construction, $d \in \mathbb{N}$ divides m and n. If any $d' \in \mathbb{Z}$ divides both m and n, since $d = am + bn$, d' also divides d. Therefore d is the nonnegative gcd of m and n. □

Example 7.8. We saw in Example 7.7 that $gcd(15, 21) = 3$. We see immediately that

$$3 \times 15 + (-2) \times 21 = 3$$

We also found that $gcd(657, 963) = 9$. The reader will check that

$$22 \times 657 + (-15) \times 963 = 9.$$

A good algorithmic method for finding gcd's and numbers a, b such that $am + bn = gcd(m, n)$ is the Euclidean algorithm; see Niven, Zuckermann and Montgomery [31], Theorem 1.11. For example, we find that

$$gcd(42823, 6409) = 17$$

and that

$$(-22) \times 42823 + 147 \times 6409 = 17.$$

Definition 7.6. Given any two integers $m, n \in \mathbb{Z}$, not both zero, we say that m *and* n *are relatively prime* if $gcd(m, n) = 1$.

Proposition 7.2 has the following very useful corollary.

Proposition 7.3. *(Bezout Criterion) Given any two integers $m, n \in \mathbb{Z}$, not both zero, m and n are relatively prime if and only if there exists some integers $a, b \in \mathbb{Z}$ such that*

$$am + bn = 1.$$

Proof. If $m \neq 0$ or $n \neq 0$ and $d = gcd(m, n) = 1$, then Proposition 7.2 implies that exists some integers $a, b \in \mathbb{Z}$ such that

$$am + bn = 1.$$

Conversely, any integer d dividing both m and n must divide 1, so $gcd(m, n) = 1$. □

Example 7.9. It is easy to check that $42823 = 17 \times 2519$ and $6409 = 17 \times 377$. Since $gcd(42823, 6409) = 17$, we must have $gcd(2519, 377) = 1$, so 2519 and 377 are relatively prime. We also have

$$(-22) \times 2519 + 147 \times 377 = 1.$$

Neither 2519 nor 377 is prime, as the reader should check.

We now prove a classical result (and a gem) of elementary number theory.

Theorem 7.4. *(Chinese Remainder Theorem) Let $n_1, \ldots, n_m$ $(m \geq 1)$ be any positive integers that are pairwise relatively prime (which means that n_i and n_j are relatively prime for all $i < j$), and let $a_1, \ldots, a_m$ be any integers $(a_i \in \mathbb{Z})$. Then there is some $x \in \mathbb{Z}$ such that*

$$x \equiv a_i \pmod{n_i} \quad i = 1, \ldots, m. \tag{C}$$

If x_0 is any solution of the system of congruences (C), then $x \in \mathbb{Z}$ is a solution of the system (C) iff $x \equiv x_0 \pmod{n}$, where $n = n_1 \cdots n_m$.

Proof. The proof given in Niven, Zuckermann and Montgomery [31] is one of the simplest proofs we are aware of; see Section 2.3, Theorem 2.18. It relies on two simple facts about gcd's:

(1) If m, p, q are positive natural numbers and if m is relatively prime with p and q, then m is relatively prime with pq. This follows easily from Proposition 7.3. See Niven, Zuckermann and Montgomery [31], Theorem 1.8.

(2) If m and n are positive natural numbers and if m and n are relatively prime, then there is some integer x such that $mx \equiv 1 \pmod{n}$. Again, this follows immediately from Proposition 7.3. See Niven, Zuckermann and Montgomery [31], Theorem 2.9.

The case where $m = 1$ is trivial since we can can pick $x = a_1$, so we assume that $m \geq 2$. Let $n = n_1 \cdots n_m$. Each n/n_i is a natural number, and by induction using (1), we see that $gcd(n/n_j, n_j) = 1$ for $j = 1, \ldots, m$. Hence by (2), there is some integer b_j such that

$$(n/n_j)b_j \equiv 1 \pmod{n_j}, \quad j = 1, \ldots, m. \tag{1}$$

Since n/n_j contains n_i for $i \neq j$, we have

$$(n/n_j)b_j \equiv 0 \pmod{n_i}, \quad i \neq j. \tag{2}$$

We claim that a solution of the system of congruences (C) is given by

$$x_0 = \sum_{j=1}^{m} \frac{n}{n_j} b_j a_j, \tag{3}$$

as we now verify. By (1), we have $(n/n_j)b_j a_j \equiv a_j \pmod{n_j}$ for $j = 1, \ldots, m$, and by (2) $(n/n_j)b_j a_j \equiv 0 \pmod{n_i}$ if $i \neq j$, so from (3) by taking the residue modulo n_i we get

$$x_0 \equiv \frac{n}{n_i} b_i a_i \equiv a_i \pmod{n_i},$$

which means that x_0 is a solution of the system (C).

If $x \in \mathbb{Z}$ is another solution of the system

$$x \equiv a_i \pmod{n_i} \quad i = 1, \ldots, m, \tag{C}$$

then by subtraction we obtain

$$x \equiv x_0 \pmod{n_i}, \quad i = 1, \ldots, m,$$

which is easily seen to be equivalent to $x \equiv x_0 \pmod{n}$. Finally, if $x \equiv x_0 \pmod{n}$, then we deduce immediately that x is a solution of the system (C). □

Remark: If $m, n > 0$ and $gcd(m, n) = 1$, an inverse x of m modulo n, namely an integer x such that $mx \equiv 1 \pmod{n}$, can be computed using the Euclidean algorithm; see Niven, Zuckermann and Montgomery [31], Theorem 1.11. Thus the proof of Theorem 7.4 is constructive.

Example 7.10. Consider the system of congruences

$$\begin{aligned} x &\equiv 5 \pmod{7} \\ x &\equiv 7 \pmod{11} \\ x &\equiv 3 \pmod{13}. \end{aligned}$$

We easily check that $n_1 = 7, n_2 = 11, n_3 = 13$ are pairwise relatively prime. We also have $a_1 = 5, a_2 = 7, a_3 = 3$, and $n = 7 \times 11 \times 13 = 1001$. The reader should check that

$$\begin{aligned} (-2) \times n_2n_3 + 21 \times n_1 &= 1 \\ 4 \times n_1n_3 + (-33) \times n_2 &= 1 \\ (-1) \times n_1n_2 + 6 \times n_3 &= 1. \end{aligned}$$

Consequently, we can pick $b_1 = -2$ as the inverse of n_2n_3 modulo n_1, $b_2 = 4$ as the inverse of n_1n_3 modulo n_2, and $b_3 = -1$ as the inverse of n_1n_2 modulo n_3. Theorem 7.4 tells us that a solution is given by

$$x_0 = 11 \times 13 \times (-2) \times 5 + 7 \times 13 \times 4 \times 7 + 7 \times 11 \times (-1) \times 3 = 887.$$

We can then check that $x_0 = 887$ works, and since $887 < 1001$, it is the smallest positive solution.

7.5 Proof of the DPRM: Main Steps

The easier direction is the following result.

Proposition 7.5. *Every Diophantine (total) function is computable. Every Diophantine subset of* $\mathbb{N}$ *is listable (recursively enumerable).*

Proof sketch. First we propose an informal argument for the second statement. Suppose S is given as

$$S = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\},$$

Using the extended pairing function $\langle x_1, \ldots, x_n \rangle_n$ of Section 4.1, we enumerate all n-tuples $(x_1, \ldots, x_n) \in \mathbb{N}^n$, and during this process we compute $P(a, x_1, \ldots, x_n)$. If $P(a, x_1, \ldots, x_n)$ is zero, then we output a, else we go on. This way, S is the range of a computable function, and it is listable.

A more rigorous argument of Proposition 7.5 presented by Martin Davis in [8] proceeds by first proving that if a total function is Diophantine, then it is computable. Then in a second step it is shown that a Diophantine set is listable. To prove this it is necessary to tweak the characterization of a listable set as follows.

Proposition 7.6. *A set $S \subseteq \mathbb{N}$ is listable iff there are two (total) computable functions $f, g\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that*

$$S = \{a \in \mathbb{N} \mid (\exists x)(f(a, x) = g(a, x))\}.$$

Proof. If $S = \emptyset$, then we let f be the constant function equal to 0 and g be the constant function equal to 1. If $S \neq \emptyset$ is listable, then by Definition 5.6 (see also Proposition 5.9), there is a total computable function $h\colon \mathbb{N} \to \mathbb{N}$ such that S is equal to the range of h. If we let f be given by $f(a, x) = a$ and $g(a, x) = h(x)$ for all $a, x \in \mathbb{N}$, then

$$S = \mathrm{range}(h) = \{a \in \mathbb{N} \mid (\exists x)(a = h(x))\} = \{a \in \mathbb{N} \mid (\exists x)(f(a, x) = g(a, x))\}.$$

Conversely, assume that

$$S = \{a \in \mathbb{N} \mid (\exists x)(f(a, x) = g(a, x))\}$$

with f, g total computable. Observe that for any fixed $a \in \mathbb{N}$, the equation $f(a, x) = g(a, x)$ has a solution $x \in \mathbb{N}$ iff the function

$$h(x) = \min x(f(a, x) = g(a, x))$$

is defined, so S is equal to the domain of h. Since f and g are computable and the equality predicate is primitive recursive, the function h is partial computable and by Proposition 5.9, its domain $\mathrm{dom}(h) = S$ is listable. □

A key technical result used in the proof of Proposition 7.5 and Theorem 7.8 is the sequence number theorem. This is a variant of a result that Gödel proved to establish his first incompleteness theorem.

Theorem 7.7. *(Sequence Number Theorem) There is a (total) Diophantine function $(i, u) \mapsto S(i, u)$ such that*

(1) $S(i, u) \leq u$ for all $i, u \in \mathbb{N}$.

(2) For any $N \in \mathbb{N} - \{0\}$ and any sequence $(a_1, \ldots, a_n) \in \mathbb{N}^N$, there is some $u \in \mathbb{N}$ such that

$$S(i, u) = a_i \quad \textit{for} \quad 1 \leq i \leq N.$$

We have $w = S(i, u)$ iff w is the remainder of the division of $K(u)$ by $1 + iL(u)$.

Sketch of proof. Theorem 7.7 is Theorem 1.3 in Davis [8]. The proof needs a slight adjustment because Davis assumes that all numbers in question are *positive* natural numbers, but we don't. The function $(i, u) \mapsto w = S(i, u)$ is defined by the following set of equations where z, v are the parameters:

$$\begin{aligned} 2u &= (x + y)(x + y + 1) + 2x \\ x &= w + z(1 + iy) \\ 1 + iy &= w + v + 1. \end{aligned}$$

In view of Example 7.6, we have $u = J(x, y)$, so

$$x = K(u) \quad \text{and} \quad y = L(u).$$

The third equation asserts that $w < 1 + iy$, so together with the second equation $x = w + z(1 + iy)$, we deduce that w is the remainder of the division of x by $1 + iy \neq 0$. Thus the above equations define a total function S. The second equation implies that $w \leq x = K(u) \leq u$, which is (1).

To prove Condition (2) we use the Chinese remainder theorem, Theorem 7.4.

One might worry that Davis assumes that the numbers a_i are strictly positive, but as we just saw the Chinese remainder theorem is valid even if $a_i = 0$, so there is no problem.

We can now prove that Condition (2) holds as follows. Consider any sequence $(a_1, \ldots, a_N) \in \mathbb{N}^N$. If $N = 1$, pick $y = a_1 + 1$ and proceed to the step where the Chinese remainder theorem is used. If $N \geq 2$, choose $y \in \mathbb{N}$ so that $y > a_i$ for $i = 1, \ldots, N$ and y is divisible by i for $i = 1, \ldots, N - 1$. For example $y = (\max\{a_i\} + 1)(N - 1)!$ will do. We claim that the natural numbers $1 + y, 1 + 2y, \ldots, 1 + Ny$ are pairwise relatively prime.

If not, some natural number $d \geq 1$ divides both $1 + iy$ and $1 + jy$ for some i, j such that $1 \leq i < j \leq N$. Then d divides $j(1 + iy) - i(1 + jy) = j - i$, which implies that $1 \leq d < N$. However y was chosen so that it is divisible by k for $k = 1, \ldots, N - 1$, so d would divide y, and since d also divides $1 + iy$, we must have $d = 1$.

We can now apply the Chinese remainder theorem with $n_i = 1 + id$ for $i = 1, \ldots, N$. Therefore there is some $x \in \mathbb{N}$ such that

$$\begin{aligned}
x &\equiv a_1 \pmod{1 + y}\\
x &\equiv a_2 \pmod{1 + 2y}\\
&\vdots\\
x &\equiv a_N \pmod{1 + Ny}.
\end{aligned}$$

Let $u = J(x, y)$ so that $x = K(u)$ and $y = L(u)$. We have

$$K(u) \equiv a_i \pmod{1 + iL(u)}, \quad i = 1, \ldots, N.$$

By definition of y, we also have $a_i < y = L(u) < 1 + iL(u)$, and then we see that a_i is the remainder of the division of $K(u)$ by $1 + iL(u)$, which is equal to $S(i, u)$ by definition of S. □

Interestingly, Davis states that the function S is primitive recursive, but does not provide a proof. However, a proof can be extracted from his book Davis [7]; see Chapter 3, Sections 1 and 2.

The proof that S is primitive recursive uses the remainder function $rem\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined such that if $n > 0$, then $rem(m, n) = r$ is the remainder of the division of m by n, namely the unique $r \in \mathbb{N}$ such that $r < n$ and $m = nq+r$ for some $q \in \mathbb{N}$, else $rem(m, 0) = m$. We leave it as an exercise to prove that rem is primitive recursive. Using rem we define S as

$$S(i, u) = rem(K(u), 1 + iL(u)).$$

See the proof of Theorem 2.4 in Davis [7], observing that since here the index i ranges from 1 to N, the term $1 + L(u)(i + 1)$ of Davis' proof can be replaced by $1 + iL(u)$.

Let us now assume that the total function $f \colon \mathbb{N}^m \to \mathbb{N}$ is Diophantine, so that there is a polynomial $P(x_1, \ldots, x_n, z, y_1, \ldots, y_p)$ such that

$$c = f(a_1, \ldots, a_n) \quad \text{iff} \quad (\exists b_1, \ldots, b_p)(P(a_1, \ldots, a_m, c, b_1, \ldots, b_p) = 0).$$

By grouping the monomials with positive coefficients together and the monomials with negative coefficients together we can write

$$P(x_1, \ldots, x_n, z, y_1, \ldots, y_p) = Q(x_1, \ldots, x_n, z, y_1, \ldots, y_p) - R(x_1, \ldots, x_n, z, y_1, \ldots, y_p),$$

where $Q(x_1, \ldots, x_n, z, y_1, \ldots, y_p)$ and $R(x_1, \ldots, x_n, z, y_1, \ldots, y_p)$ have *positive integer coefficients*. Using Q and R we can express the definition of f as

$$c = f(a_1, \ldots, a_n) \text{ iff } (\exists b_1, \ldots, b_p)(Q(a_1, \ldots, a_m, c, b_1, \ldots, b_p) = R(a_1, \ldots, a_m, c, b_1, \ldots, b_p)).$$

Using the sequence number theorem we we can find $u \in \mathbb{N}$ such that $c = S(1, u)$, $b_1 = S(2, u), \ldots, b_p = S(p + 1, u)$, and we deduce that

$$\begin{aligned} f(a_1, \ldots, a_n) = S(1, \min\nolimits_u[Q(a_1, \ldots, a_m, S(1, u), S(2, u), \ldots, S(p + 1, u)) \\ = R(a_1, \ldots, a_m, S(1, u), S(2, u), \ldots, S(p + 1, u))]). \end{aligned}$$

Now we explained before that the polynomials Q and R having positive integer coefficients compute *primitive recursive* functions, which are special kinds of total functions. Since S is also primitive recursive, using the fact that the computable functions are closed under composition and minimization if it yields a total function (which is the case since f is assumed to be total), we deduce that f is computable.

We can now tackle Diophantine sets. Assume that S is Diophantine so that there is a polynomial $P(x, y_1, \ldots, y_p)$ such that

$$a \in S \quad \text{iff} \quad (\exists b_1, \ldots, b_p)(P(a, b_1, \ldots, b_p) = 0).$$

As above, we can write

$$P(x, y_1, \ldots, y_p) = Q(x, y_1, \ldots, y_p) - R(x, y_1, \ldots, y_p),$$

where $Q(x, y_1, \ldots, y_p)$ and $R(x, y_1, \ldots, y_p)$ have *positive integer coefficients*. Then we have

$$a \in S \quad \text{iff} \quad (\exists b_1, \ldots, b_p)(Q(a, b_1, \ldots, b_p) = R(a, b_1, \ldots, b_p)),$$

and by the sequence number theorem we can find $u \in \mathbb{N}$ such that $b_1 = S(1,u), \ldots, b_p = S(p,u)$, so

$$a \in S \quad \text{iff} \quad (\exists u)(Q(a, S(1,u), \ldots, S(p,u)) = R(a, S(1,u), \ldots, S(p,u))).$$

Since Q and R compute primitive recursive functions and S is primitive recursive, by Proposition 7.6, S is listable □

The main theorem of the theory of Diophantine sets and functions is the following deep result.

Theorem 7.8. *(Davis-Putnam-Robinson-Matiyasevich, 1970) Every total computable function is Diophantine. Every listable subset of* $\mathbb{N}$ *is Diophantine.*

Theorem 7.8 is often referred to as the *DPRM theorem.* A complete proof of Theorem 7.8 is provided in Davis [8]. We provide all the steps except the most technical one, the fact that the exponential function $h(n,k) = n^k$ is Diophantine.

Almost complete proof. As noted by Davis, although the proof is certainly long and nontrivial, it only uses elementary facts of number theory, nothing more sophisticated than the Chinese remainder theorem. Nevetherless, the proof is a tour de force.

One of the most difficult steps is to show that the exponential function $h(n,k) = n^k$ is Diophantine. This is done using the Pell equation. According to Martin Davis, the proof given in Davis [8] uses a combination of ideas from Matiyasevich and Julia Robinson. Matiyasevich's proof used the Fibonacci numbers.

We now provide details for all the steps of the proof, except the first one.

Step 1. The most difficult and most technical step is to prove that the exponential function $(n,k) \mapsto n^k$ is Diophantine. This involves proving twenty four "easy lemmas," which takes six pages (this is Section 2). The fact that the exponential function is Diophantine is established in Section 3; this is Theorem 3.3 (Section 3 has four pages).

There is a small issue, which is that Davis [8] assumes that all variables range over *positive integers*, so his proof that the exponential function $h(n,k) = n^k$ is Diophantine works only for $n, k > 0$. However, as in the 1976 survey paper by Davis, Matiyasevich and Robinson [9], we assume that the variables may take the value 0, that is, belong to $\mathbb{N}$. This problem is easily taken care of. If E is the set of Equations I-XII (with parameters) listed on Pages 244 and 247 of Davis [8] in which the variables (n,k,m) define the exponential function h in the sense that there are values of the parameters that satisfy E iff $m = h(n,k) = n^k$, create the new equation with the extra new parameters n', k', k'',

$$((n - n' - 1)^2 + (k - k' - 1)^2 + E^2)(k^2 + (m-1)^2)(n^2 + (k - k'' - 1)^2 + m^2) = 0. \tag{$*$}$$

The above equation has a solution with respect to the parameters iff

$$(n - n' - 1)^2 + (k - k' - 1)^2 + E^2 = 0 \tag{$*_1$}$$

or

$$k^2 + (m-1)^2 = 0 \tag{$*_2$}$$

or

$$n^2 + (k - k'' - 1)^2 + m^2 = 0. \tag{$*_3$}$$

The Equation $(*_1)$ is equivalent to

$$\begin{aligned} n &= n' + 1 \\ k &= k' + 1 \\ E &= 0, \end{aligned}$$

which are equivalent to

$$n > 0, \;\; k > 0, \;\; E = 0. \tag{$*_4$}$$

These equations have a solution in the parameters iff $n, k > 0$ and $m = n^k$.

The Equation $(*_2)$ is equivalent to

$$k = 0, \;\; m = 1, \tag{$*_5$}$$

which defines the exponential for $k = 0$ since $n^0 = 1$ for all $n \in \mathbb{N}$.

The Equation $(*_3)$ is equivalent to

$$\begin{aligned} n &= 0 \\ k &= k'' + 1 \\ m &= 0, \end{aligned}$$

which is equivalent to

$$n = 0, \;\; k > 0, \;\; m = 0, \tag{$*_6$}$$

which define the exponential for $n = 0$ and $k > 0$ since $0^k = 0$ for all $k > 0$. In summary, the Equation $(*)$ defines the exponential function $m = n^k$ for all $m, k \in \mathbb{N}$.

Step 2. Use the fact that the exponential is Diophantine to prove that two crucial functions are Diophantine:

$$\begin{aligned} f(n,k) &= \binom{n}{k} \\ g(n) &= n!. \end{aligned}$$

This is proven in Theorem 4.1. We prove that the functions f and g are Diophantine provided that the exponential function is Diophantine in Section 7.7.

At this stage we know that the Diophantine relations are closed under conjunction, disjunction, and existential quantifiers. In order to prove that the Diophantine functions are closed under primitive recursion and minimization (if the function obtained by minimization is total) it is critical to prove closure under bounded universal quantification. This is the next step.

Step 3.

Definition 7.7. Call a predicate (relation) φ a *Diophantine predicate* if it is of the form

$$\varphi(x_1, \ldots, x_n) \equiv (\exists y_1, \ldots, y_p)(P(x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

where $P(x_1, \ldots, x_m, y_1, \ldots, y_p)$ is a polynomial with integer coefficients.

Of course, for any $(a_1, \ldots, a_m) \in \mathbb{N}^m$, $\varphi(a_1, \ldots, a_m)$ holds (equivalently $(a_1, \ldots, a_m) \in \varphi$) iff there is some $(b_1, \ldots, b_p) \in \mathbb{N}^p$ such that $P(a_1, \ldots, a_m, b_1, \ldots, b_p) = 0$.

It is convenient to abbreviate $(\exists y_1, \ldots, y_p)$ as $(\exists \overline{y})$. Given two Diophantine predicates $\varphi = (\exists \overline{u})(P(x_1, \ldots, x_m, \overline{u}) = 0)$ and $\psi = (\exists \overline{v})(Q(x_1, \ldots, x_m, \overline{v}) = 0)$ over the *same variables* $x_1, \ldots, x_m$, we define the predicates

$$\begin{aligned}
\varphi \wedge \psi &\equiv (\exists \overline{u})(P(x_1, \ldots, x_m, \overline{u}) = 0) \wedge (\exists \overline{v})(Q(x_1, \ldots, x_m, \overline{v}) = 0)\\
\varphi \vee \psi &\equiv (\exists \overline{u})(P(x_1, \ldots, x_m, \overline{u}) = 0) \vee (\exists \overline{v})(Q(x_1, \ldots, x_m, \overline{v}) = 0)\\
\exists z \varphi &\equiv \exists z (\exists \overline{u})(P(x_1, \ldots, x_m, \overline{u}) = 0),
\end{aligned}$$

where z is any variable occurring or not in φ. We may rename variables so that $\overline{u}$ and $\overline{v}$ are disjoint and that z does not occur in $\overline{u}$.

The above predicates are Diophantine (using the squaring trick and the product trick) because

$$\begin{aligned}
\varphi \wedge \psi &\equiv (\exists \overline{u})(\exists \overline{v})(P(x_1, \ldots, x_m, \overline{u})^2 + Q(x_1, \ldots, x_m, \overline{v})^2 = 0)\\
\varphi \vee \psi &\equiv (\exists \overline{u})(\exists \overline{v})(P(x_1, \ldots, x_m, \overline{u})Q(x_1, \ldots, x_m, \overline{v}) = 0)\\
\exists z \varphi &\equiv \exists z (\exists \overline{u})(P(x_1, \ldots, x_m, \overline{u}) = 0).
\end{aligned}$$

Observe that if $z = x_i$ for some variable x_i, then $m \geq 2$ and $\exists z \varphi$ is a predicate only involving the variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m$, so that it defines a subset of $\mathbb{N}^{m-1}$. We will use these closure properties when constructing Diophantine predicates.

In general universal quantification applied to a Diophantine predicate does not yield a Diophantine predicate, but bounded universal quantification does.

Definition 7.8. Given a polynomial $P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p)$ with integer coefficients, the *bounded existentially quantified predicate*

$$(\exists z \leq y)(\exists y_1, \ldots, y_p)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

holds iff for any $a_1, \ldots, a_m \in \mathbb{N}$ and any $b \in \mathbb{N}$, there is some $c \leq b$ and some $b_1, \ldots, b_p \in \mathbb{N}$ such that $P(b, c, a_1, \ldots, a_m, b_1, \ldots, b_p) = 0$ holds. The *bounded universally quantified predicate*

$$(\forall z \leq y)(\exists y_1, \ldots, y_p)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

holds iff for any $a_1, \ldots, a_m \in \mathbb{N}$ and any $b \in \mathbb{N}$, for every $c \leq b$, there are some $b_1, \ldots, b_p \in \mathbb{N}$ such that $P(b, c, a_1, \ldots, a_m, b_1, \ldots, b_p) = 0$ holds.

Proposition 7.9. *Given a polynomial $P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p)$ with integer coefficients, the predicate*

$$(\forall z \leq y)(\exists y_1, \ldots, y_p)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

holds iff the predicate

$$(\exists u)(\forall z \leq y)(\exists y_1 \leq u) \cdots (\exists y_p \leq u)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

holds.

Proof. The second statement obviously implies the first. If the predicate

$$(\forall z \leq y)(\exists y_1, \ldots, y_p)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

holds, then for any $b \in \mathbb{N}$ and any $a_1, \ldots, a_m \in \mathbb{N}$, for each $k = 0, \ldots, b$, there exist $b_1^{(k)}, \ldots, b_p^{(k)} \in \mathbb{N}$ such that $P(b, k, a_1, \ldots, a_m, b_1^{(k)}, \ldots, b_p^{(k)}) = 0$ for $k = 0, \ldots, b$. If we pick

$$u = \max\{b_j^{(k)} \mid 0 \leq k \leq b,\ 1 \leq j \leq p\},$$

then

$$(\exists u)(\forall z \leq y)(\exists y_1 \leq u) \cdots (\exists y_p \leq u)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

holds. □

We have the following key theorem.

Theorem 7.10. *(Bounded Quantifier Theorem) Given any polynomial $P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p)$ with integer coefficients, the bounded existentially quantified predicate*

$$(\exists z \leq y)(\exists y_1, \ldots, y_p)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

and the bounded universally quantified predicate

$$(\forall z \leq y)(\exists y_1, \ldots, y_p)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

are also Diophantine.

Recall that $x \leq y$ is Diophantine definable as $y = x + x'$. The first statement is easy to prove since

$$(\exists z \leq y)(\exists y_1, \ldots, y_p)(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0)$$

holds iff

$$(\exists z, y_1, \ldots, y_p)[(P(y, z, x_1, \ldots, x_m, y_1, \ldots, y_p) = 0) \wedge (z \leq y)].$$

The proof of the second statement is far more complicated. In particular is uses the fact that the factorial function $n \mapsto n!$ and the binomial $\binom{n}{k}$ are Diophantine (both of which use the fact that the exponential function $(n, k) \mapsto n^k$ is Diophantine). One proof is given in Davis [8]; see Theorem 5.1. A slightly shorter proof is given in Davis, Matiyasevich and Robinson [9]; see Section 4. Here is this crucial result and its very beautiful and clever proof.

Proposition 7.11. *Let $P(x, y, k, z_1, \ldots, z_\nu)$ be a polynomial with x, y and k among its parameters and $z_1, \ldots, z_\nu$ its variables. Then*

$$(\forall k \leq x)(\exists z_1 \leq y) \cdots (\exists z_\nu \leq y)(P(x, y, k, z_1, \ldots, z_\nu) = 0) \qquad (\dagger_1)$$

holds if and only if

$$(\exists b_1, \ldots, b_\nu)\left[\binom{b_1}{y+1} \equiv \cdots \equiv \binom{b_\nu}{y+1} \equiv P(x, y, Q! - 1, b_1, \ldots, b_\nu) \right.$$
$$\left. \equiv 0 \left(\operatorname{mod} \binom{Q! - 1}{x+1}\right)\right] \qquad (\dagger_2)$$

holds, where $Q(x, y)$ is a polynomial such that

$$Q(x, y) > |P(x, y, k, z_1, \ldots, z_\nu)| + 2x + y + 1, \qquad (*_1)$$

for all $k \leq x$ and all $z_1 \leq y, \ldots, z_\nu \leq y$. Also $b_1, \ldots, b_\nu$ may be chosen such that $b_i \leq \binom{Q!-1}{x+1}$.

Proof. Consider the product

$$\binom{Q! - 1}{x+1} = \frac{(Q! - 1)!}{(x+1)!(Q! - 1 - (x+1))!} = \frac{(Q! - 1)(Q! - 2) \cdots (Q! - 1 - (x+1) + 1)}{(x+1)!}$$
$$= (Q! - 1)\left(\frac{Q!}{2} - 1\right) \cdots \left(\frac{Q!}{x+1} - 1\right).$$

Since $Q(x, y) > |P(x, y, k, z_1, \ldots, z_\nu)| + 2x + y + 1 \geq 2x + 2 > x + 1$, all the factors on the right-hand side are integers.

Claim 1. If a prime p divides $\binom{Q!-1}{x+1}$, then $p > Q$.

For this we prove that every prime $p \leq Q$ divides $Q!/(k+1)$ for all $k \leq x$. Indeed, since $Q \geq 2x + 2$ and $k \leq x$, we have $2(k+1) \leq 2x + 2$, so

$$Q! = Q(Q-1) \cdots (2x+2) \cdots 2(k+1) \cdots (k+1)k!.$$

If $p = k + 1$, then $k + 1$ still occurs in $Q!/(k+1)$, and if $p \leq Q$ and $p \neq k + 1$, then p occurs in $Q!/(k+1)$.

Now if a prime p divides $\binom{Q!-1}{x+1}$, then p divides some factor $\frac{Q!}{k+1} - 1$ (with $k \leq x$), so if $p \leq Q$, then from the previous fact p divides $Q!/(k+1)$, which implies that p divides 1, a contradiction.

Claim 2. Any two distinct factors $\frac{Q!}{i+1} - 1$ and $\frac{Q!}{j+1} - 1$ $(i, j \leq x)$ are relatively prime.

If a prime p divides both $\frac{Q!}{i+1} - 1$ and $\frac{Q!}{j+1} - 1$, then $\frac{Q!}{i+1} - 1 = k_1 p$ and $\frac{Q!}{j+1} - 1 = k_2 p$ for some natural numbers k_1, k_2, so

$$Q! - i - 1 = k_1(i+1)p, \quad Q! - j - 1 = k_2(j+1)p,$$

and by substraction

$$j - i = (k_1(i+1) - k_2(j+1))p,$$

which means that p divides $|j - i|$. However, $i, j \leq Q$ and by Claim 1, $p > Q$, so $i = j$, which means that $\frac{Q!}{i+1} - 1$ and $\frac{Q!}{j+1} - 1$ $(i, j \leq x)$ are relatively prime if $i \neq j$.

Claim 3. If a prime p_k divides $\frac{Q!}{k+1} - 1$ $(k \leq x)$, then

$$Q! - 1 \equiv k \pmod{p_k}.$$

Since $\frac{Q!}{k+1} - 1 = qp_k$ for some natural number q, we have $Q! - k - 1 = q(k+1)p_k$, so

$$Q! - 1 = k + q(k+1)p_k,$$

namely $Q! - 1 \equiv k \pmod{p_k}$.

Claim 4. For any choice of a prime p_k dividing $\frac{Q!}{k+1} - 1$ for $k \leq x$, we have

$$P(x, y, Q! - 1, b_1, \ldots, b_\nu) \equiv P(x, y, k, \mathrm{Rem}(b_1, p_k), \ldots, \mathrm{Rem}(b_\nu, p_k)) \pmod{p_k},\ k \leq x, \quad (*_2)$$

where $\mathrm{Rem}(b_i, p_k)$ is the remainder of the division of b_i by p_k.

Claim 4 follows immediately from Claim 3 by taking residues modulo p_k in the polynomial $P(x, y, Q! - 1, b_1, \ldots, b_\nu)$.

We are now ready for the proof itself.

Step a. First we prove that $(\dagger_2)$ implies $(\dagger_1)$. Assume that there exists some natural numbers $b_1, \ldots, b_\nu$ such that

$$\binom{b_1}{y+1} \equiv \cdots \equiv \binom{b_\nu}{y+1} \equiv P(x, y, Q! - 1, b_1, \ldots, b_\nu) \equiv 0 \left(\mathrm{mod} \binom{Q! - 1}{x+1}\right).$$

Since any chosen prime p_k dividing $\frac{Q!}{k+1} - 1$ also divides divides $\binom{Q!-1}{x+1}$, we deduce from the congruence

$$\binom{b_i}{y+1} \equiv 0 \left(\mathrm{mod} \binom{Q! - 1}{x+1}\right)$$

that p_k divides $b_i(b_i - 1) \cdots (b_i - y)$ for $i = 1, \ldots, \nu$, so p_k divides some factor $b_i - h$ with $h \leq y$, which implies that $\mathrm{Rem}(b_i, p_k) \leq y$ for $i = 1, \ldots, \nu$. By $(*_1)$ and Claim 1, we have

$$|P(x, y, k, \mathrm{Rem}(b_1, p_k), \ldots, \mathrm{Rem}(b_\nu, p_k))| \leq Q < p_k. \qquad (*_3)$$

By hypothesis, since p_k divides $\binom{Q!-1}{x+1}$, we have

$$P(x, y, Q! - 1, b_1, \ldots, b_\nu) \equiv 0 \pmod{p_k},$$

for all $k \leq x$, and by $(*_2)$ and $(*_3)$, we deduce that

$$P(x, y, k, \mathrm{Rem}(b_1, p_k), \ldots, \mathrm{Rem}(b_\nu, p_k)) = 0,$$

which is ($\dagger_1$) of our proposition with $z_i = \mathrm{Rem}(b_i, p_k)$.

Step b. Now we prove that ($\dagger_1$) implies ($\dagger_2$). Suppose that there are some natural numbers $z_{1k} \leq y, \ldots, z_{\nu k} \leq y$ such that

$$P(x, y, k, z_{1k}, \ldots, z_{\nu k}) = 0, \quad \text{for all } k \leq x. \tag{$*_4$}$$

Since there are finitely many tuples of natural numbers $(k, z_1, \ldots, z_\nu)$ such that $k \leq x$ and $z_i \leq y$ for $i = 1, \ldots, \nu$, we can find a polynomial $Q(x, y)$ satisfying ($*_1$). For example, we can choose $Q(x, y) = 2x + y + 2 + C$, for $C \geq 0$ large enough. By Claim 2, since the distinct factors $\frac{Q!}{i+1} - 1$ and $\frac{Q!}{j+1} - 1$ ($i, j \leq x$) are relatively prime, by the Chinese remainder theorem (Theorem 7.4) there exist $b_1, \ldots, b_\nu < \binom{Q!-1}{x+1}$ such that

$$b_i \cong z_{ik} \left(\mathrm{mod}\, \frac{Q!}{k+1} - 1 \right), \quad k \leq x. \tag{$*_5$}$$

Since $z_{ik} \leq y$, one of the factors in the product $z_{ik}(z_{ik} - 1) \cdots (z_{ik} - y)$ is zero, so ($*_5$) implies that

$$b_i(b_i - 1) \cdots (b_i - y) \cong 0 \left(\mathrm{mod}\, \frac{Q!}{k+1} - 1 \right), \quad 1 \leq i \leq \nu. \tag{$*_6$}$$

By Claim 2, since the divisors $\frac{Q!}{k+1} - 1$ are pairwise relatively prime, their product $\binom{Q!-1}{x+1}$ divides $b_i(b_i - 1) \cdots (b_i - y)$, that is,

$$b_i(b_i - 1) \cdots (b_i - y) \equiv 0 \left(\mathrm{mod} \binom{Q! - 1}{x + 1} \right), \quad 1 \leq i \leq \nu, x \leq k.$$

By Claim 1 and ($*_1$) , since all the primes dividing $\binom{Q!-1}{x+1}$ ($k \leq x$) are greater than $Q > y+1$, we deduce that

$$\binom{b_i}{y+1} \equiv 0 \left(\mathrm{mod} \binom{Q! - 1}{x + 1} \right), \quad 1 \leq i \leq \nu. \tag{$*_7$}$$

Finally, since

$$Q! - 1 - k = (k + 1) \left(\frac{Q!}{k+1} - 1 \right),$$

we have

$$Q! - 1 \equiv k \left(\mathrm{mod}\, \frac{Q!}{k+1} - 1 \right),$$

so by ($*_5$), we have

$$P(x, y, Q! - 1, b_1, \ldots, b_\nu) \equiv P(x, y, k, z_{1k}, \ldots, z_{\nu k}) \left(\mathrm{mod}\, \frac{Q!}{k+1} - 1 \right). \tag{$*_8$}$$

Since by hypothesis ($*_4$), $P(x, y, k, z_{1k}, \ldots, z_{\nu k}) = 0$, and the moduli $\frac{Q!}{k+1} - 1$ are pairwise relatively prime, we conclude that

$$P(x, y, Q! - 1, b_1, \ldots, b_\nu) \equiv 0 \left(\mathrm{mod} \binom{Q! - 1}{x + 1} \right). \tag{$*_9$}$$

But ($*_7$) and ($*_9$) are the conjuncts in ($\dagger_2$) of our proposition, and this finishes the proof. □

Since by Step 2 the factorial function and the binomial coefficient functions are Diophantine, and since the divisibility relation $n \equiv 0 \pmod{m}$ is Diophantine (since $n \equiv 0 \pmod{m}$ iff $(\exists k)(n = km)$), the right-hand side $(\dagger_2)$ in Proposition 7.11 is Diophantine. This proves the hard part of Theorem 7.10, namely that applying bounded universal quantification to a Diophantine predicate yields a Diophantine predicate.

Davis et al. [9] (Section 4) show how Theorem 7.10 can be used to construct a Diophantine polynomial F with one parameter a such that the equation $F = 0$ *has a solution* for any fixed $a > 0$ iff some planar graph *cannot* be colored with a colors. The positive solution of the four color conjecture implies that the equation $F = 0$ has no solution for $a = 4$ (for sure, it has no solution for $a = 5$).

We have completed the hard work and the next step is relatively simple in comparison.

Step 4. To prove that every (total) computable function is Diophantine, we simply have to prove that the base functions are Diophantine and that the Diophantine functions are closed under (extended) composition, primitive recursion, and minimization (yielding total functions). Since the class of computable functions is the smallest class with these properties, it is contained in the class of Diophantine functions.

(1) The zero function $y = Z(x)$ is defined by the Diophantine equation

$$y = 0.$$

The successor function $y = \mathbf{Succ}(x) = x + 1$ is defined by the Diophantine equation

$$y = x + 1.$$

The projection function $y = P_i^n(x_1, \ldots, x_n)$ is defined by the Diophantine equation

$$y = x_i.$$

(2) Suppose that the m functions $g_i \colon \mathbb{N}^n \to \mathbb{N}$ are Diophantine and that $f \colon \mathbb{N}^m \to \mathbb{N}$ is also Diophantine. This means that each g_i has a Diophantine definition

$$(\exists \overline{y}_i)(P_i(x_1, \ldots, x_n, z_i, \overline{y}_i) = 0)$$

which holds iff $z_i = g_i(x_1, \ldots, x_n)$, where $\overline{y}_i$ denotes a sequence of parameters, and f has a Diophantine definition

$$(\exists \overline{t})(Q(u_1, \ldots, u_m, v, \overline{t}) = 0)$$

which holds iff $v = f(u_1, \ldots, v_m)$, where $\overline{t}$ denotes a sequence of parameters. By renaming the parameters we may assume that they are disjoint and also disjoint from the variables $\overline{z}$. Then the Diophantine definition

$$\begin{aligned}(\exists \overline{z})(\exists \overline{y}_1) \cdots (\exists \overline{y}_m)(\exists \overline{t})[(P_1(x_1, \ldots, x_n, z_1, \overline{y}_1) = 0) \land \cdots \land (P_m(x_1, \ldots, x_n, z_m, \overline{y}_m) = 0)\\ \land (Q(z_1, \ldots, z_m, v, \overline{t}) = 0)]\end{aligned}$$

holds iff

$$v = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)).$$

We can use the squaring trick to convert the conjunction of equations into a single equation. This proves closure under composition.

(3) Suppose $g\colon \mathbb{N}^m \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+2} \to \mathbb{N}$ are Diophantine. We wish to define $f\colon \mathbb{N}^{m+1} \to \mathbb{N}$ by primitive recursion by

$$\begin{aligned} f(0, x_1, \ldots, x_m) &= g(x_1, \ldots, x_m) \\ f(x + 1, x_1, \ldots, x_m) &= h(x, f(x, x_1, \ldots, x_m), x_1, \ldots, x_m). \end{aligned}$$

This is achieved using the sequence number theorem and the bounded quantifier theorem as follows. Assume that g has a Diophantine definition

$$(\exists \overline{s})(P(x_1, \ldots, x_m, v, \overline{s}) = 0)$$

which holds iff $v = g(x_1, \ldots, x_m)$, where $\overline{s}$ denotes a sequence of parameters, and h has a Diophantine definition

$$(\exists \overline{z})(Q(t_1, t_2, x_1, \ldots, x_m, w, \overline{z}) = 0)$$

which holds iff $w = h(t_1, t_2, x_1, \ldots, x_m)$, where $\overline{z}$ denotes a sequence of parameters. We rename variables so that $\overline{s}$ and $\overline{z}$ are disjoint. Theorem 7.7 shows that S is Diophantine, so we claim that the Diophantine definition

$$\begin{aligned} \exists u\, [&\exists v\, ((v = S(1, u)) \wedge (\exists \overline{s})(P(x_1, \ldots, x_m, v, \overline{s}) = 0)) \\ &\wedge (\forall t \leq x)[(t = x) \vee \exists w((w = S(t + 2, u)) \\ &\wedge (\exists \overline{z})(Q(t, S(t + 1, u), x_1, \ldots, x_m, w, \overline{z}) = 0))] \\ &\wedge (y = S(x + 1, u))] \end{aligned}$$

holds iff

$$y = f(x, x_1, \ldots, x_m).$$

We used the fact that the Diophantine predicates are closed under conjunction, disjunction, existential quantification, and composition. The equations $v = S(1, u), w = S(t + 2, u)$ and $y = S(x + 1, u)$ should be replaced by the Diophantine definition of S from Theorem 7.7, and the equation $Q(t, S(t + 1, u), x_1, \ldots, x_m, w, \overline{z}) = 0$ involves a composition so it should also use the Diophantine definition of S. We leave the details and the verification that this works to the reader. The idea is that u is used to record the values $f(0, x_1, \ldots, x_m), \ldots, f(x, x_1, \ldots, x_m)$ as $S(1, u), \ldots, S(x + 1, u)$. Since in Theorem 7.7 the index i used to index sequences starts from 1 and not 0, as t ranges from 0 to x have to use the index $t + 1$ which ranges from 1 to $x + 1$. This is also the reason why we have to compute $S(t + 2, u) = f(t + 1, x_1 \ldots, x_m)$ from $S(t + 1, u) = f(t, x_1, \ldots, x_m)$.

Since $A \implies B$ is logically equivalent to $\neg A \vee B$, the formula

$$(\forall t \leq x)[(t = x) \vee \exists w((w = S(t+2, u)) \wedge (\exists \overline{z})(Q(t, S(t+1, u), x_1, \ldots, x_m, w, \overline{z}) = 0))]$$

asserts that for all t such that $0 \leq t \leq x$, if $t \neq x$, so in fact if $0 \leq t < x$, then

$$[\exists w((w = S(t+2, u)) \wedge (\exists \overline{z})(Q(t, S(t+1, u), x_1, \ldots, x_m, w, \overline{z}) = 0))]$$

holds. Since $S(t+1, u) = f(t, x_1, \ldots, x_m)$, the Diophantine definition

$$(\exists \overline{z})(Q(t, S(t+1, u), x_1, \ldots, x_m, w, \overline{z}) = 0))]$$

computes $w = h(t, f(t, x_1, \ldots, x_m), x_1, \ldots, x_m) = f(t+1, x_1 \ldots, x_m)$, which is saved in $S(t+2, u)$.

(4) Assume that $g \colon \mathbb{N}^{m+1} \to \mathbb{N}$ is Diophantine and that for all $(a_1, \ldots, a_m) \in \mathbb{N}^m$ there is some $a \in \mathbb{N}$ such that $g(a, a_1, \ldots, a_m) = 0$. We wish to show that the function $f \colon \mathbb{N}^m \to \mathbb{N}$ given by minimization as

$$f(a_1, \ldots, a_m) = \min x\, (g(x, a_1, \ldots a_m) = 0)$$

is also Diophantine. Assume that g has a Diophantine definition

$$(\exists \overline{s})(P(x, x_1, \ldots, x_m, z, \overline{s}) = 0)$$

which holds iff $z = g(x, x_1, \ldots, x_m)$, where $\overline{s}$ denotes a sequence of parameters. We claim that the Diophantine definition

$$\begin{aligned}&(\exists \overline{s})(P(y, x_1, \ldots, x_m, 0, \overline{s}) = 0)\\ &\qquad \wedge [(\forall t \leq y)[(t = y) \vee \exists z(\exists \overline{u})\, ((P(t, x_1, \ldots, x_m, z, \overline{u}) = 0) \wedge (z > 0)]]\end{aligned}$$

holds if

$$y = f(a_1, \ldots, a_m) = \min x\, (g(x, a_1, \ldots a_m) = 0).$$

The predicate

$$(\exists \overline{s})(P(y, x_1, \ldots, , x_m, 0, \overline{s}) = 0)$$

asserts that $g(y, x_1, \ldots, x_m) = 0$, and the predicate

$$(\forall t \leq y)[(t = y) \vee \exists z(\exists \overline{u})\, ((P(t, x_1, \ldots, x_m, z, \overline{u}) = 0) \wedge (z > 0)]$$

asserts that $g(t, x_1, \ldots, x_m) > 0$ for all $t < y$, so y is indeed the smallest number for which $g(y, x_1, \ldots, x_m) = 0$.

Therefore we have finally proven that every (total) computable function is Diophantine.

Step 5. Every listable set is Diophantine.

By Proposition 7.6, a set $S \subseteq \mathbb{N}$ is listable iff there are two (total) computable functions $f, g\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that

$$S = \{a \in \mathbb{N} \mid (\exists x)(f(a, x) = g(a, x))\}.$$

But then

$$a \in S \quad \text{iff} \quad \exists x \exists z((z = f(a, x)) \wedge (z = g(a, x))).$$

By Step 4, the computable functions f and g have Diophantine definitions

$$(\exists \overline{u})(P(y, x, z, \overline{u}) = 0)$$

iff $z = f(y, x)$ and

$$(\exists \overline{v})(Q(y, x, z, \overline{v}) = 0)$$

iff $z = g(y, x)$, so $a \in S$ has the Diophantine definition

$$\exists x \exists z[(\exists \overline{u})(P(a, x, z, \overline{u}) = 0) \wedge (\exists \overline{v})(Q(a, x, z, \overline{v}) = 0)].$$

This is the famous result that we were seeking. □

Using some results from the theory of computation it is now easy to deduce that Hilbert's tenth problem is undecidable. To achieve this, recall that there are listable sets that are not computable. For example, it is shown in Lemma 5.11 that $K = \{x \in \mathbb{N} \mid \varphi_x(x) \text{ is defined}\}$ is listable but not computable. Since K is listable, by Theorem 7.8, it is defined by some Diophantine equation

$$P(a, x_1, \ldots, x_n) = 0,$$

which means that

$$K = \{a \in \mathbb{N} \mid (\exists x_1 \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\}.$$

We have the following strong form of the undecidability of Hilbert's tenth problem, in the sense that it shows that Hilbert's tenth problem is already undecidable for a fixed Diophantine equation in one parameter.

Theorem 7.12. *There is no algorithm which takes as input the polynomial $P(a, x_1, \ldots, x_n)$ defining K and any natural number $a \in \mathbb{N}$ and decides whether*

$$P(a, x_1, \ldots, x_n) = 0.$$

Consequently, Hilbert's tenth problem is undecidable.

Proof. If there was such an algorithm, then K would be decidable, a contradiction.

Any algorithm for solving Hilbert's tenth problem could be used to decide whether or not $P(a, x_1, \ldots, x_n) = 0$, but we just showed that there is no such algorithm. □

It is an open problem whether Hilbert's tenth problem is undecidable if we allow *rational solutions* (that is, $x_1, \ldots, x_n \in \mathbb{Q}$).

Alexandra Shlapentokh proved that various extensions of Hilbert's tenth problem are undecidable. These results deal with some algebraic number theory beyond the scope of these notes. Incidentally, Alexandra was an undergraduate at Penn, and she worked on a logic project for me (finding a Gentzen system for a subset of temporal logic).

Having now settled once and for all the undecidability of Hilbert's tenth problem, we now briefly explore some interesting consequences of Theorem 7.8.

The fact that a set is listable if and only if it is Diophantine also holds for m-ary relations.

7.6 The DPRM For Relations

Definition 7.9. A relation $R \subseteq \mathbb{N}^m$ $(m \geq 2)$ is *listable* if the set

$$\widehat{R} = \{\langle x_1, \ldots, x_m \rangle_m \in \mathbb{N} \mid (x_1, \ldots, x_m) \in R\}$$

is listable, where $\langle x_1, \ldots, x_m \rangle_m$ is the extended pairing function of Definition 4.2.

Proposition 7.6 is easily generalized to the following characterization of listable relations.

Proposition 7.13. *A relation $R \subseteq \mathbb{N}^m$ $(m \geq 2)$ is listable iff there are two (total) computable functions $f, g \colon \mathbb{N}^{m+1} \to \mathbb{N}$ such that*

$$R = \{(a_1, \ldots a_m) \in \mathbb{N}^m \mid (\exists x)(f(a_1, \ldots, a_m, x) = g(a_1, \ldots, a_m, x))\}.$$

Proof. If $R = \emptyset$, then we let f be the constant function equal to 0 and g be the constant function equal to 1. If $R \neq \emptyset$ is listable, then by Definition 5.6 (see also Proposition 5.9), there is a total computable function $h \colon \mathbb{N} \to \mathbb{N}$ such that $\widehat{R}$ is equal to the range of h. If we let f be given by $f(a_1, \ldots, a_m, x) = \langle a_1, \ldots, a_m \rangle_m$ (which is primitive recursive) and $g(a_1, \ldots, a_m, x) = h(x)$ for all $a, x \in \mathbb{N}$, then

$$\begin{aligned} R &= \{(a_1, \ldots, a_m) \in \mathbb{N}^m \mid (\exists x)(\langle a_1, \ldots, a_m \rangle_m = h(x))\} \\ &= \{(a_1, \ldots, a_m) \in \mathbb{N}^m \mid (\exists x)(f(a_1, \ldots, a_m, x) = g(a_1, \ldots, a_m, x))\}. \end{aligned}$$

Conversely, assume that

$$R = \{(a_1, \ldots a_m) \in \mathbb{N}^m \mid (\exists x)(f(a_1, \ldots, a_m, x) = g(a_1, \ldots, a_m, x))\}$$

with f, g total computable. Using the uniform projection function Π of Definition 4.3, which is primitive recursive, we have

$$\widehat{R} = \{a \in \mathbb{N} \mid (\exists x)(f(\Pi(1, m, a), \ldots, \Pi(m, m, a), x) = g(\Pi(1, m, a), \ldots, \Pi(m, m, a), x))\}.$$

As a composition of (total) computable functions, $\widehat{f}$ and $\widehat{g}$ given by

$$\begin{aligned}\widehat{f}(a, x) &= f(\Pi(1, m, a), \ldots, \Pi(m, m, a), x)\\ \widehat{g}(a, x) &= g(\Pi(1, m, a), \ldots, \Pi(m, m, a), x)\end{aligned}$$

are total computable, so by Proposition 7.6 the set $\widehat{R}$ is listable. □

Using Proposition 7.13 it is easy to generalize the DPRM to relations.

Theorem 7.14. *(DPRM for Relations) A relation $R \subseteq \mathbb{N}^m$ $(m \geq 2)$ is listable if and only if it is Diophantine.*

Proof. Assume that R is Diophantine so that there is a polynomial $P(x_1, \ldots, x_m, y_1, \ldots, y_p)$ such that

$$(a_1, \ldots, a_m) \in R \quad \text{iff} \quad (\exists b_1, \ldots, b_p)(P(a_1, \ldots, a_m, b_1, \ldots, b_p) = 0).$$

By grouping monomials with the same sign together we can write

$$P(x_1, \ldots, x_m, y_1, \ldots, y_p) = Q(x_1, \ldots, x_m, y_1, \ldots, y_p) - R(x_1, \ldots, x_m, y_1, \ldots, y_p),$$

where $Q(x_1, \ldots, x_m, y_1, \ldots, y_p)$ and $R(x_1, \ldots, x_m, y_1, \ldots, y_p)$ have *positive integer coefficients.* Then we have

$$(a_1, \ldots, a_m) \in R \quad \text{iff} \quad (\exists b_1, \ldots, b_p)(Q(a_1, \ldots, a_m, b_1, \ldots, b_p) = R(a_1, \ldots, a_m, b_1, \ldots, b_p)),$$

and by the sequence number theorem we can find $u \in \mathbb{N}$ such that $b_1 = S(1, u), \ldots, b_p = S(p, u)$, so

$$\begin{aligned}(a_1, \ldots, a_m) \in R \quad \text{iff} \quad &(\exists u)(Q(a_1, \ldots, a_m, S(1, u), \ldots, S(p, u))\\ &= R(a_1, \ldots, a_m, S(1, u), \ldots, S(p, u))).\end{aligned}$$

Since Q and R compute primitive recursive functions and S is primitive recursive, by Proposition 7.13, R is listable.

Conversely, assume that R is listable. By Proposition 7.13, a relation $R \subseteq \mathbb{N}^m$ is listable iff there are two (total) computable functions $f, g \colon \mathbb{N}^{m+1} \to \mathbb{N}$ such that

$$R = \{(a_1, \ldots a_m) \in \mathbb{N}^m \mid (\exists x)(f(a_1, \ldots, a_m, x) = g(a_1, \ldots, a_m, x))\}.$$

But then

$$(a_1, \ldots, a_m) \in R \quad \text{iff} \quad \exists x \exists z((z = f(a_1, \ldots, a_m, x)) \land (z = g(a_1, \ldots, a_m, x))).$$

By Theorem 7.8, the computable functions f and g have Diophantine definitions and we finish the proof as in Step 5 of Theorem 7.8 to obtain a Diophantine definition of R. □

7.7 Some Applications of the DPRM Theorem

The first application of the DRPM theorem is a particularly striking way of defining the listable subsets of $\mathbb{N}$ as the nonnegative ranges of polynomials with integer coefficients. This result is due to Hilary Putnam.

Theorem 7.15. *For every listable subset S of $\mathbb{N}$, there is some polynomial $Q(x, x_1, \ldots, x_n)$ with integer coefficients such that*

$$S = \{Q(a, b_1, \ldots, b_n) \mid Q(a, b_1, \ldots, b_n) \in \mathbb{N},\ a, b_1, \ldots, b_n \in \mathbb{N}\}.$$

Proof. By the DPRM theorem (Theorem 7.8), there is some polynomial $P(x, x_1, \ldots, x_n)$ with integer coefficients such that

$$S = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_n)(P(a, x_1, \ldots, x_n) = 0)\}.$$

Let $Q(x, x_1, \ldots, x_n)$ be given by

$$Q(x, x_1, \ldots, x_n) = (x + 1)(1 - P^2(x, x_1, \ldots, x_n)) - 1.$$

We claim that Q satisfies the statement of the theorem. If $a \in S$, then $P(a, b_1, \ldots, b_n) = 0$ for some $b_1, \ldots, b_n \in \mathbb{N}$, so

$$Q(a, b_1, \ldots, b_n) = (a + 1)(1 - 0) - 1 = a.$$

This shows that all $a \in S$ show up the the nonnegative range of Q. Conversely, assume that $Q(a, b_1, \ldots, b_n) \geq 0$ for some $a, b_1, \ldots, b_n \in \mathbb{N}$. Then by definition of Q we must have

$$(a + 1)(1 - P^2(a, b_1, \ldots, b_n)) - 1 \geq 0,$$

that is,

$$(a + 1)(1 - P^2(a, b_1, \ldots, b_n)) \geq 1,$$

and since $a \in \mathbb{N}$, this implies that $P^2(a, b_1, \ldots, b_n) < 1$, but since P is a polynomial with integer coefficients and $a, b_1, \ldots, b_n \in \mathbb{N}$, the expression $P^2(a, b_1, \ldots, b_n)$ must be a nonnegative integer, so we must have

$$P(a, b_1, \ldots, b_n) = 0,$$

which shows that $a \in S$. $\square$

Remark: It should be noted that in general, the polynomials Q arising in Theorem 7.15 may take on negative integer values, and to obtain all listable sets, we must restrict ourself to their nonnegative range.

As an example, the set S_3 of natural numbers that are congruent to 1 or 2 modulo 3 is given by

$$S_3 = \{a \in \mathbb{N} \mid (\exists y)(3y + 1 - a^2 = 0)\}.$$

so by Theorem 7.15, S_3 is the nonnegative range of the polynomial

$$\begin{aligned} Q(x,y) &= (x+1)(1-(3y+1-x^2)^2)) - 1 \\ &= -(x+1)((3y-x^2)^2 + 2(3y-x^2))) - 1 \\ &= (x+1)(x^2-3y)(2-(x^2-3y)) - 1. \end{aligned}$$

Observe that $Q(x,y)$ takes on negative values. For example, $Q(0,0) = -1$. Also, in order for $Q(x,y)$ to be nonnegative, $(x^2-3y)(2-(x^2-3y))$ must be positive, but this can only happen if $x^2 - 3y = 1$, that is, $x^2 = 3y+1$, which is the original equation defining S_3.

There is no miracle. The nonnegativity of $Q(x, x_1, \ldots, x_n)$ must subsume the solvability of the equation $P(x, x_1, \ldots, x_n) = 0$.

A particularly interesting listable set is the set of primes. By Theorem 7.15, in theory, the set of primes is the positive range of some polynomial with integer coefficients.

Remarkably, some explicit polynomials have been found. This is a nontrivial task. In particular, the process involves showing that the exponential function is definable, which was the stumbling block to the completion of the DPRM theorem for many years.

We now explain how to express primality in terms of equations, provided that we allow free uses of the exponential function. The key idea is to express primality using the Bezout identity (Proposition 7.2). We will obtain a set of equations involving the function factorial ($s!$). The factorial function can be equationally defined using the binomial coefficient $\binom{t}{s}$, which in turn can be defined equationally in terms of the exponential function. This is as far as we will go, since proving that the exponential function is Diophantine definable is a long and complicated process.

Recall that Proposition 7.2 (the Bezout identity) implies that for any two integers $m, n \in \mathbb{Z}$, if $d = gcd(m,n)$, then there are some integers $a, b \in \mathbb{Z}$ such that

$$am + bn = d.$$

If both $m, n > 0$, then $d > 0$, so if we write $m = q_1 d$ and $n = q_2 d$ (with $q_1, q_2 \in \mathbb{N}$), then for any $k \in \mathbb{Z}$ we also have

$$(a+kq_2)m+(b-kq_1)n = am+bn+kq_2m-kq_1n = am+bn+kq_2q_1d-kq_1q_2d = am+bn = d.$$

As a consequence, if $a < 0$, in which case we must have $b > 0$, we can pick $k \in \mathbb{Z}$ large enough so that $a + kq_2 \geq 0$ and $b - kq_1 \leq 0$, that is $kq_2 \geq -a$ and $kq_1 \geq b$, so $k \geq \max(-a, b)$ will do. Therefore, if $m > 0$ and $n > 0$, we may assume that $a \geq 0$ and $b \leq 0$, or equivalently that the equation

$$am - bn = d \qquad (*_B)$$

holds for some $a, b \in \mathbb{N}$.

Remark: By picking $k > \max(-a, b)$ we can ensure that $a > 0$ and $b < 0$ in $am + bn = d$, but we don't need this stronger condition. Also, if $m = 0$ and $n > 0$, or $m > 0$ and $n = 0$, the condition $(*_B)$ needs to be replaced by

$$am - bn = d \quad \text{or} \quad bn - am = d,$$

for some $m, n \in \mathbb{N}$.

Now $m, n > 0$ are relatively prime iff $gcd(d) = 1$, which by the Bezout identity and the above discussion is equivalent to the fact that the equation

$$am - bn = 1$$

has a solution for some $m, n \in \mathbb{N}$. We can now apply this fact to assert that a number p is prime.

Observe that by the Bezout identity, if $p = s + 1$ and $q = s!$, then we can assert that p and q are relatively prime ($\gcd(p, q) = 1$) as the fact that the Diophantine equation

$$ap - bq = 1$$

is satisfied for some $a, b \in \mathbb{N}$. Then $p \in \mathbb{N}$ is prime iff $p \geq 2$ and p has no divisor h such that $1 < h < p$ iff $p \geq 2$ and $gcd(p, q) = gcd(p, (p - 1)!) = 1$. We leave the details an an exercise.

Then it is not hard to see that $p \in \mathbb{N}$ is prime iff the following set (P) of equations has a solution for $a, b, s, r, q \in \mathbb{N}$:

$$\begin{aligned} p &= s + 1 \\ p &= r + 2 \\ q &= s! \\ ap - bq &= 1. \end{aligned} \tag{P}$$

The problem with the above is that the equation $q = s!$ is not Diophantine. The next step is to show that the factorial function is Diophantine, and this involves a lot of work. One way to proceed is to show that the above system is equivalent to a system allowing the use of the exponential function $exp(m, n) = m^n$.

The first trick is express the factorial function in terms of the exponential function and the binomial coefficient. Indeed, for $t \geq s \in \mathbb{N}$ (with $s \geq 1$ fixed), since

$$\binom{t}{s} = \frac{t!}{s!(t - s)!} = \frac{t(t - 1) \cdots (t - s + 1)}{s!},$$

we have

$$s! = \frac{t(t - 1) \cdots (t - s + 1)}{\binom{t}{s}}.$$

For $s = 1$ we have

$$s! = \frac{t^1}{\binom{t}{1}}$$

since $1! = 1$ and

$$\frac{t^1}{\binom{t}{1}} = \frac{t}{t} = 1.$$

For $s \geq 2$, if we replace every term in the product in the numerator by t, we deduce that

$$\begin{aligned} s! \leq \frac{t^s}{\binom{t}{s}} &= \frac{s!t^s}{t(t-1)\cdots(t-s+1)} \\ &= s!\left(1 + \frac{1}{t-1}\right)\cdots\left(1 + \frac{s-1}{t-s+1}\right). \end{aligned}$$

Observe that if let t go to infinity, then for $k = 1, \ldots, s-1$

$$\lim_{t \mapsto \infty} \left(1 + \frac{k}{t-k}\right) = 1,$$

which implies that

$$\lim_{t \mapsto \infty} \left(1 + \frac{1}{t-1}\right)\cdots\left(1 + \frac{s-1}{t-s+1}\right) = 1,$$

and so

$$\lim_{t \mapsto \infty} \frac{t^s}{\binom{t}{s}} = s!.$$

More precisely, it is not hard to see that if $t \geq 2s^{s+2}$, then

$$\left(1 + \frac{1}{t-1}\right)\cdots\left(1 + \frac{s-1}{t-s+1}\right) \leq 1 + \frac{1}{s^{s-1}},$$

with $s^{s-1} > s!$ (since $s \geq 2$), and so

$$s! = \left\lfloor \frac{t^s}{\binom{t}{s}} \right\rfloor = q, \qquad (*_!)$$

where q the largest natural number (the floor) such that

$$q \leq \frac{t^s}{\binom{t}{s}} < q + 1.$$

As we already know, the above formula also holds for $s = 1$. But then after some thinking we can show that $q = s!$ is equivalent to the following equations (where all the variables range over $\mathbb{N}$):

$$t = 2s^{s+2} \tag{1}$$
$$t^s = qu + w \tag{2}$$
$$u = \binom{t}{s} \tag{3}$$
$$u = w + x + 1. \tag{4}$$

For $s = 0$, since $0^2 = 0$, the first equation yields $t = 0$, and then by the third equation, $u = \binom{0}{0} = 1$. The fourth equation forces $w = x = 0$. Since $0^0 = 1$, the second equation yields $q = 1$, which is indeed $0!$.

Let us now assume that $s \geq 1$. From (2) and (3) we have

$$q = \frac{t^s}{\binom{t}{s}} - \frac{w}{\binom{t}{s}},$$

so

$$q \leq \frac{t^s}{\binom{t}{s}}.$$

By (2) and (4) we have

$$t^s = qu + w = qu + u - x - 1 = (q+1)u - x - 1,$$

so using (3) we get

$$q + 1 = \frac{t^s}{\binom{t}{s}} + \frac{x+1}{\binom{t}{s}},$$

which implies that

$$\frac{t^s}{\binom{t}{s}} < q + 1,$$

and since by (1), $t = 2s^{s+2}$, we get

$$q = \left\lfloor \frac{t^s}{\binom{t}{s}} \right\rfloor = s!$$

This astute maneuver shows that $s!$ is equationally definable if we allow the exponential function $exp(m, n) = m^n$ and the binomial coefficient $\binom{t}{s}$.

Actually, another trick shows that the binomial coefficients are definable in terms of the exponential function too. Since

$$(y+1)^t = \sum_{i=0}^{t} \binom{t}{i} y^i,$$

if y is large enough, in fact $y > 2^t$ will do, then it turns out that the binomial coefficients $\binom{t}{i}$ are the digits in the expansion of $(y+1)^t$ in base y.

We claim that $u = \binom{t}{s}$ is equivalent to the following system of equations (where all the variables range over $\mathbb{N}$):

$$\begin{aligned} y &= 2^t + 1 && (5)\\ z &= y + 1 && (6)\\ z^t &= \ell y^{s+1} + uy^s + m && (7)\\ u + v &= 2^t && (8)\\ m + n + 1 &= y^s. && (9) \end{aligned}$$

If $t = 0$, then Equations (5) and (6) yield $y = 2, z = 3$. Equation (7) yields

$$1 = \ell 2^{s+1} + u2^s + m.$$

Since $y^{s+1} = 2^{s+1} \geq 2$, we must have $\ell = 0$.

If $s = 0$, then $y^s = 2^0 = 1$ and Equation (9) yields $m + n + 1 = 1$, so $m = n = 0$. We have $2^t = 2^0 = 1$, so Equation (7) implies that $u = 1$, and then $v = 0$. We get $\binom{0}{0} = u = 1$, as desired.

If $s \geq 1$, then $y^s = 2^s \geq 2$, so we must have $u = 0$. Then Equation (9) implies that $m = 1$, and then $n = 2^s - 1$ and $v = 1$. We get $\binom{0}{s} = u = 0$, as desired.

If $t \geq 1$ and $s > t$, we claim that $y > 2^t$ implies that $(y+1)^t < y^s$. This is because

$$(y+1)^t < (y+y)^t = (2y)^t = 2^t y^t < y^{t+1}.$$

Assume that $t \geq 1$ and $s > t$. Since $z = y + 1$ and the equation $y = 2^t + 1$ implies that $y > 2^t$, the equation

$$(y+1)^t = z^t = \ell y^{s+1} + uy^s + m$$

and the fact that $(y+1)^t < y^s$ implies that $\ell = u = 0$. Then $m = (y+1)^t$, $v = 2^t$, and $n = y^s - (y+1)^t - 1$, which is a natural number since $(y+1)^t < y^s$. Therefore $\binom{t}{s} = u = 0$ if $1 \leq t < s$, as desired.

Finally, assume that $t \geq 1$ and $0 \leq s \leq t$. Using the binomial formula, we have

$$\begin{aligned} (y+1)^t &= \sum_{k=0}^{t} \binom{t}{t-k} y^{t-k}\\ &= \sum_{k=0}^{t-s-1} \binom{t}{t-k} y^{t-k} + \binom{t}{s} y^s + \sum_{k=t-s+1}^{t} \binom{t}{t-k} y^{t-k}\\ &= \left(\sum_{k=0}^{t-s-1} \binom{t}{t-k} y^{t-s-1-k} \right) y^{s+1} + \binom{t}{s} y^s + \sum_{k=t-s+1}^{t} \binom{t}{t-k} y^{t-k}. \end{aligned}$$

The equation $y = 2^t + 1$ implies that $y > 2^t$, and since $\binom{t}{k} \leq 2^t < y$ (because of the well-known identity $\sum_{k=0}^{t} \binom{t}{k} = 2^t$), we deduce that $\binom{t}{s}$ is the coefficient of y^s in the representation of $(y+1)^t$ in base $y > 2^t$. Consequently, the unique solutions of the equation

$$(y+1)^t = z^t = \ell y^{s+1} + uy^s + m$$

are

$$\begin{aligned} m &= \sum_{k=t-s+1}^{t} \binom{t}{t-k} y^{t-k} \\ u &= \binom{s}{t} \\ \ell &= \sum_{k=0}^{t-s-1} \binom{t}{t-k} y^{t-s-1-k}. \end{aligned}$$

Since they appear in the representation of $(y+1)^t$ in base y, the numbers u and v satisfy the inequalities

$$\begin{aligned} m &< y^s \\ u &\leq 2^t, \end{aligned}$$

so the equations

$$\begin{aligned} u + v &= 2^t \\ m + n + 1 &= y^s \end{aligned}$$

are satisfied. Therefore, there is a unique solution $u = \binom{t}{s}$, as desired.

In summary, the binomial coefficients can be equationally defined by the Equations (5)–(9) (with $s, t \in \mathbb{N}$) and the factorial function can be equationally defined by the Equations (1)–(2) and (4)–(9). In both cases we allow the use of the exponential function. Since the equation $q = s!$ in the set (P) of four equations stated earlier can be replaced by the equations (1)–(2) and (4)–(9) we deduce that any prime p is equationally defined, provided that we allow the use of the exponential function.

The final step is to show that the exponential function can be eliminated in favor of polynomial equations. This is the hardest step which was overcome by Matyasevich by building up on results of Robinson.

We refer the interested reader to the remarkable expository paper by Davis, Matiyasevich and Robinson [9] for details. Here is a polynomial of total degree 25 in 26 variables (due to

J. Jones, D. Sato, H. Wada, D. Wiens) which produces the primes as its positive range:

$$
\begin{aligned}
&(k+2)\Big[1-([wz+h+j-q]^2+[(gk+2g+k+1)(h+j)+h-z]^2\\
&+[16(k+1)^3(k+2)(n+1)^2+1-f^2]^2\\
&+[2n+p+q+z-e]^2+[e^3(e+2)(a+1)^2+1-o^2]^2\\
&+[(a^2-1)y^2+1-x^2]^2+[16r^2y^4(a^2-1)+1-u^2]^2\\
&+[((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2]^2\\
&+[(a^2-1)l^2+1-m^2]^2+[ai+k+1-l-i]^2+[n+l+v-y]^2\\
&+[p+l(a-n-1)+b(2an+2a-n^2-2n-2)-m]^2\\
&+[q+y(a-p-1)+s(2ap+2a-p^2-2p-2)-x]^2\\
&+[z+pl(a-p)+t(2ap-p^2-1)-pm]^2)\Big].
\end{aligned}
$$

Around 2004, Nachi Gupta, an undergraduate student at Penn, and I tried to produce the prime 2 as one of the values of the positive range of the above polynomial. It turns out that this leads to values of the variables that are so large that we never succeeded!

Other interesting applications of the DPRM theorem are the re-statements of famous open problems, such as the Riemann hypothesis, as the unsolvability of certain Diophantine equations. For all this, see Davis, Matiyasevich and Robinson [9]. One may also obtain a nice variant of Gödel's incompleteness theorem.

7.8 Gödel's Incompleteness Theorem

Gödel published his famous incompleteness theorem in 1931. At the time, his result rocked the mathematical world, and certainly the community of logicians.

In order to understand why his result had such impact one needs to step back in time. In the late 1800's, Hilbert had advanced the thesis that it should be possible to completely formalize mathematics in such a way that every true statement should be provable "mechanically." In modern terminology, Hilbert believed that one could design a theorem prover that should be complete. His quest is known as *Hilbert's program*. In order to achieve his goal, Hilbert was led to investigate the notion of proof, and with some collaborators including Ackerman, Hilbert developed a significant amount of what is known as *proof theory*. When the young Gödel announced his incompleteness theorem, Hilbert's program came to an abrupt halt. Even the quest for a complete proof system for arithmetic was impossible.

It should be noted that when Gödel proved his incompleteness theorem, computability theory basically did not exist, so Gödel had to start from scratch. His proof is really a tour de force. Gödel's theorem also triggered extensive research on the notion of computability and undecidability between 1931 and 1936, the major players being Church, Gödel himself, Herbrand, Kleene, Rosser, Turing, and Post.

In this section we will give a (deceptively) short proof that relies on the DPRM and the existence of universal functions. The proof is short because the hard work lies in the proof of the DPRM!

The first step is to translate the fact that there is a universal partial computable function φ_{univ} (see Proposition 4.7), such that for all $x, y \in \mathbb{N}$, if φ_x is the xth partial computable function, then

$$\varphi_x(y) = \varphi_{univ}(x, y).$$

Also recall from Definition 5.7 that for any acceptable indexing of the partial computable functions, the listable (c.e. r.e.) sets W_x are given by

$$W_x = dom(\varphi_x), \quad x \in \mathbb{N}.$$

Since φ_{univ} is a partial computable function, it can be converted into a Diophantine equation so that we have the following result.

Theorem 7.16. *(Universal Equation Theorem) There is a Diophantine equation $U(m, a, x_1, \ldots x_\nu) = 0$ such that for every listable (c.e., r.e.) set W_m ($m \in \mathbb{N}$) we have*

$$a \in W_m \quad \textit{iff} \quad (\exists x_1, \ldots, x_\nu)(U(m, a, x_1, \ldots, x_\nu) = 0).$$

Proof. We have

$$W_m = \{a \in \mathbb{N} \mid (\exists x_1)(\varphi_{univ}(m, a) = x_1)\},$$

and since φ_{univ} is partial computable, by the DPRM (Theorem 7.8), there is Diophantine polynomial $U(m, a, x_1, \ldots, x_\nu)$ such that

$$x_1 = \varphi_{univ}(m, a) \quad \text{iff} \quad (\exists x_2, \ldots, x_\nu)(U(m, a, x_1, \ldots, x_\nu) = 0),$$

and so

$$W_m = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_\nu)(U(m, a, x_1, \ldots, x_\nu) = 0)\},$$

as claimed. □

The Diophantine equation $U(m, a, x_1, \ldots x_\nu) = 0$ is called a *universal Diophantine equation*. It is customary to denote $U(m, a, x_1, \ldots x_\nu)$ by $P_m(a, x_1, \ldots, x_\nu)$.

Gödel's incompleteness theorem applies to sets of logical (first-order) formulae of arithmetic built from the mathematical symbols $0, S, +, \cdot, <$ and the logical connectives $\wedge, \vee, \neg, \Rightarrow$ $, =, \forall, \exists$. Recall that logical equivalence, $\equiv$, is defined by

$$P \equiv Q \quad \text{iff} \quad (P \Rightarrow Q) \wedge (Q \Rightarrow P).$$

The term

$$\underbrace{S(S(\cdots(S}_{n}(0))\cdots))$$

is denoted by $S^n(0)$, and represents the natural number n.

For example,

$$\exists x(S(S(S(0))) < (S(S(0)) + x)),$$

$$\exists x\exists y\exists z((0 < x) \wedge (0 < y) \wedge (0 < z) \wedge ((x \cdot x + y \cdot y) = z \cdot z)),$$

and

$$\forall x\forall y\forall z((0 < x) \wedge (0 < y) \wedge (0 < z) \Rightarrow \neg((x \cdot x \cdot x \cdot x + y \cdot y \cdot y \cdot y) = z \cdot z \cdot z \cdot z))$$

are formulae in the language of arithmetic. All three are true. The first formula is satisfied by $x = S(S(0))$, the second by $x = S^3(0), y = S^4(0)$ and $z = S^5(0)$ (since $3^2 + 4^2 = 9 + 16 = 25 = 5^2$), and the third formula asserts a special case of Fermat's famous theorem: for every $n \geq 3$, the equation $x^n + y^n = z^n$ has *no solution* with $x, y, z \in \mathbb{N}$ and $x > 0, y > 0$, $z > 0$. The third formula corrresponds to $n = 4$. Even for this case, the proof is hard.

To be completely rigorous we should explain precisely what is a formal proof. Roughly speaking, a proof system consists of axioms and inference rule. A proof is a certain kind of tree whose nodes are labeled with formulae, and this tree is constructed in such a way that for every node some inference rule is applied. Proof systems are discussed in Chapter **??** and in more detail in Chapter **??**. The reader is invited to review this material. Such proof systems are also presented in Gallier [18, 17].

Given a polynomial $P(x_1, \ldots, x_m)$ in $\mathbb{Z}[x_1, \ldots, x_m]$, we need a way to "prove" that some natural numbers $n_1, \ldots, n_m \in \mathbb{N}$ are a solution of the Diophantine equation

$$P(x_1, \ldots, x_m) = 0,$$

which means that we need to have enough formulae of arithmetic to allow us to simplify the expression $P(n_1, \ldots, n_m)$ and check whether or not it is equal to zero.

For example, if $P(x, y) = 2x - 3y - 1$, we have the solution $x = 2$ and $y = 1$. What we do is to group all monomials with positive signs, $2x$, and all monomials with negative signs, $3y + 1$, plug in the values for x and y, simplify using the arithmetic tables for $+$ and $\cdot$, and then compare the results. If they are equal, then we proved that the equation has a solution.

In our language, $x = S^2(0)$, $2x = S^2(0) \cdot x$, and $y = S^1(0)$, $3y + 1 = S^3(0) \cdot y + S(0)$. We need to simplify the expressions

$$2x = S^2(0) \cdot S^2(0) \quad \text{and} \quad 3y + 1 = S^3(0) \cdot S(0) + S(0).$$

Using the formulae

$$\begin{aligned} S^m(0) + S^n(0) &= S^{m+n}(0) \\ S^m(0) \cdot S^n(0) &= S^{mn}(0) \\ S^m(0) < S^n(0) &\quad \text{iff} \quad m < n, \end{aligned}$$

with $m, n \in \mathbb{N}$, we simplify $S^2(0) \cdot S^2(0)$ to $S^4(0)$, $S^3(0) \cdot S(0) + S(0)$ to $S^4(0)$, and we see that the results are equal.

In general, given a polynomial $P(x_1, \ldots, x_m)$ in $\mathbb{Z}[x_1, \ldots, x_m]$, we write it as

$$P(x_1, \ldots, x_m) = P_{\mathrm{pos}}(x_1, \ldots, x_m) - P_{\mathrm{neg}}(x_1, \ldots, x_m),$$

where $P_{\mathrm{pos}}(x_1, \ldots, x_m)$ consists of the monomials with positive coefficients, and $-P_{\mathrm{neg}}(x_1, \ldots, x_m)$ consists of the monomials with negative coefficients. Next we plug in $S^{n_1}(0), \ldots, S^{n_m}(0)$ in $P_{\mathrm{pos}}(x_1, \ldots, x_m)$, and evaluate using the formulae for the addition and multiplication tables obtaining a term of the form $S^p(0)$. Similarly, we plug in $S^{n_1}(0), \ldots, S^{n_m}(0)$ in $P_{\mathrm{neg}}(x_1, \ldots, x_m)$, and evaluate using the formulae for the addition and multiplication tables obtaining a term of the form $S^q(0)$. Then, since exactly one of the formulae

$$S^p(0) = S^q(0), \quad \text{or} \quad S^p(0) < S^q(0), \quad \text{or} \quad S^q(0) < S^p(0)$$

is true, we obtain a proof that either $P(n_1, \ldots, n_m) = 0$ or $P(n_1, \ldots, n_m) \neq 0$.

A more economical way that does use not an infinite number of formulae expressing the addition and multiplication tables is to use various axiomatizations of arithmetic.

One axiomatization known as *Robinson arithmetic* (R. M. Robinson (1950)) consists of the following seven axioms:

$$\begin{aligned}
&\forall x \neg (S(x) = 0)\\
&\forall x \forall y ((S(x) = S(y)) \Rightarrow (x = y))\\
&\forall y ((y = 0) \vee \exists x (S(x) = y))\\
&\forall x (x + 0 = x)\\
&\forall x \forall y (x + S(y) = S(x + y))\\
&\forall x (x \cdot 0 = 0)\\
&\forall x \forall y (x \cdot S(y) = x \cdot y + x).
\end{aligned}$$

Peano arithmetic is obtained from Robinson arithmetic by adding a rule schema expressing induction:

$$[\varphi(0) \wedge \forall n (\varphi(n) \Rightarrow \varphi(n + 1))] \Rightarrow \forall m \varphi(m),$$

where $\varphi(x)$ is any (first-order) formula of arithmetic. To deal with $<$, we also have the axiom

$$\forall x \forall y (x < y \equiv \exists z (S(z) + x = y)).$$

It is easy to prove that the formulae

$$\begin{aligned}
S^m(0) + S^n(0) &= S^{m+n}(0)\\
S^m(0) \cdot S^n(0) &= S^{mn}(0)\\
S^m(0) &< S^n(0) \quad \text{iff} \quad m < n,
\end{aligned}$$

are provable in Robinson arithmetic, and thus in Peano arithmetic (with $m, n \in \mathbb{N}$).

Gödel's incompleteness applies to sets $\mathcal{A}$ of formulae of arithmetic that are "nice" and strong enough. A set $\mathcal{A}$ of formulae is nice if it is listable and *consistent* (see Definition 6.3), which means that it is impossible to prove φ and $\neg\varphi$ from $\mathcal{A}$ for some formula φ. In other words, $\mathcal{A}$ is free of contradictions.

Since the axioms of Peano arithmetic are obviously true statements about $\mathbb{N}$ and since the induction principle holds for $\mathbb{N}$, the set of all formulae provable in Robinson arithmetic and in Peano arithmetic is consistent.

As in Section 6.3, it is possible to assign a Gödel number $\#(A)$ to every first-order sentence A in the language of arithmetic; see Enderton [11] (Chapter III) or Kleene I.M. [24] (Chapter X). With a slight abuse of notation, we say that a set T is sentences of arithmetic is computable (*resp.* listable) iff the set of Gödel numbers $\#(A)$ of sentences A in T is computable (*resp.* listable). It can be shown that the set of all formulae provable in Robinson arithmetic and in Peano arithmetic are listable.

Here is a rather strong version of Gödel's incompleteness from Davis, Matiyasevich and Robinson [9].

Theorem 7.17. *(Gödel's Incompleteness Theorem) Let $\mathcal{A}$ be a set of formulae of arithmetic satisfying the following properties:*

(a) The set $\mathcal{A}$ is consistent.

(b) The set $\mathcal{A}$ is listable (c.e., r.e.)

(c) The set $\mathcal{A}$ is strong enough to prove all formulae

$$\begin{aligned} S^m(0) + S^n(0) &= S^{m+n}(0) \\ S^m(0) \cdot S^n(0) &= S^{mn}(0) \\ S^m(0) < S^n(0) \quad &\textit{iff} \quad m < n, \end{aligned}$$

for all $m, n \in \mathbb{N}$.

Then we can construct a Diophantine equation $F(x_1, \ldots, x_\nu) = 0$ corresponding to $\mathcal{A}$ such that $F(x_1, \ldots, x_\nu) = 0$ has **no** *solution with $x_1, \ldots, x_\nu \in \mathbb{N}$ but the formula*

$$\neg(\exists x_1, \ldots, x_\nu)(F(x_1, \ldots, x_\nu) = 0) \tag{$*$}$$

is **not** *provable from $\mathcal{A}$. In other words, there is a true statement of arithmetic not provable from $\mathcal{A}$; that is, $\mathcal{A}$ is incomplete.*

Proof. Define the subset $A \subseteq \mathbb{N}$ as follows:

$$A = \{a \in \mathbb{N} \mid \neg(\exists x_1, \ldots, x_\nu)(P_a(a, x_1, \ldots, x_\nu) = 0) \text{ is provable from } \mathcal{A}\}, \tag{$**$}$$

where $P_m(a, x_1, \ldots, x_\nu)$ is defined just after Theorem 7.16. Because by (b) $\mathcal{A}$ is listable, it is easy to see (because the set of formulae provable from a listable set is listable) that A is listable, so by the DPRM A is Diophantine, and by Theorem 7.16, there is some $k \in \mathbb{N}$ such that

$$A = W_k = \{a \in \mathbb{N} \mid (\exists x_1, \ldots, x_\nu)(P_k(a, x_1, \ldots, x_\nu) = 0).$$

The trick is now to see whether $k \in W_k$ or not. We claim that $k \notin W_k$.

We proceed by contradiction. Assume that $k \in W_k$. This means that

$$(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0), \tag{$\dagger_1$}$$

and since $A = W_k$, by $(**)$, that

$$\neg(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0) \text{ is provable from } \mathcal{A}. \tag{$\dagger_2$}$$

By ($\dagger_1$) and (c), since the equation $P_k(k, x_1, \ldots, x_\nu) = 0$ has a solution, we can prove the formula

$$(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$$

from $\mathcal{A}$. By ($\dagger_2$), the formula $\neg(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$ is provable from $\mathcal{A}$, but since $(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$ is also provable from $\mathcal{A}$, this contradicts the fact that $\mathcal{A}$ is consistent (which is hypothesis (a)).

Therefore we must have $k \notin W_k$. This means that $P_k(k, x_1, \ldots, x_\nu) = 0$ has **no** solution with $x_1, \ldots, x_\nu \in \mathbb{N}$, and since $A = W_k$, the formula

$$\neg(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$$

is **not** provable from $\mathcal{A}$, since otherwise, by definition of $A = W_k$, we would have $k \in W_k$, contradicting the fact that $k \notin W_k$. □

Remark: Going back to the proof of Theorem 6.16, observe that A plays the role of $\{F_x \mid \neg F_x \text{ is provable}\}$, that k plays the role of x_0, and that the fact that

$$\neg(\exists x_1, \ldots, x_\nu)(P_k(k, x_1, \ldots, x_\nu) = 0)$$

is **not** provable from $\mathcal{A}$ corresponds to $\neg F_{x_0}$ being true.

As a corollary of Theorem 7.17, since the theorems provable in Robinson arithmetic satisfy (a), (b), (c), we deduce that there are true theorems of arithmetic not provable in Robinson arithmetic; in short, Robinson arithmetic is incomplete. Since Robinson arithmetic does not have induction axioms, this shows that induction is not the culprit behind incompleteness. Since Peano arithmetic is an extension (consistent) of Robinson arithmetic, it is also incomplete. This is Gödel's original incompleteness theorem, but Gödel had to develop from scratch the tools needed to prove his result, so his proof is very different (and a tour de force).

But the situation is even more dramatic. Adding a true unprovable statement to a set $\mathcal{A}$ satisfying (a), (b), (c) preserves properties (a), (b), (c), so there is no escape from incompleteness (unless perhaps we allow unreasonable sets of formulae violating (b)). The reader should compare this situation with the results given by Theorem 6.14 and Theorem 6.15.

Gödel's incomplenetess theorem is a negative result, in the sense that it shows that there is no hope of obtaining proof systems capable of proving all true statements for various mathematical theories such as arithmetic. We can also view Gödel's incomplenetess theorem positively as evidence that mathematicians will never be replaced by computers! There is always room for creativity.

The true but unprovable formulae arising in Gödel's incompleteness theorem are rather contrived and by no means "natural." For many years after Gödel's proof was published logicians looked for natural incompleteness phenomena. In the early 1980's such results were found, starting with a result of Kirby and Paris. Harvey Friedman then found more spectacular instances of natural incompleteness, one of which involves a finite miniaturization of Kruskal's tree theorem. The proof of such results uses some deep methods of proof theory involving a tool known as ordinal notations. In particular, two ordinals denoted ϵ_0 and Γ_0 play an important role in consistency proofs; see Takeuti [42] and Schütte [38]. A survey of such results can be found in Gallier [12] and an introduction to ordinals and cardinals is provided in Chapter **??**.

Chapter 8

The Post Correspondence Problem; Applications to Undecidability Results

8.1 The Post Correspondence Problem

The Post correspondence problem (due to Emil Post) is another undecidable problem that turns out to be a very helpful tool for proving problems in logic or in formal language theory to be undecidable.

Definition 8.1. Let Σ be an alphabet with at least two letters. An instance of the *Post Correspondence problem* (for short, PCP) is given by two nonempty sequences $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$ of strings $u_i, v_i \in \Sigma^*$. Equivalently, an instance of the PCP is a sequence of pairs $(u_1, v_1), \ldots, (u_m, v_m)$.

The problem is to find whether there is a (finite) sequence $(i_1, \ldots, i_p)$, with $i_j \in \{1, \ldots, m\}$ for $j = 1, \ldots, p$, so that

$$u_{i_1} u_{i_2} \cdots u_{i_p} = v_{i_1} v_{i_2} \cdots v_{i_p}.$$

Example 8.1. Consider the following problem:

$$(abab, ababaaa), (aaabbb, bb), (aab, baab), (ba, baa), (ab, ba), (aa, a).$$

There is a solution for the string 1234556:

$$abab\, aaabbb\, aab\, ba\, ab\, ab\, aa = ababaaa\, bb\, baab\, baa\, ba\, ba\, a.$$

If you are not convinced that this is a hard problem, try solving the following instance of the PCP:

$$\{(aab, a), (ab, abb), (ab, bab), (ba, aab).\}$$

The shortest solution is a sequence of length 66.

We are beginning to suspect that this is a hard problem. Indeed, it is undecidable!

Theorem 8.1. *(Emil Post, 1946) The Post correspondence problem is undecidable, provided that the alphabet Σ has at least two symbols.*

There are several ways of proving Theorem 8.1, but the strategy is more or less the same: reduce the halting problem to the PCP, by encoding sequences of ID's as partial solutions of the PCP. In Machtey and Young [29] (Section 2.6), the undecidability of the PCP is shown by demonstrating how to simulate the computation of a Turing machine as a sequence of ID's. We give a proof involving special kinds of RAM programs (called Post machines in Manna [30]), which is an adaptation of a proof due to Dana Scott presented in Manna [30] (Section 1.5.4, Theorem 1.8).

Proof. The first step of the proof is to show that a RAM program with $p \geq 2$ registers can be simulated by a RAM program using a single register. The main idea of the simulation is that by using the instructions `add`, `tail`, and `jmp`, it is possible to perform cyclic permutations on the string held by a register.

First we can also assume that RAM programs only uses instructions of the form

(1_j)	N		$\texttt{add}_j$	X
(2)	N		`tail`	X
(6_ja)	N	X	$\texttt{jmp}_j$	$N1a$
(6_jb)	N	X	$\texttt{jmp}_j$	$N1b$
(7)	N		`continue`	

We can simulate $p \geq 2$ registers with a single register, by encoding the contents $r_1, \ldots, r_p$ of the p registers as the string

$$r_1\#r_2\#\cdots\#r_p,$$

using a single marker $\#$. For instance, if $p = 2$, the effect of the instruction $\texttt{add}_b$ on register $R1$ is achieved as follows: Assuming that the initial contents are

$$aab\#baba$$

using cyclic permutations (also inserting or deleting $\#$ whenever necessary), we get

$$\begin{aligned}
&aab\#baba\\
&ab\#baba\#a\\
&b\#baba\#aa\\
&baba\#aab\\
&baba\#aabb \qquad \text{(add } b \text{ on the right)}\\
&aba\#aabb\#b\\
&ba\#aabb\#ba\\
&a\#aabb\#bab\\
&aabb\#baba
\end{aligned}$$

Similarly, the effect of the instruction `tail` on register $R2$ is achieved as follows

$$\begin{array}{ll}
aab\#baba & \\
ab\#baba\#a & \\
b\#baba\#aa & \\
baba\#aab & \\
aba\#aab & \text{(delete the leftmost letter)} \\
ba\#aab\#a & \\
a\#aab\#ab & \\
aab\#aba &
\end{array}$$

Since the halting problem for RAM programs is undecidable and since every RAM program can be simulated by another RAM with a single register, the halting problem for RAM programs with a single register is undecidable.

The second step of the proof is to reduce the halting problem for RAM programs with one register to the PCP (over an alphabet with at least two symbols).

Recall that $\Sigma = \{a_1, \ldots, a_k\}$. First, it is easily shown that every RAM program P (with a single register X) is equivalent to a RAM program P' such that all instructions are labeled with distinct line numbers, and such that there is only one occurrence of the instruction `continue` at the end of the program.

In order to obtain a reasonably simple reduction of the halting problem for RAM programs with a single register to the PCP, we modify the jump instructions as follows: the new instruction

$$\texttt{Jmp}\, N_1, \ldots, N_k, N_{k+1}$$

tests whether $head(X) = a_j$, with $1 \leq j \leq k$. Since there is a single register X, it is omitted in the instruction `Jmp`. If $head(X) = a_j$, then jump to the instruction labeled N_j *and* perform the `tail` instruction so that $X = tail(X)$, otherwise if $X = \epsilon$ (which implies that $j = k + 1$), jump to the instruction labeled N_{k+1}. The instruction `tail` is eliminated. We leave it as an exercise to show how to simulate the new instruction

$$\texttt{Jmp}\, N_1, \ldots, N_k, N_{k+1}$$

using the instructions `tail`, $\texttt{jmp}_j\, Na$ and $\texttt{jmp}_j\, Nb$, and vice-versa. From now on we will use the second version using the instructions

$$\texttt{Jmp}\, N_1, \ldots, N_k, N_{k+1}.$$

For the purpose of deciding whether a RAM program terminates, we may assume without loss of generality that we deal with programs that clear the register X when they halt. In fact, by adding an extra symbol # to the alphabet (which now has $k+1$ symbols), we may also assume that in every instruction

$$\texttt{Jmp}\, N_1, \ldots, N_{k+1}, N_{k+2},$$

N_{k+2} is the line number of the last instruction in the RAM program, which must be a `continue`. This implies that the program clears the register X before it halts. We can execute the instruction $\texttt{add}_{k+1}\ X$ at the very beginning of the program and perform an $\texttt{add}_{k+1}\ X$ after each `tail` instruction to make sure that in the new program the register X always has # as its rightmost symbol. When the original program performs an instruction

$$\texttt{Jmp}\, N_1, \ldots, N_{k+1}, N_{k+2}$$

with $X = \epsilon$, the new program performs the instruction

$$\texttt{Jmp}\, N_1, \ldots, N_{k+2}, N_1.$$

Since X is never empty during execution of the new program, the line number N_1 is irrelevant. Finally, when the original program halts, the new program clears the register X and then jumps to the last `continue`. We leave the details as an exercise.

From now on, we assume that $\Sigma = \{a_1, \ldots, a_k, \#\}$. Given a RAM program P satisfying all the restrictions described above, we construct an instance of the PCP as follows. Assume that P has q lines numbered $N1, \ldots, Nq$. The alphabet Δ of the PCP is $\Delta = \Sigma \cup \{*, N_0, N_1, \ldots, N_q\}$. Indeed, the construction requires one more line number N_0 to force a solution of the PCP to start with some specified pair.

The lists U and V are constructed so that given any nonempty input $x = x_1 \cdots x_m$ (with $x_i \in \Sigma$), the only possible U-lists u and V-lists v that could lead to a solution are of the form

$$u = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}\, w_n * N_{i_{n+1}} *,$$

where each w_i is of the form

$$w_i = *\, w_{i,1} * \cdots * w_{i,n_i} \quad \text{or} \quad w_i = \epsilon,$$

with

$$w_0 = *\, x_1 * x_2 * \cdots * x_m,$$

where $w_{i,j} \in \Sigma$, $1 \leq j \leq n_i$, $1 \leq i \leq n$.

The sequence $N_1, \ldots, N_{i_{n+1}}$ is the sequence of line numbers of the instructions executed by the RAM program P after n steps, started on input x, and w_j is the value of the (single) register X just after executing the jth step, *i.e.*, the instruction at line number N_{i_j}. We make sure that the V-list is always ahead of the U-list by one instruction.

The lists U and V are defined according to the following rules. Rather than defining U and V explicitly, we define the pairs $(u_i,\, v_i)$, where $u_i \in U$ and $v_i \in V$.

To get started: we have the initial pair

$$(N_0,\; N_0 * x_1 * x_2 * \cdots * x_m * N_1 *).$$

To simulate an instruction

$$N_i \qquad \texttt{add}_j \quad X,$$

create the pair

$$(* \, N_i,\; a_j * N_{i+1} *), \quad \text{for all } a \in \Sigma.$$

To simulate an instruction of the form

$$N_i \qquad \texttt{Jmp}\, N_1, \ldots, N_{k+1}, N_{k+2},$$

create the pairs

$$(* \, N_i * \, a_j,\; N_j *), \quad 1 \leq j \leq k+1,$$

and

$$(* \, N_i * N_q, N_q).$$

To build up the register contents, we need pairs

$$(* \, a,\; a *), \quad \text{for all } a \in \Sigma.$$

Note that we used the alphabet $\Delta = \Sigma \cup \{*, N_0, N_1, \ldots, N_q\}$, which uses more than 2 symbols in general. Let us finish our reduction for an instance of the PCP over the alphabet Δ. Then after this construction is finished we will explain how to convert the instance of the PCP that we obtained to an instance of the PCP over a two-symbol alphabet.

The pairs of the PCP are designed so that the only possible U-lists u and V-lists v that could lead to a solution are of the form

$$u = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}\, w_n * N_{i_{n+1}} *,$$

where each w_i is of the form

$$w_i = * \, w_{i,1} * \cdots * w_{i,n_i} \quad \text{or} \quad w_i = \epsilon,$$

with

$$w_0 = * \, x_1 * x_2 * \cdots * x_m,$$

where $w_{i,j} \in \Sigma$, $1 \leq j \leq n_i$, $1 \leq i \leq n$, and where v is an encoding of n steps of the computation of the RAM program P on input $x = x_1 \cdots x_m$, and u lags behind v by one step.

For example, let us see how the U-list and the V-list are updated, assuming that N_{i_n} is the following instruction:

N_{i_n} $\quad$ `add`$_b$ $\quad$ X

Just after execution of the instruction at line number N_{i_n}, we have

$$u = N_0\, w_0 * N_1\, w_1 * \cdots * \; N_{i_{n-1}}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n} \; * .$$

Since $w_{n-1} = * \, w_{n-1,1} * \cdots * w_{n-1,n_{n-1}}$, using the pairs

$$(* \, w_{n-1,1},\; w_{n-1,1} *),\; (* \, w_{n-1,2},\; w_{n-1,2} \, *), \cdots, (* \, w_{n-1,n_{n-1}},\; w_{n-1,n_{n-1}} \, *),$$

we get

$$u = N_0\, w_0 * N_1\, w_1 * \cdots * \; N_{i_{n-1}}\, w_{n-1}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}}\, w_{n-1} * N_{i_n}\, w_{n-1} \; * .$$

Next we use the pair

$$(* \, N_{i_n},\, b * N_{i_{n+1}} \, *)$$

simulating `add`$_b$, and we get

$$u = N_0\, w_0 * N_1\, w_1 * \cdots * \; N_{i_{n-1}}\, w_{n-1} * N_{i_n}$$

and

$$v = N_0\, w_0 * N_1\, w_1 * \cdots * N_{i_{n-1}} \; w_{n-1} * N_{i_n}\, w_{n-1} \; * b * N_{i_{n+1}} \; * .$$

Observe that the only chance for getting a solution of the PCP is to start with the pairs involving N_0. It is easy to see that the PCP constructed from P has a solution iff P halts on input x. However, the halting problem for RAM's with a single register is undecidable, and thus, the PCP over the alphabet Δ is also undecidable.

It remains to show that we can recode the instance of the PCP that we obtained over the alphabet $\Delta = \Sigma \cup \{*, N_0, N_1, \ldots, N_q\}$ as an instance of the PCP over the alphabet $\{a_1, *\}$. To achieve this, we recode each symbol a_i in Σ as $*a_1^i$ (with $a_{k+1} = \#$) and each N_j as $*a_1^{k+j+2}$. This way, we are only using the alphabet $\Delta = \{a_1, *\}$. We need the second character $*$, whose purpose is to avoid trivial solutions of the form

$$(u, u).$$

This could happen if we had used pairs (a, a) to build up the register. Then we substitute $*a_1^i$ for a_i and $*a_1^{k+j+2}$ for N_j in the pairs that we created. Observe that the pairs $(*a, a*)$ become pairs involving longer strings. It is easy to see that the original PCP over Δ has a solution iff the new PCP over $\{a_1, *\}$ has a solution, so the PCP over two-letter alphabet is undecidable. □

In the next two sections we present some undecidability results for context-free grammars and context-free languages.

8.2 Some Undecidability Results for CFG's

Theorem 8.2. *It is undecidable whether a context-free grammar is ambiguous.*

Proof. We reduce the PCP to the ambiguity problem for CFG's. Given any instance $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$ of the PCP, let $c_1, \ldots, c_m$ be m new symbols, and consider the following languages:

$$\begin{aligned} L_U = \{u_{i_1} \cdots u_{i_p} c_{i_p} \cdots c_{i_1} \mid 1 \leq i_j \leq m, \\ 1 \leq j \leq p,\, p \geq 1\}, \\ L_V = \{v_{i_1} \cdots v_{i_p} c_{i_p} \cdots c_{i_1} \mid 1 \leq i_j \leq m, \\ 1 \leq j \leq p,\, p \geq 1\}, \end{aligned}$$

and $L_{U,V} = L_U \cup L_V$.

We can easily construct a CFG, $G_{U,V}$, generating $L_{U,V}$. The productions are:

$$\begin{aligned} S &\longrightarrow S_U \\ S &\longrightarrow S_V \\ S_U &\longrightarrow u_i S_U c_i \\ S_U &\longrightarrow u_i c_i \\ S_V &\longrightarrow v_i S_V c_i \\ S_V &\longrightarrow v_i c_i. \end{aligned}$$

It is easily seen that the PCP for (U, V) has a solution iff $L_U \cap L_V \neq \emptyset$ iff G is ambiguous. □

Remark: As a corollary, we also obtain the following result: it is undecidable for arbitrary context-free grammars G_1 and G_2 whether $L(G_1) \cap L(G_2) = \emptyset$ (see also Theorem 8.4).

Recall that the computations of a Turing Machine, M, can be described in terms of instantaneous descriptions, $upav$.

We can encode computations

$$ID_0 \vdash ID_1 \vdash \cdots \vdash ID_n$$

halting in a proper ID, as the language, L_M, consisting all of strings

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k}\#w_{2k+1}^R,$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $k \geq 0$, w_0 is a starting ID, $w_i \vdash w_{i+1}$ for all i with $0 \leq i < 2k+1$ and w_{2k+1} is proper halting ID in the first case, $0 \leq i < 2k$ and w_{2k} is proper halting ID in the second case.

The language L_M turns out to be the intersection of two context-free languages L_M^0 and L_M^1 defined as follows:

(1) The strings in L_M^0 are of the form

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k}\#w_{2k+1}^R$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $w_{2i} \vdash w_{2i+1}$ for all $i \geq 0$, and w_{2k} is a proper halting ID in the second case.

(2) The strings in L_M^1 are of the form

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k}\#w_{2k+1}^R$$

or

$$w_0\#w_1^R\#w_2\#w_3^R\#\cdots\#w_{2k-2}\#w_{2k-1}^R\#w_{2k},$$

where $w_{2i+1} \vdash w_{2i+2}$ for all $i \geq 0$, w_0 is a starting ID, and w_{2k+1} is a proper halting ID in the first case.

Theorem 8.3. *Given any Turing machine M, the languages L_M^0 and L_M^1 are context-free, and $L_M = L_M^0 \cap L_M^1$.*

Proof. We can construct PDA's accepting L_M^0 and L_M^1. It is easily checked that $L_M = L_M^0 \cap L_M^1$. □

As a corollary, we obtain the following undecidability result:

Theorem 8.4. *It is undecidable for arbitrary context-free grammars G_1 and G_2 whether $L(G_1) \cap L(G_2) = \emptyset$.*

Proof. We can reduce the problem of deciding whether a partial recursive function is undefined everywhere to the above problem. By Rice's theorem, the first problem is undecidable.

However, this problem is equivalent to deciding whether a Turing machine never halts in a proper ID. By Theorem 8.3, the languages L_M^0 and L_M^1 are context-free. Thus, we can construct context-free grammars G_1 and G_2 so that $L_M^0 = L(G_1)$ and $L_M^1 = L(G_2)$. Then M never halts in a proper ID iff $L_M = \emptyset$ iff (by Theorem 8.3), $L_M = L(G_1) \cap L(G_2) = \emptyset$. □

Given a Turing machine M, the language L_M is defined over the alphabet $\Delta = \Gamma \cup Q \cup \{\#\}$. The following fact is also useful to prove undecidability:

Theorem 8.5. *Given any Turing machine M, the language $\Delta^* - L_M$ is context-free.*

Proof. One can easily check that the conditions for not belonging to L_M can be checked by a PDA. □

As a corollary, we obtain:

Theorem 8.6. *Given any context-free grammar, $G = (V, \Sigma, P, S)$, it is undecidable whether $L(G) = \Sigma^*$.*

Proof. We can reduce the problem of deciding whether a Turing machine never halts in a proper ID to the above problem.

Indeed, given M, by Theorem 8.5, the language $\Delta^* - L_M$ is context-free. Thus, there is a CFG, G, so that $L(G) = \Delta^* - L_M$. However, M never halts in a proper ID iff $L_M = \emptyset$ iff $L(G) = \Delta^*$. □

As a consequence, we also obtain the following:

Theorem 8.7. *Given any two context-free grammar, G_1 and G_2, and any regular language, R, the following facts hold:*

(1) $L(G_1) = L(G_2)$ is undecidable.

(2) $L(G_1) \subseteq L(G_2)$ is undecidable.

(3) $L(G_1) = R$ is undecidable.

(4) $R \subseteq L(G_2)$ is undecidable.

In contrast to (4), the property $L(G_1) \subseteq R$ is decidable!

8.3 More Undecidable Properties of Languages; Greibach's Theorem

We discuss a nice theorem of S. Greibach, which is a sort of version of Rice's theorem for families of languages.

Let $\mathcal{L}$ be a countable family of languages. We assume that there is a coding function $c\colon \mathcal{L} \to \mathbb{N}$ and that this function can be extended to code the regular languages (all alphabets are subsets of some given countably infinite set).

We also assume that $\mathcal{L}$ is effectively closed under union, and concatenation with the regular languages.

This means that given any two languages L_1 and L_2 in $\mathcal{L}$, we have $L_1 \cup L_2 \in \mathcal{L}$, and $c(L_1 \cup L_2)$ is given by a recursive function of $c(L_1)$ and $c(L_2)$, and that for every regular language R, we have $L_1R \in \mathcal{L}$, $RL_1 \in \mathcal{L}$, and $c(RL_1)$ and $c(L_1R)$ are recursive functions of $c(R)$ and $c(L_1)$.

Given any language, $L \subseteq \Sigma^*$, and any string, $w \in \Sigma^*$, we define L/w by

$$L/w = \{u \in \Sigma^* \mid uw \in L\}.$$

Theorem 8.8. *(Greibach) Let $\mathcal{L}$ be a countable family of languages that is effectively closed under union and concatenation with the regular languages, and assume that the problem $L = \Sigma^*$ is undecidable for $L \in \mathcal{L}$ and any given sufficiently large alphabet Σ. Let P be any nontrivial property of languages that is true for the regular languages, so that if $P(L)$ holds for any $L \in \mathcal{L}$, then $P(L/a)$ also holds for any letter a. Then P is undecidable for $\mathcal{L}$.*

Proof. Since P is nontrivial for $\mathcal{L}$, there is some $L_0 \in \mathcal{L}$ so that $P(L_0)$ is false.

Let Σ be large enough, so that $L_0 \subseteq \Sigma^*$, and the problem $L = \Sigma^*$ is undecidable for $L \in \mathcal{L}$.

We show that given any $L \in \mathcal{L}$, with $L \subseteq \Sigma^*$, we can construct a language $L_1 \in \mathcal{L}$, so that $L = \Sigma^*$ iff $P(L_1)$ holds. Thus, the problem $L = \Sigma^*$ for $L \in \mathcal{L}$ reduces to property P for $\mathcal{L}$, and since for Σ big enough, the first problem is undecidable, so is the second.

For any $L \in \mathcal{L}$, with $L \subseteq \Sigma^*$, let

$$L_1 = L_0\#\Sigma^* \cup \Sigma^*\#L.$$

Since $\mathcal{L}$ is effectively closed under union and concatenation with the regular languages, we have $L_1 \in \mathcal{L}$.

If $L = \Sigma^*$, then $L_1 = \Sigma^*\#\Sigma^*$, a regular language, and thus, $P(L_1)$ holds, since P holds for the regular languages.

Conversely, we would like to prove that if $L \neq \Sigma^*$, then $P(L_1)$ is false.

Since $L \neq \Sigma^*$, there is some $w \notin L$. But then,

$$L_1/\#w = L_0.$$

Since P is preserved under quotient by a single letter, by a trivial induction, if $P(L_1)$ holds, then $P(L_0)$ also holds. However, $P(L_0)$ is false, so $P(L_1)$ must be false.

Thus, we proved that $L = \Sigma^*$ iff $P(L_1)$ holds, as claimed. □

Greibach's theorem can be used to show that it is undecidable whether a context-free grammar generates a regular language.

It can also be used to show that it is undecidable whether a context-free language is inherently ambiguous.

8.4 Undecidability of Validity in First-Order Logic

The PCP can also be used to give a quick proof of Church's famous result stating that validity in first-order logic is undecidable. Here we are considering first-order formulae as defined in Section **??**. Given a first-order language **L** consisting of constant symbols c, function symbols f, and predicate symbols P, a *first-order structure* $\mathcal{M}$ consists of a nonempty domain M, of an assigment of some element of $c_{\mathcal{M}} \in M$ to every constant symbol c, of a function $f_{\mathcal{M}} \colon M^n \to M$ to every n-ary function symbol f, and to a boolean-valued function $P_{\mathcal{M}} \colon M^m \to \{\mathbf{T}, \mathbf{F}\}$ to any m-ary predicate symbol P.

Then given any assignment $\rho \colon X \to M$ to the first-order variables $x_i \in X$, we can define recursively the truth value $\varphi_{\mathcal{M}}[\rho]$ of every first-order formula φ. If φ is a sentence, which means that φ has no free variables, then the truth value $\varphi_{\mathcal{M}}[\rho]$ is independent of ρ, so we simply write $\varphi_{\mathcal{M}}$. Details can be found in Gallier [18], Enderton [11], or Shoenfield [39]. The formula φ is *valid in* $\mathcal{M}$ if $\varphi_{\mathcal{M}}[\rho] = \mathbf{T}$ for all ρ. We also say that $\mathcal{M}$ is a *model* of φ and we write

$$\mathcal{M} \models \varphi.$$

The formula φ is *valid* (or *universally valid*) if it is valid in *every* first-order structure $\mathcal{M}$; this denoted by

$$\models \varphi.$$

The *validity problem* in first-order logic is to decide whether there is algorithm to decide whether any first-order formula is valid.

Theorem 8.9. *(Church, 1936) The validity problem for first-order logic is undecidable.*

Proof. The following proof due to R. Floyd is given in Manna [30] (Section 2.16). The proof consists in reducing the PCP over the alphabet $\{0, 1\}$ to the validity problem. Given an instance $S = (U, V)$ of the PCP, we construct a first-order sentence Φ_S (using a computable function) such that S has a solution if and only if Φ_S is valid. Since the PCP is undecidable, so is the validity problem for first-order logic.

For this construction, we need a constant symbol a, two unary function symbols f_0 and f_1, and a binary predicate symbol P. We denote the term

$$f_{s_p}(\cdots(f_{s_2}(f_{s_1}(x))\cdots)$$

as $f_{s_1s_2\cdots s_p}$, where $s_i \in \{0,1\}$. Suppose S is the set of pairs

$$S = \{(u_1, v_1), \ldots, (u_m, v_m)\}.$$

The key ingredent is the sentence

$$\begin{aligned}\Phi_S \equiv \Big(&\bigwedge_{i=1}^{m} P(f_{u_i}(a), f_{v_i}(a)) \land \forall x \forall y \Big(P(x,y) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(x), f_{v_i}(x))\Big)\Big)\\ &\Rightarrow \exists z P(z,z).\end{aligned}$$

We claim that the PCP S has a solution iff Φ_S is valid.

Step 1. We prove that if Φ_S is valid, then the PCP has a solution. Consider the first-order structure $\mathcal{M}$ with domain $\{0,1\}^*$, with $a_{\mathcal{M}} = \epsilon$, $(f_0)_{\mathcal{M}}$ is concatenation on the right with 0 ($(f_0)_{\mathcal{M}}(x) = x0$), $(f_1)_{\mathcal{M}}$ is concatenation on the right with 1 ($(f_1)_{\mathcal{M}}(x) = x1$), and

$$P(x,y) = \mathbf{T} \quad \text{iff} \quad x = u_{i_1}u_{i_2}\cdots u_{i_n}, \quad y = v_{i_1}v_{i_2}\cdots v_{i_n},$$

for some nonempty sequence $i_1, i_2, \ldots, i_n$ with $1 \leq i_j \leq m$.

Since Φ_S is valid, it must be valid in $\mathcal{M}$, but then we see immediately that both

$$\bigwedge_{i=1}^{m} P(f_{u_i}(a), f_{v_i}(a))$$

and

$$\forall x \forall y \Big(P(x,y) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(x), f_{v_i}(x))\Big)$$

are valid in $\mathcal{M}$, thus

$$\exists z P(z,z)$$

is also valid in $\mathcal{M}$. This means that there is some nonempty sequence $i_1, i_2, \ldots, i_n$ with $1 \leq i_j \leq m$ such that

$$z = u_{i_1}u_{i_2}\cdots u_{i_n} = v_{i_1}v_{i_2}\cdots v_{i_n},$$

and so we have a solution of the PCP.

Step 2. We prove that if the PCP has a solution, then Φ_S is valid. Let $i_1, i_2, \ldots, i_n$ be a nonempty sequence with $1 \leq i_j \leq m$ such that

$$u_{i_1}u_{i_2}\cdots u_{i_n} = v_{i_1}v_{i_2}\cdots v_{i_n},$$

which means that $i_1, i_2, \ldots, i_n$ is a solution of the PCP S. We prove that for every first-order structure $\mathcal{M}$, if

$$\bigwedge_{i=1}^{m} P(f_{u_i}(a), f_{v_i}(a))$$

and

$$\forall x \forall y \Big(P(x, y) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(x), f_{v_i}(x)) \Big)$$

are valid in $\mathcal{M}$, then

$$\exists z P(z, z)$$

is also valid in $\mathcal{M}$. But then Φ_S is valid in *every* first-order structure $\mathcal{M}$, and thus it is valid.

To finish the proof, assume that $\mathcal{M}$ is any first-order structure such that

$$\bigwedge_{i=1}^{m} P(f_{u_i}(a), f_{v_i}(a)) \qquad (*_1)$$

and

$$\forall x \forall y \Big(P(x, y) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(x), f_{v_i}(x)) \Big) \qquad (*_2)$$

are valid in $\mathcal{M}$. Using $(*_1)$, by repeated application on $(*_2)$, we deduce that

$$P(f_{u_{i_1} u_{i_2} \cdots u_{i_n}}(a), f_{v_{i_1} v_{i_2} \cdots v_{i_n}}(a)),$$

is valid in $\mathcal{M}$. For example, since (u_{i_1}, v_{i_1}) is a pair in the PCP instance, by $(*_1)$ the proposition $P(f_{u_{i_1}}(a), f_{v_{i_1}}(a))$ holds, so by $(*_2)$ with $x = f_{u_{i_1}}(a)$ and $v = f_{v_{i_1}}(a)$, we get the implication

$$P(f_{u_{i_1}}(a), f_{v_{i_1}}(a)) \Rightarrow \bigwedge_{i=1}^{m} P(f_{u_i}(f_{u_{i_1}}(a)), f_{v_i}(f_{v_{i_1}}(a))),$$

and since $P(f_{u_{i_1}}(a), f_{v_{i_1}}(a))$ holds, we deduce that $\bigwedge_{i=1}^{m} P(f_{u_i}(f_{u_{i_1}}(a)), f_{v_i}(f_{v_{i_1}}(a)))$ holds, and consequently $P(f_{u_{i_2}}(f_{u_{i_1}}(a)), f_{v_{i_2}}(f_{v_{i_1}}(a))) = P(f_{u_{i_1} u_{i_2}}(a), f_{v_{i_1} v_{2_1}}(a))$ holds.

Since by hypothesis

$$u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n},$$

we deduce that $\exists z P(z, z)$ is valid in $\mathcal{M}$, and so Φ_S is valid in $\mathcal{M}$, as claimed. $\square$

There are other ways of proving Church's theorem. Among other sources, see Shoenfield [39] (Section 6.8) and Machtey and Young [29] (Chapter 4, theorem 4.3.6). These proofs are rather long and involve complicated arguments. Floyd's proof has the virtue of being quite short and transparent, if we accept the undecidability of the PCP.

Lewis shows the stronger result than even with a *single* unary function symbol f, one constant a, and one binary predicate symbol P, the validity problem is undecidable; see

Lewis [27] (Chapter IIC). Lewis' proof is a very clever reduction of a tiling problem. Lewis' book also contains an extensive classification of undecidable classes of first-order sentences. On the positive side, Dreben and Goldfarb [10] contains a very complete study of classes of first-order sentences for which the validity problem *is* decidable.

Chapter 9

Computational Complexity; $\mathcal{P}$ and $\mathcal{NP}$

9.1 The Class $\mathcal{P}$

In the previous two chapters, we clarified what it means for a problem to be decidable or undecidable. This chapter is heavily inspired by Lewis and Papadimitriou's excellent treatment [28].

In principle, if a problem is decidable, then there is an algorithm (i.e., a procedure that halts for every input) that decides every instance of the problem.

However, from a practical point of view, knowing that a problem is decidable may be useless, if the number of steps (*time complexity*) required by the algorithm is excessive, for example, exponential in the size of the input, or worse.

For instance, consider the *traveling salesman problem*, which can be formulated as follows:

We have a set $\{c_1, \ldots, c_n\}$ of cities, and an $n \times n$ matrix $D = (d_{ij})$ of nonnegative integers, the *distance matrix*, where d_{ij} denotes the distance between c_i and c_j, which means that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all $i \neq j$.

The problem is to find a *shortest tour* of the cities, that is, a permutation π of $\{1, \ldots, n\}$ so that the *cost*

$$C(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)}$$

is as small as possible (minimal).

One way to solve the problem is to consider all possible tours, i.e., $n!$ permutations. Actually, since the starting point is irrelevant, we need only consider $(n-1)!$ tours, but this still grows very fast. For example, when $n = 40$, it turns out that $39!$ exceeds 10^{45}, a huge number.

Consider the 4×4 symmetric matrix given by

$$D = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix},$$

and the budget $B = 4$. The tour specified by the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix}$$

has cost 4, since

$$\begin{aligned} c(\pi) &= d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)} \\ &= d_{14} + d_{42} + d_{23} + d_{31} \\ &= 1 + 1 + 1 + 1 = 4. \end{aligned}$$

The cities in this tour are traversed in the order

$$(1, 4, 2, 3, 1).$$

Remark: The permutation π shown above is described in Cauchy's *two-line notation*,

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix},$$

where every element in the second row is the image of the element immediately above it in the first row: thus

$$\pi(1) = 1,\ \pi(2) = 4,\ \pi(3) = 2,\ \pi(4) = 3.$$

Thus, to capture the essence of practically feasible algorithms, we must limit our computational devices to run only for a number of steps that is bounded by a *polynomial* in the length of the input.

We are led to the definition of polynomially bounded computational models.

We talked about problems being decidable in polynomial time. Obviously, this is equivalent to deciding some property of a certain class of objects, for example, finite graphs.

Our framework requires that we first encode these classes of objects as strings (or numbers), since $\mathcal{P}$ consists of languages.

Thus, when we say that a property is decidable in polynomial time, we are really talking about the encoding of this property as a language. Thus, we have to be careful about these encodings, but it is rare that encodings cause problems.

Definition 9.1. A deterministic Turing machine M is said to be *polynomially bounded* if there is a polynomial $p(X)$ so that the following holds: for every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is *polynomially decidable* if there is a polynomially bounded Turing machine that accepts L. The family of all polynomially decidable languages is denoted by $\mathcal{P}$.

Remark: Even though Definition 9.1 is formulated for Turing machines, it can also be formulated for other models, such as RAM programs. The reason is that the conversion of a Turing machine into a RAM program (and vice versa) produces a program (or a machine) whose size is polynomial in the original device.

The following proposition, although trivial, is useful:

Proposition 9.1. *The class $\mathcal{P}$ is closed under complementation.*

Of course, many languages do not belong to $\mathcal{P}$. One way to obtain such languages is to use a diagonal argument. But there are also many natural languages that are not in $\mathcal{P}$, although this may be very hard to prove for some of these languages.

Let us consider a few more problems in order to get a better feeling for the family $\mathcal{P}$.

9.2 Directed Graphs, Paths

Recall that a *directed graph*, G, is a pair $G = (V, E)$, where $E \subseteq V \times V$. Every $u \in V$ is called a *node* (or *vertex*) and a pair $(u, v) \in E$ is called an *edge* of G.

We will restrict ourselves to *simple graphs*, that is, graphs without edges of the form (u, u); equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of $n + 1$ edges ($n \geq 0$)

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from u to v is simply a single edge, (u, v).)

A graph G is *strongly connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself.

We will restrict out attention to finite graphs, i.e. graphs (V, E) where V is a finite set.

Definition 9.2. Given a graph G, an *Eulerian cycle* is a cycle in G that passes through all the nodes (possibly more than once) and every edge of G exactly once. A *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Eulerian Cycle Problem: Given a graph G, is there an Eulerian cycle in G?

Hamiltonian Cycle Problem: Given a graph G, is there an Hamiltonian cycle in G?

9.3 Eulerian Cycles

The following graph is a directed graph version of the Königsberg bridge problem, solved by Euler in 1736.

The nodes A, B, C, D correspond to four areas of land in Königsberg and the edges to the seven bridges joining these areas of land.

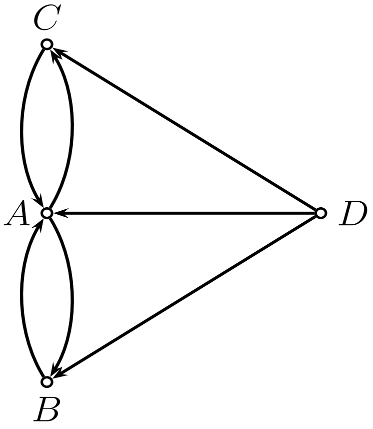


Figure 9.1: A directed graph modeling the Königsberg bridge problem.

The problem is to find a closed path that crosses every bridge exactly once and returns to the starting point.

In fact, the problem is unsolvable, as shown by Euler, because some nodes do not have the same number of incoming and outgoing edges (in the undirected version of the problem, some nodes do not have an even degree.)

It may come as a surprise that the Eulerian Cycle Problem does have a polynomial time algorithm, but that so far, not such algorithm is known for the Hamiltonian Cycle Problem. The reason why the Eulerian Cycle Problem is decidable in polynomial time is the following theorem due to Euler:

Theorem 9.2. *A graph $G = (V, E)$ has an Eulerian cycle iff the following properties hold:*

(1) The graph G is strongly connected.

(2) Every node has the same number of incoming and outgoing edges.

Proving that properties (1) and (2) hold if G has an Eulerian cycle is fairly easy. The converse is harder, but not that bad (try!).

Theorem 9.2 shows that it is necessary to check whether a graph is strongly connected. This can be done by computing the transitive closure of E, which can be done in polynomial time (in fact, $O(n^3)$).

Checking property (2) can clearly be done in polynomial time. Thus, the Eulerian cycle problem is in $\mathcal{P}$. Unfortunately, no theorem analogous to Theorem 9.2 is known for Hamiltonian cycles.

9.4 Hamiltonian Cycles

A game invented by Sir William Hamilton in 1859 uses a regular solid dodecahedron whose twenty vertices are labeled with the names of famous cities.

The player is challenged to "travel around the world" by finding a closed cycle along the edges of the dodecahedron which passes through every city exactly once (this is the undirected version of the Hamiltonian cycle problem). See Figure 9.2.

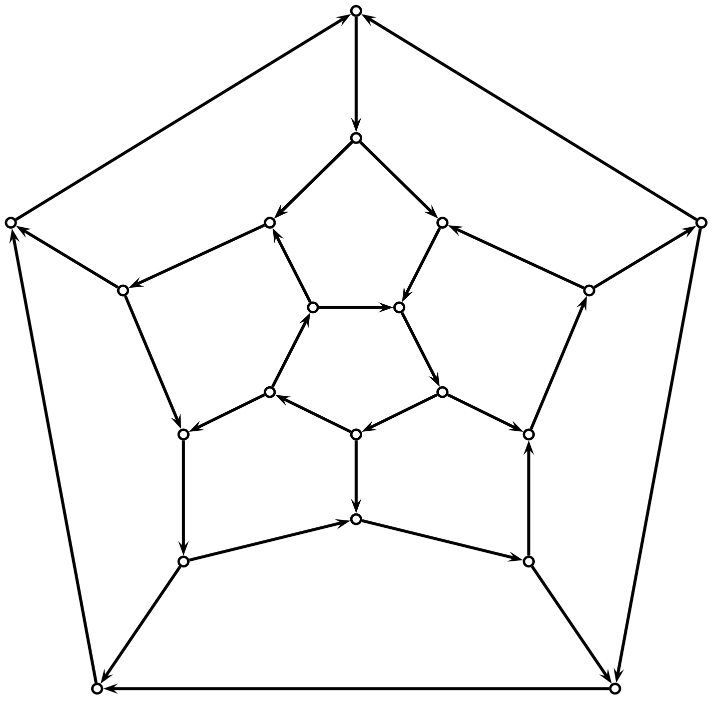


Figure 9.2: A tour "around the world."

In graphical terms, assuming an orientation of the edges between cities, the graph D shown in Figure 9.2 is a plane projection of a regular dodecahedron and we want to know if there is a Hamiltonian cycle in this directed graph. Finding a Hamiltonian cycle in this graph does not appear to be so easy!

A solution is shown in Figure 9.3 below.

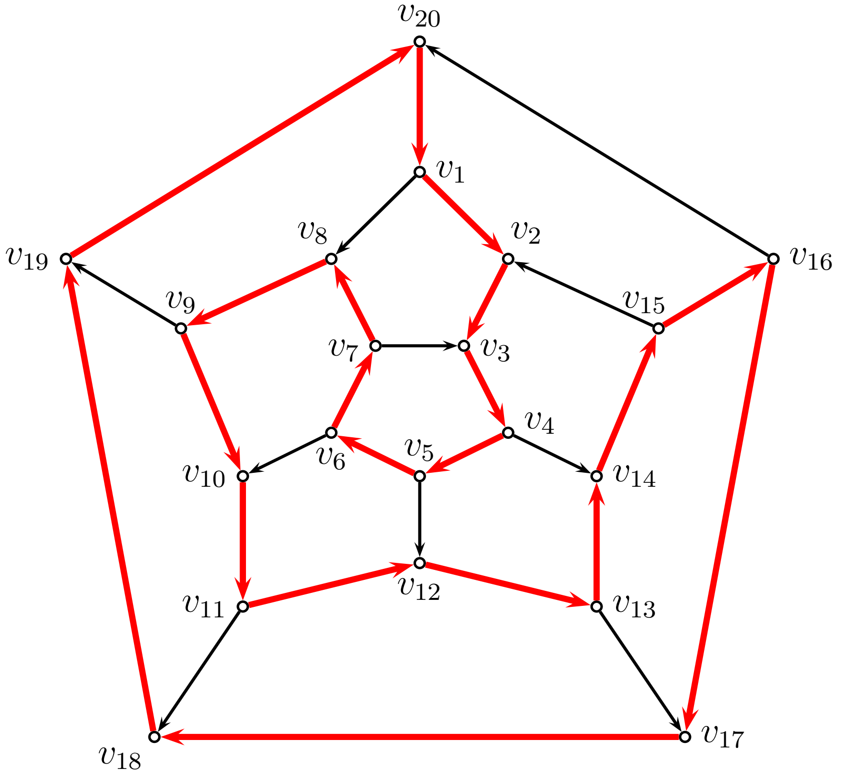


Figure 9.3: A Hamiltonian cycle in D.

9.5 Propositional Logic and Satisfiability

We define the syntax and the semantics of propositions in conjunctive normal form (CNF).

The syntax has to do with the legal form of propositions in CNF. Such propositions are interpreted as truth functions, by assigning truth values to their variables.

We begin by defining propositions in CNF. Such propositions are constructed from a countable set, **PV**, of *propositional (or boolean) variables*, say

$$\mathbf{PV} = \{x_1, x_2, \ldots, \},$$

using the connectives $\wedge$ (and), $\vee$ (or) and $\neg$ (negation).

Definition 9.3. We define a *literal (or atomic proposition)*, L, as $L = x$ or $L = \neg x$, also denoted by $\overline{x}$, where $x \in \mathbf{PV}$.

A *clause*, C, is a disjunction of pairwise distinct literals,

$$C = (L_1 \vee L_2 \vee \cdots \vee L_m).$$

Thus, a clause may also be viewed as a nonempty *set*

$$C = \{L_1, L_2, \ldots, L_m\}.$$

We also have a special clause, the *empty clause*, denoted $\perp$ or $\square$ (or $\{\}$). It corresponds to the truth value false.

A *proposition in CNF, or boolean formula*, P, is a conjunction of pairwise distinct clauses

$$P = C_1 \wedge C_2 \wedge \cdots \wedge C_n.$$

Thus, a boolean formula may also be viewed as a nonempty *set*

$$P = \{C_1, \ldots, C_n\},$$

but this time, the comma is interpreted as conjunction. We also allow the proposition $\perp$, and sometimes the proposition $\top$ (corresponding to the truth value true).

For example, here is a boolean formula:

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}.$$

In order to interpret boolean formulae, we use truth assignments.

Definition 9.4. We let $\mathrm{BOOL} = \{\mathbf{F}, \mathbf{T}\}$, the set of *truth values*, where $\mathbf{F}$ stands for false and $\mathbf{T}$ stands for true. A *truth assignment (or valuation)*, v, is any function $v\colon \mathbf{PV} \to \mathrm{BOOL}$.

Example 9.1. The function $v_F\colon \mathbf{PV} \to \mathrm{BOOL}$ given by

$$v_F(x_i) = \mathbf{F} \quad \text{for all } i \geq 1$$

is a truth assigmnent, and so is the function $v_T\colon \mathbf{PV} \to \mathrm{BOOL}$ given by

$$v_T(x_i) = \mathbf{T} \quad \text{for all } i \geq 1.$$

The function $v\colon \mathbf{PV} \to \mathrm{BOOL}$ given by

$$\begin{aligned} v(x_1) &= \mathbf{T} \\ v(x_2) &= \mathbf{F} \\ v(x_3) &= \mathbf{T} \\ v(x_i) &= \mathbf{T} \quad \text{for all } i \geq 4 \end{aligned}$$

is also a truth assignment.

Definition 9.5. Given a truth assignment $v \colon \mathbf{PV} \to \mathrm{BOOL}$, we define the *truth value* $\widehat{v}(X)$ of a literal, clause, and boolean formula, X, using the following recursive definition:

(1) $\widehat{v}(\perp) = \mathbf{F}$, $\widehat{v}(\top) = \mathbf{T}$.

(2) $\widehat{v}(x) = v(x)$, if $x \in \mathbf{PV}$.

(3) $\widehat{v}(\overline{x}) = \overline{v(x)}$, if $x \in \mathbf{PV}$, where $\overline{v(x)} = \mathbf{F}$ if $v(x) = \mathbf{T}$ and $\overline{v(x)} = \mathbf{T}$ if $v(x) = \mathbf{F}$.

(4) $\widehat{v}(C) = \mathbf{F}$ if C is a clause and iff $\widehat{v}(L_i) = \mathbf{F}$ for all literals L_i in C, otherwise $\mathbf{T}$.

(5) $\widehat{v}(P) = \mathbf{T}$ if P is a boolean formula and iff $\widehat{v}(C_j) = \mathbf{T}$ for all clauses C_j in P, otherwise $\mathbf{F}$.

Since a boolean formula P only contains a finite number of variables, say $\{x_{i_1}, \ldots, x_{i_n}\}$, one should expect that its truth value $\widehat{v}(P)$ depends only on the truth values assigned by the truth assignment v to the variables in the set $\{x_{i_1}, \ldots, x_{i_n}\}$, and this is indeed the case. The following proposition is easily shown by induction on the depth of P (viewed as a tree).

Proposition 9.3. *Let P be a boolean formula containing the set of variables $\{x_{i_1}, \ldots, x_{i_n}\}$. If $v_1 \colon \mathbf{PV} \to \mathrm{BOOL}$ and $v_2 \colon \mathbf{PV} \to \mathrm{BOOL}$ are any truth assignments agreeing on the set of variables $\{x_{i_1}, \ldots, x_{i_n}\}$, which means that*

$$v_1(x_{i_j}) = v_2(x_{i_j}) \quad \textit{for } j = 1, \ldots, n,$$

then $\widehat{v_1}(P) = \widehat{v_2}(P)$.

In view of Proposition 9.3, given any boolean formula P, we only need to specify the values of a truth assignment v for the variables occurring on P.

Example 9.2. Given the boolean formula

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\},$$

we only need to specify $v(x_1), v(x_2), v(x_3)$. Thus there are $2^3 = 8$ distinct truth assignments:

$$\begin{array}{lcl}
\mathbf{F}, \mathbf{F}, \mathbf{F} & \qquad\qquad & \mathbf{T}, \mathbf{F}, \mathbf{F} \\
\mathbf{F}, \mathbf{F}, \mathbf{T} & & \mathbf{T}, \mathbf{F}, \mathbf{T} \\
\mathbf{F}, \mathbf{T}, \mathbf{F} & & \mathbf{T}, \mathbf{T}, \mathbf{F} \\
\mathbf{F}, \mathbf{T}, \mathbf{T} & & \mathbf{T}, \mathbf{T}, \mathbf{T}.
\end{array}$$

In general, there are 2^n distinct truth assignments to n distinct variables.

Example 9.3. Here is an example showing the evaluation of the truth value $\widehat{v}(P)$ for the boolean formula

$$\begin{aligned}
P &= (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_3} \vee x_1) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \\
&= \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\},
\end{aligned}$$

and the truth assignment

$$v(x_1) = \mathbf{T}, \quad v(x_2) = \mathbf{F}, \quad v(x_3) = \mathbf{F}.$$

For the literals, we have

$$\widehat{v}(x_1) = \mathbf{T}, \quad \widehat{v}(x_2) = \mathbf{F}, \quad \widehat{v}(x_3) = \mathbf{F}, \quad \widehat{v}(\overline{x_1}) = \mathbf{F}, \quad \widehat{v}(\overline{x_2}) = \mathbf{T}, \quad \widehat{v}(\overline{x_3}) = \mathbf{T},$$

for the clauses

$$\begin{aligned}
\widehat{v}(x_1 \vee x_2 \vee x_3) &= \widehat{v}(x_1) \vee \widehat{v}(x_2) \vee \widehat{v}(x_3) = \mathbf{T} \vee \mathbf{F} \vee \mathbf{F} = \mathbf{T}, \\
\widehat{v}(\overline{x_1} \vee x_2) &= \widehat{v}(\overline{x_1}) \vee \widehat{v}(x_2) = \mathbf{F} \vee \mathbf{F} = \mathbf{F}, \\
\widehat{v}(\overline{x_2} \vee x_3) &= \widehat{v}(\overline{x_2}) \vee \widehat{v}(x_3) = \mathbf{T} \vee \mathbf{F} = \mathbf{T}, \\
\widehat{v}(\overline{x_3} \vee x_1) &= \widehat{v}(\overline{x_3}) \vee \widehat{v}(x_1) = \mathbf{T} \vee \mathbf{T} = \mathbf{T}, \\
\widehat{v}(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) &= \widehat{v}(\overline{x_1}) \vee \widehat{v}(\overline{x_2}) \vee \widehat{v}(\overline{x_3}) = \mathbf{F} \vee \mathbf{T} \vee \mathbf{T} = \mathbf{T},
\end{aligned}$$

and for the conjunction of the clauses,

$$\begin{aligned}
\widehat{v}(P) &= \widehat{v}(x_1 \vee x_2 \vee x_3) \wedge \widehat{v}(\overline{x_1} \vee x_2) \wedge \widehat{v}(\overline{x_2} \vee x_3) \wedge \widehat{v}(\overline{x_3} \vee x_1) \wedge \widehat{v}(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \\
&= \mathbf{T} \wedge \mathbf{F} \wedge \mathbf{T} \wedge \mathbf{T} \wedge \mathbf{T} = \mathbf{F}.
\end{aligned}$$

Therefore, $\widehat{v}(P) = \mathbf{F}$.

Definition 9.6. We say that a truth assignment v *satisfies* a boolean formula P, if $\widehat{v}(P) = \mathbf{T}$. In this case, we also write

$$v \models P.$$

A boolean formula P is *satisfiable* if $v \models P$ for some truth assignment v, otherwise, it is *unsatisfiable*. A boolean formula P is *valid (or a tautology)* if $v \models P$ for all truth assignments v, in which case we write

$$\models P.$$

One should check that the boolean formula

$$P = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}$$

is **un**satisfiable.

One may think that it is easy to test whether a proposition is satisfiable or not. Try it, it is not that easy!

As a matter of fact, the *satisfiability problem*, testing whether a boolean formula is satisfiable, also denoted SAT, is not known to be in $\mathcal{P}$. Moreover, it is an $\mathcal{NP}$-complete problem (see Section 9.6). Most people believe that the satisfiability problem is **not** in $\mathcal{P}$, but a proof still eludes us!

Before we explain what is the class $\mathcal{NP}$, we state the following result.

Proposition 9.4. *The satisfiability problem for clauses containing at most two literals (2-satisfiability, or* 2-SAT*) is solvable in polynomial time.*

Proof sketch. The first step consists in observing that if every clause in P contains at most two literals, then we can reduce the problem to testing satisfiability when every clause has exactly two literals.

Indeed, if P contains some clause (x), then any valuation satisfying P must make x true. Then all clauses containing x will be true, and we can delete them, whereas we can delete $\overline{x}$ from every clause containing it, since $\overline{x}$ is false.

Similarly, if P contains some clause $(\overline{x})$, then any valuation satisfying P must make x false. Then all clauses containing $\overline{x}$ will be true, and we can delete them, whereas we can delete x from every clause containing it.

Thus in a finite number of steps, either all the clauses were satisfied and P is satisfiable, or we get the empty clause and P is unsatisfiable, or we get a set of clauses with exactly two literals. The number of steps is clearly linear in the number of literals in P. Here are some examples illustrating the three possible oucomes.

(1) Consider the conjunction of clauses

$$P_1 = (x_1 \vee \overline{x_2}) \wedge (x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee \overline{x_3}).$$

We must set x_2 to $\mathbf{T}$, so $(x_1 \vee \overline{x_2})$ becomes (x_1) and $(x_2 \vee \overline{x_3})$ becomes $\mathbf{T}$ and can be deleted. We now have

$$P = (x_1) \wedge (x_1 \vee x_3).$$

We must set x_1 to $\mathbf{T}$, so $(x_1 \vee x_3)$ becomes $\mathbf{T}$ and all the clauses are satisfied.

(2) Consider the conjunction of clauses

$$P_2 = (x_1) \wedge (x_3) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3}).$$

We must set x_1 to $\mathbf{T}$, so $(\overline{x_1} \vee x_2)$ becomes (x_2). We now have

$$(x_3) \wedge (x_2) \wedge (\overline{x_2} \vee \overline{x_3}).$$

We must set x_3 to $\mathbf{T}$, so $(\overline{x_2} \vee \overline{x_3})$ becomes $(\neg x_2)$. We now have

$$(x_2) \wedge (\overline{x_2}).$$

We must set x_2 to $\mathbf{T}$, so $(\overline{x_2})$ becomes the empty clause, which means that P_2 is unsatisfiable.

For the second step, we construct a directed graph from P. The purpose of this graph is to propagate truth. The nodes of this graph are the literals in P, and edges are defined as follows:

(1) For every clause $(\overline{x} \vee y)$, there is an edge from x to y and an edge from $\overline{y}$ to $\overline{x}$.

(2) For every clause $(x \vee y)$, there is an edge from $\overline{x}$ to y and an edge from $\overline{y}$ to x

(3) For every clause $(\overline{x} \vee \overline{y})$, there is an edge from x to $\overline{y}$ and an edge from y to $\overline{x}$.

Then it can be shown that P is unsatisfiable iff there is some x so that there is a cycle containing x and $\overline{x}$. As a consequence, 2-satisfiability is in $\mathcal{P}$. □

Example 9.4. Consider the following conjunction of clauses:

$$P = (x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}).$$

It is satisfied by any valuation v such that $v(x_1) = \mathbf{T}$, and if $v(x_2) = \mathbf{F}$ then $v(x_3) = \mathbf{F}$. The construction of the graph associated with P is shown in Figure 9.4.

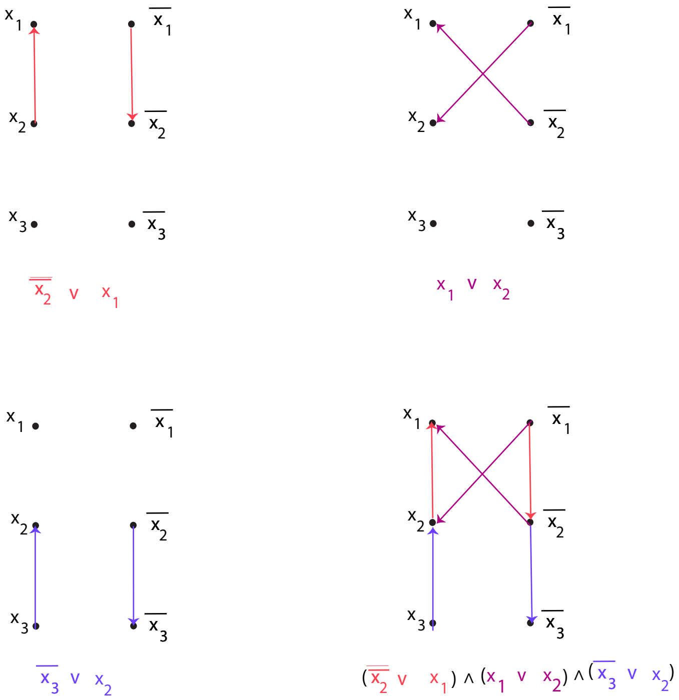


Figure 9.4: The graph coresponding to the clauses of Example 9.4.

9.6 The Class $\mathcal{NP}$, Polynomial Reducibility, $\mathcal{NP}$-Completeness

One will observe that the hard part in trying to solve either the Hamiltonian cycle problem or the satisfiability problem, SAT, is to *find* a solution, but that *checking* that a candidate solution is indeed a solution can be done easily in polynomial time.

This is the essence of problems that can be solved *nondetermistically* in polynomial time: a solution can be guessed and then checked in polynomial time.

Definition 9.7. A nondeterministic Turing machine M is said to be *polynomially bounded* if there is a polynomial $p(X)$ so that the following holds: For every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is *nondeterministic polynomially decidable* if there is a polynomially bounded nondeterministic Turing machine that accepts L. The family of all nondeterministic polynomially decidable languages is denoted by $\mathcal{NP}$.

Observe that Definition 9.7 has to do with testing *membership* of a string w in a language L. Here the language L consists of the strings encodings all objects satisfying a given property P. So in this sense, a reason (a certificate) why $w \in L$ is not actually produced by the machine. The machine just decides whether $w \in L$, that is, whether the object coded by w satisfies the property P.

For example, if the problem is the satisfiability of sets of clauses, then L is the set SAT of strings encoding all satisfiable propositions in CNF. Given any proposition P in CNF encoded as a string $s(P)$, a Turing machine accepting SAT will nondeterminiscally guess a truth assignment, and check in polynomial time whether this truth assignment satisfies P.

In the case of clauses we can easily design such a language. The key point is that we can represent the propositional variable x_i as a string in binary, namely as the binary representation $bin(x_i)$ of the number i. Our language for encoding clauses uses the alphabet

$$\Delta = \{0, 1, \wedge, \vee, \neg, (,)\}.$$

The encoding $s(P)$ (a string in Δ^*) of a proposition P in CNF is defined recursively as follows.

(1) The variable x_i is represented the binary representation $s(x_i) = bin(i)$ of the number i.

(2) The literal $\neg x_i$ is represented by the string $s(\neg x_i) = \neg s(x_i) = \neg bin(i)$.

(3) The clause

$$C = (L_1 \vee \cdots \vee L_m)$$

is represented by the string

$$s(C) = (s(L_1) \vee \cdots \vee s(L_m)).$$

(3) The proposition P in CNF

$$P = C_1 \wedge \cdots \wedge C_p$$

is represented by the string

$$s(P) = s(C_1) \wedge \cdots \wedge s(C_p).$$

Example 9.5. The proposition

$$P = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3)$$

is encoded by the string

$$s(P) = (1 \vee 10 \vee 11) \wedge (\neg 1 \vee \neg 10) \wedge (1 \vee \neg 11) \wedge (10 \vee 11).$$

If we assign the truth value **F** to x_1, to satisfy the clause $(x_1 \vee \neg x_3)$ we must assign **F** to x_3, and then to satisfy the clauses $(x_1 \vee x_2 \vee x_3)$ and $(x_2 \vee x_3)$, we must assign **T** to x_2.

If we assign the truth value **T** to x_1, to satisfy the clause $(\neg x_1 \vee \neg x_2)$ we must assign **F** to x_2, and then to satisfy the clause $(x_2 \vee x_3)$, we must assign **T** to x_3.

Therefore there are two truth assignments satisfying the proposition P,

$$x_1 := \mathbf{F},\ x_2 := \mathbf{T},\ x_3 := \mathbf{F}$$
$$x_1 := \mathbf{T},\ x_2 := \mathbf{F},\ x_3 := \mathbf{T}.$$

The language SAT $\subseteq \Delta^*$ consists of all string encodings $s(P)$ of propositions that are satisfiable. For example, the string

$$s(P) = (1 \vee 10 \vee 11) \wedge (\neg 1 \vee \neg 10) \wedge (1 \vee \neg 11) \wedge (10 \vee 11)$$

belongs to the language SAT. On the other hand, the proposition

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_1 \vee \neg x_2)$$

is not satisfiable, and thus its encoding

$$(1 \vee 10) \wedge (\neg 1 \vee 10) \wedge (\neg 10 \vee 1) \wedge (\neg 1 \vee \neg 10)$$

does *not* belong to SAT.

Remark: The language consisting of all string encodings of propositions in CNF, satisfiable or not, is a context-free language.

Note that a nondeterminitsic Turing machine operating in polynomial time accepting a string in SAT encoding a satisfiable clause does not actually produce a truth assignment, called a certificate, as output. The machine simply accepts or rejects $s(P)$ depending on whether P is satisfiable or not.

Similarly, if the problem is the existence of a Hamiltonian cycle, then L is the set of strings encoding all directed graphs having a Hamiltonian cycle. Given any directed graph G encoded as a string $s(G)$, a Turing machine accepting L will nondeterminiscally guess a cycle in G, and check in polynomial time whether this is a Hamiltonian cycle. But such a Hamiltonian cycle (if any), called a certificate, is not actually produced as output.

Here is a way to encode a simple directed graph $G = (V, E)$. A slight complication arises with *isolated nodes*, which are the nodes $u \in V$ such that there is no edge $(u, v) \in E$ or $(v, u) \in E$ for some $v \in V$, in other words, the nodes that are not the endpoint of any edge.

If $V = \{v_1, \ldots, v_n\}$, as in the case of clauses, we encode the node v_i as the binary representation $s(v_i) = bin(i)$ of the number i. We use alphabet

$$\Delta = \{0, 1, \rightarrow, (,), \#\}.$$

The string encoding $s(G)$ of the graph $G = (V, E)$ is obtained by concatenating the strings $(s(v_i) \rightarrow s(v_j))$ in the order where $(s(v_i) \rightarrow s(v_j))$ precedes $(s(v_k) \rightarrow s(v_l))$ if either $i = k$ and $j < l$, or $i < k$, possibly followed by the string

$$\#s(v_{i_1})\# \cdots \#s(v_{i_k})$$

corresponding to the the isolated vertices, if any, where $v_{i_1}, \ldots, v_{i_k}$ are listed in increasing order of the indices.

Example 9.6. Consider the graph $G = (V, E)$ shown in Figure 9.5 where $V = \{v_1, \ldots, v_5\}$ consists of five nodes and the set of edges is

$$E = \{(v_1, v_2),\ (v_1, v_4),\ (v_1, v_5),\ (v_2, v_3),\ (v_2, v_4),\ (v_3, v_1),\ (v_3, v_4),\ (v_4, v_5),\ (v_5, v_3)\}.$$

The string encoding of this graph is

$$\begin{aligned} s(G) = (1 \rightarrow 10)(1 \rightarrow 100)(1 \rightarrow 101)(10 \rightarrow 11)(10 \rightarrow 100)(11 \rightarrow 1)(11 \rightarrow 100) \\ (100 \rightarrow 101)(101 \rightarrow 11). \end{aligned}$$

Observe that the cycle $v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_5 \rightarrow v_3 \rightarrow v_1$ is a Hamiltonian cycle.

The language HAM $\subseteq \Delta^*$ consists of all encodings $s(G)$ of directed graphs G that have a Hamiltonian cycle. Thus for the graph above, $s(G) \in$ HAM.

It is possible to give an alternate definition of $\mathcal{NP}$ that explicitly involves certificates. This definition relies on the notion of a polynomially balanced language; see Section 10.3, Definition 10.3. The trick is to consider strings of form $x; y \in \Sigma^*$ (with $x, y \in \Sigma^*$, where ; is a special symbol not in Σ), such that for some given polynomial $p(X)$, we have $|y| \leq p(|x|)$.

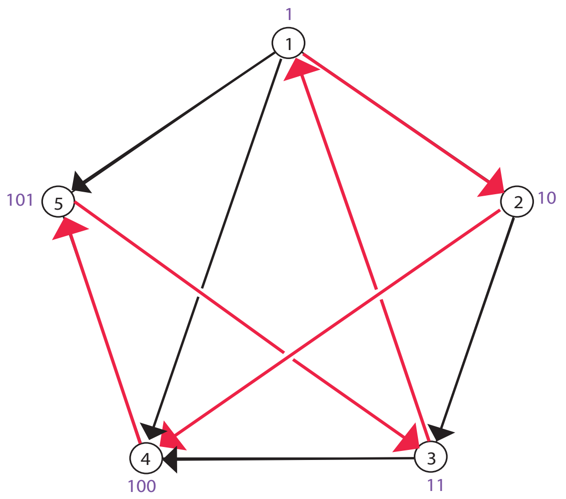


Figure 9.5: A pentagonal graph with a Hamiltonian cycle.

If a language L' consisting of strings of the form $x; y$ with $|y| \leq p(|x|)$ (for some given p) is in $\mathcal{P}$, then the language

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}$$

is in $\mathcal{NP}$, and every language in $\mathcal{NP}$ arises in this fashion; see Theorem 10.1. The set of strings $\{y \in \Sigma^* \mid x; y \in L'\}$ can be regarded as the set of certificates for the fact that $x \in L$. The fact that $|y| \leq p(|x|)$ ensures that the certificate y is not too big, so that L' can be accepted *deterministically* in polynomial time. We will come back to this point of view in Section 10.3.

For example, going back to Example 9.5, examples of strings $x; y$ are

$$(1 \lor 10 \lor 11) \land (\neg 1 \lor \neg 10) \land (1 \lor \neg 11) \land (10 \lor 11); \mathbf{FTF}$$

and

$$(1 \lor 10 \lor 11) \land (\neg 1 \lor \neg 10) \land (1 \lor \neg 11) \land (10 \lor 11); \mathbf{TFT}.$$

This time, a *deterministic* Turing machine accepts such strings in polynomial time by checking that the certificates **FTF** or **TFT** satisfy the proposition.

For Example 9.6 dealing with Hamiltonian cycles, here is an example of a string $x; y$ where the certificate y is an encoding of a Hamiltonian cycle:

$$\begin{gathered}(1 \to 10)(1 \to 100)(1 \to 101)(10 \to 11)(10 \to 100)(11 \to 1)(11 \to 100)\\(100 \to 101)(101 \to 11); 1 \to 10 \to 100 \to 101 \to 11 \to 1.\end{gathered}$$

Returning to the definition of $\mathcal{NP}$ given in Definition 9.7, of course, we have the inclusion

$$\mathcal{P} \subseteq \mathcal{NP},$$

but whether or not we have equality is one of the most famous open problems of theoretical computer science and mathematics.

In fact, the question $\mathcal{P} \neq \mathcal{NP}$ is one of the open problems listed by the CLAY Institute, together with the Poincaré conjecture and the Riemann hypothesis, among other problems, and for which *one million dollar* is offered as a reward! Actually the Poincaré conjecture was setlled by G. Perelman in 2006, but he rejected receiving the prize in 2010! He also declined the Fields Medal which was awarded to him in 2006.

It is easy to check that SAT is in $\mathcal{NP}$, and so is the Hamiltonian cycle problem.

As we saw in recursion theory, where we introduced the notion of many-one reducibility, in order to compare the "degree of difficulty" of problems, it is useful to introduce the notion of reducibility and the notion of a complete set.

Definition 9.8. A function $f\colon \Sigma^* \to \Sigma^*$ is *polynomial-time computable* if there is a polynomial $p(X)$ so that the following holds: there is a deterministic Turing machine M computing it so that for every input $x \in \Sigma^*$, there is no ID ID_n so that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),$$

where $ID_0 = q_0x$ is the starting ID.

Given two languages $L_1, L_2 \subseteq \Sigma^*$, *a polynomial-time reduction from L_1 to L_2* is a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ so that for all $u \in \Sigma^*$,

$$u \in L_1 \quad \text{iff} \quad f(u) \in L_2.$$

The notation $L_1 \leq_P L_2$ is often used to denote the fact that there is polynomial-time reduction from L_1 to L_2. Sometimes, the notation $L_1 \leq_m^P L_2$ is used to stress that this is a many-to-one reduction (that is, f is not necessarily injective). This type of reduction is also known as a *Karp reduction*.

A polynomial reduction $f\colon \Sigma^* \to \Sigma^*$ from a language L_1 to a language L_2 is a method that converts in polynomial time every string $u \in \Sigma^*$ (viewed as an instance of a problem A encoded by language L_1) to a string $f(u) \in \Sigma^*$ (viewed as an instance of a problem B encoded by language L_2) in such way that membership in L_1, that is $u \in L_1$, is equivalent to membership in L_2, that is $f(u) \in L_2$.

As a consequence, if we have a procedure to decide membership in L_2 (to solve every instance of problem B), then we have a procedure for solving membership in L_1 (to solve every instance of problem A), since given any $u \in L_1$, we can first apply f to u to produce $f(u)$, and then apply our procedure to decide whether $f(u) \in L_2$; the defining property of f says that this is equivalent to deciding whether $u \in L_1$. Furthermore, if the procedure for deciding membership in L_2 runs deterministically in polynomial time, since f runs deterministically in polynomial time, so does the procedure for deciding membership in L_1, and similarly if the procedure for deciding membership in L_2 runs non deterministically in polynomial time.

For the above reason, we see that membership in L_2 can be considered at least as hard as membership in L_1, since any method for deciding membership in L_2 yields a method for deciding membership in L_1. Thus, if we view L_1 an encoding a problem A and L_2 as encoding a problem B, then B is at least as hard as A.

The following version of Proposition 5.16 for polynomial-time reducibility is easy to prove.

Proposition 9.5. *Let A, B, C be subsets of $\mathbb{N}$ (or Σ^*). The following properties hold:*

(1) If $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

(2) If $A \leq_P B$ then $\overline{A} \leq_P \overline{B}$.

(3) If $A \leq_P B$ and $B \in \mathcal{NP}$, then $A \in \mathcal{NP}$.

(4) If $A \leq_P B$ and $A \notin \mathcal{NP}$, then $B \notin \mathcal{NP}$.

(5) If $A \leq_P B$ and $B \in \mathcal{P}$, then $A \in \mathcal{P}$.

(6) If $A \leq_P B$ and $A \notin \mathcal{P}$, then $B \notin \mathcal{P}$.

Intuitively, we see that if L_1 is a hard problem and L_1 can be reduced to L_2 in polynomial time, then L_2 is also a hard problem.

For example, one can construct a polynomial reduction *from the Hamiltonian cycle problem to the satisfiability problem* SAT. Given a directed graph $G = (V, E)$ with n nodes, say $V = \{1, \ldots, n\}$, we need to construct in polynomial time a set $F = \tau(G)$ of clauses such that G has a Hamiltonian cycle iff $\tau(G)$ is satisfiable. We need to describe a permutation of the nodes that forms a Hamiltonian cycle. For this we introduce n^2 boolean variables x_{ij}, with the intended interpretation that *x_{ij} is true iff node i is the jth node in a Hamiltonian cycle.*

To express that at least one node must appear as the jth node in a Hamiltonian cycle, we have the n clauses

$$(x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj}), \quad 1 \leq j \leq n. \tag{1}$$

The conjunction of these clauses is satisfied iff for every $j = 1, \ldots, n$ there is some node i which is the jth node in the cycle. These n clauses can be produced in time $O(n^2)$.

To express that only one node appears in the cycle, we have the clauses

$$(\overline{x_{ij}} \vee \overline{x_{kj}}), \quad 1 \leq i, j, k \leq n,\ i \neq k. \tag{2}$$

Since $(\overline{x_{ij}} \vee \overline{x_{kj}})$ is equivalent to $\overline{(x_{ij} \wedge x_{kj})}$, each such clause asserts that no two distinct nodes may appear as the jth node in the cycle. Let S_1 be the set of all clauses of type (1) or (2). These n^3 clauses can be produced in time $O(n^3)$.

The conjunction of the clauses in S_1 assert that exactly one node appear at the jth node in the Hamiltonian cycle. We still need to assert that each node i appears exactly once in the cycle. For this, we have the clauses

$$(x_{i1} \vee x_{i2} \vee \cdots \vee x_{in}), \quad 1 \leq i \leq n, \tag{3}$$

and

$$(\overline{x_{ij}} \vee \overline{x_{ik}}), \quad 1 \leq i, j, k \leq n,\ j \neq k. \tag{4}$$

Let S_2 be the set of all clauses of type (3) or (4). These n^3 clauses can be produced in time $O(n^3)$.

The conjunction of the clauses in $S_1 \cup S_2$ asserts that the x_{ij} represents a bijection of $\{1, 2, \ldots, n\}$, in the sense that for any truth assigment v satisfying all these clauses, $i \mapsto j$ iff $v(x_{ij}) = \mathbf{T}$ defines a bijection of $\{1, 2, \ldots, n\}$.

It remains to assert that this permutation of the nodes is a Hamiltonian cycle, which means that if x_{ij} and x_{kj+1} are both true then there there must be an edge (i, k). By contrapositive, this equivalent to saying that if (i, k) is *not* an edge of G, then $\overline{(x_{ij} \wedge x_{kj+1})}$ is true, which as a clause is equivalent to $(\overline{x_{ij}} \vee \overline{x_{kj+1}})$.

Therefore, for all (i, k) such that $(i, k) \notin E$ (with $i, k \in \{1, 2, \ldots, n\}$), we have the clauses

$$(\overline{x_{ij}} \vee \overline{x_{k\,j+1\ (\mathrm{mod}\ n)}}), \quad j = 1, \ldots, n. \tag{5}$$

Let S_3 be the set of clauses of type (5). These n clauses can be produced in time $O(n^2)$.

The conjunction of all the clauses in $S_1 \cup S_2 \cup S_3$ is the boolean formula $F = \tau(G)$. It can be produced in time $O(n^3)$.

We leave it as an exercise to prove that G has a Hamiltonian cycle iff $F = \tau(G)$ is satisfiable.

Example 9.7. Here is an example of a graph with four nodes and four edges shown in Figure 9.6. The Hamiltonian circuit is (x_4, x_3, x_1, x_2).

It is also possible to construct a reduction of the satisfiability problem to the Hamiltonian cycle problem but this is harder. It is easier to construct this reduction in two steps by introducing an intermediate problem, the exact cover problem, and to provide a polynomial reduction from the satisfiability problem to the exact cover problem, and a polynomial reduction from the exact cover problem to the Hamiltonian cycle problem. These reductions are carried out in Section 10.2.

The above construction of a set $F = \tau(G)$ of clauses from a graph G asserting that G has a Hamiltonian cycle iff F is satisfiable illustrates the expressive power of propositional logic.

Remarkably, *every* language in $\mathcal{NP}$ can be reduced to SAT. Thus, SAT is a hardest language in $\mathcal{NP}$ (since it is in $\mathcal{NP}$).

Definition 9.9. A language L is *$\mathcal{NP}$-hard* if there is a polynomial reduction from every language $L_1 \in \mathcal{NP}$ to L. A language L is *$\mathcal{NP}$-complete* if $L \in \mathcal{NP}$ and L is $\mathcal{NP}$-hard.

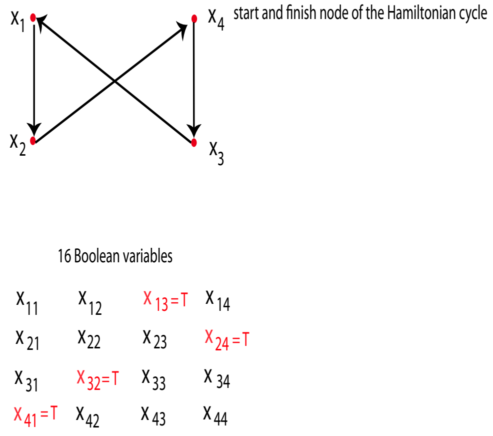

x_1
x_4
start and finish node of the Hamiltonian cycle
x_2
x_3
16 Boolean variables
x_{11} x_{12} $x_{13}=T$ x_{14}
x_{21} x_{22} x_{23} $x_{24}=T$
x_{31} $x_{32}=T$ x_{33} x_{34}
$x_{41}=T$ x_{42} x_{43} x_{44}

Figure 9.6: A directed graph with a Hamiltonian

Thus, an $\mathcal{NP}$-hard language is as hard to decide as any language in $\mathcal{NP}$.

Remark: There are $\mathcal{NP}$-hard languages that do not belong to $\mathcal{NP}$. Such languages are really difficult. A standard example is K_0, which encodes the halting problem. Since K_0 is not computable, it can't be in $\mathcal{NP}$. Furthermore, since every language L in $\mathcal{NP}$ is accepted nondeterminsticaly in polynomial time $p(X)$, for some polynomial $p(X)$, for every input w we can try all computations of length at most $p(|w|)$ (there can be exponentially many, but only a finite number), so every language in $\mathcal{NP}$ is computable. Finally, a Turing machine which takes a clause as input and tries all possible truth assignments and loops iff there is no satisfying assignment can be constructed. We can use this machine to show that 3-SAT can be reduced in polynomial time to K_0, the details are left as an exercise. Since K_0 is defined in terms of natural numbers and not strings, we need to assume that boolean propositions are first encoded as natural numbers and that our Turing machine for testing satisfiability operates on such numbers. Such a machine may not run in polynomial time because of the steps needed for decoding but this does not matter. What is important is that the reduction works in polynomnial time. An example of a computable $\mathcal{NP}$-hard language not in $\mathcal{NP}$ will be described after Theorem 9.7.

The importance of $\mathcal{NP}$-complete languages stems from the following theorem which follows immediately from Proposition 9.5.

Theorem 9.6. *Let L be an $\mathcal{NP}$-complete language. Then $\mathcal{P} = \mathcal{NP}$ iff $L \in \mathcal{P}$.*

There are analogies between $\mathcal{P}$ and the class of computable sets, and $\mathcal{NP}$ and the class of listable sets, but there are also important differences. One major difference is that the family of computable sets is properly contained in the family of listable sets, but it is an open problem whether $\mathcal{P}$ is properly contained in $\mathcal{NP}$. We also know that a set L is computable iff both L and $\overline{L}$ are listable, but it is also an open problem whether if both $L \in \mathcal{NP}$ and $\overline{L} \in \mathcal{NP}$, then $L \in \mathcal{P}$. This suggests defining

$$\mathrm{co}\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\},$$

that is, co$\mathcal{NP}$ consists of all complements of languages in $\mathcal{NP}$. Since $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{P}$ is closed under complementation,

$$\mathcal{P} \subseteq \mathrm{co}\mathcal{NP},$$

and thus

$$\mathcal{P} \subseteq \mathcal{NP} \cap \mathrm{co}\mathcal{NP},$$

but nobody knows whether the inclusion is proper. There are languages in $\mathcal{NP} \cap \mathrm{co}\mathcal{NP}$ not known to be in $\mathcal{P}$; see Section 10.3. It is unknown whether $\mathcal{NP}$ is closed under complementation, that is, nobody knows whether $\mathcal{NP} = \mathrm{co}\mathcal{NP}$. This is considered unlikely. We will come back to co$\mathcal{NP}$ in Section 10.3.

Next we prove a famous theorem of Steve Cook and Leonid Levin (proven independently): SAT is $\mathcal{NP}$-complete.

9.7 The Bounded Tiling Problem is $\mathcal{NP}$-Complete

Instead of showing directly that SAT is $\mathcal{NP}$-complete, which is rather complicated, we proceed in two steps, as suggested by Lewis and Papadimitriou.

(1) First, we define a tiling problem adapted from H. Wang (1961) by Harry Lewis, and we prove that it is $\mathcal{NP}$-complete.

(2) We show that the tiling problem can be reduced to SAT.

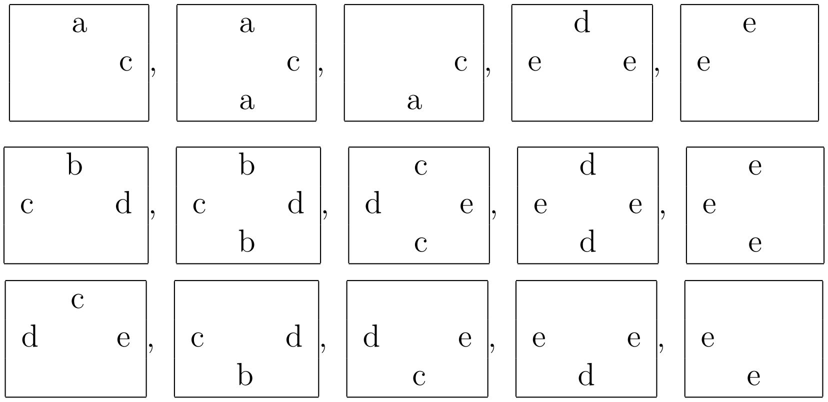
We are given a finite set $\mathcal{T} = \{t_1, \ldots, t_p\}$ of *tile patterns*, for short, *tiles*. We assume that these tiles are unit squares. Copies of these tile patterns may be used to tile a rectangle of predetermined size $2s \times s$ ($s > 1$). However, there are constraints on the way that these tiles may be adjacent horizontally and vertically.

The *horizontal constraints* are given by a relation $H \subseteq \mathcal{T} \times \mathcal{T}$, and the *vertical constraints* are given by a relation $V \subseteq \mathcal{T} \times \mathcal{T}$.

Thus, a *tiling system* is a triple $T = (\mathcal{T}, V, H)$ with V and H as above.

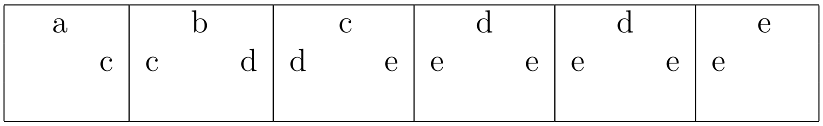
The bottom row of the rectangle of tiles is specified before the tiling process begins.

Example 9.8. For example, consider the following tile patterns:

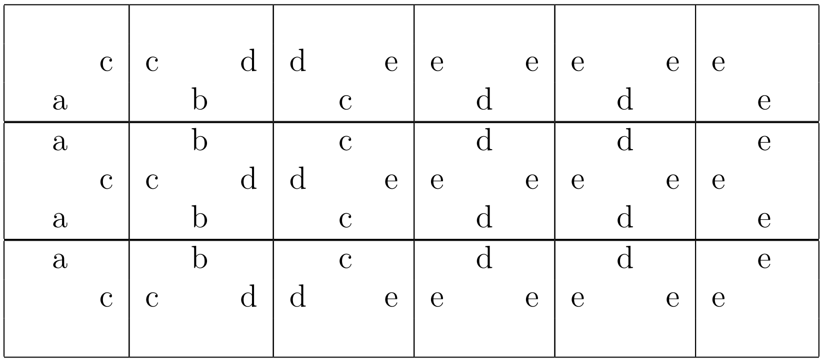


The horizontal and the vertical constraints are that the letters on adjacent edges match (blank edges do not match).

For $s = 3$, given the bottom row



we have the tiling shown below:



Formally, the problem is then as follows:

The Bounded Tiling Problem

Given any tiling system $(\mathcal{T}, V, H)$, any integer $s > 1$, and any initial row of tiles σ_0 (of length $2s$)

$$\sigma_0 \colon \{1, 2, \ldots, s, s+1, \ldots, 2s\} \to \mathcal{T},$$

find a $2s \times s$-tiling σ extending σ_0, i.e., a function

$$\sigma \colon \{1, 2, \ldots, s, s+1, \ldots, 2s\} \times \{1, \ldots, s\} \to \mathcal{T}$$

so that

(1) $\sigma(m,1) = \sigma_0(m)$, for all m with $1 \leq m \leq 2s$.

(2) $(\sigma(m,n), \sigma(m+1,n)) \in H$, for all m with $1 \leq m \leq 2s-1$, and all n, with $1 \leq n \leq s$.

(3) $(\sigma(m,n), \sigma(m,n+1)) \in V$, for all m with $1 \leq m \leq 2s$, and all n, with $1 \leq n \leq s-1$.

Example 9.9. In this example the set of tiles is shown in Figure 9.7. The horizontal constraints are schematically illustrated in Figure 9.8

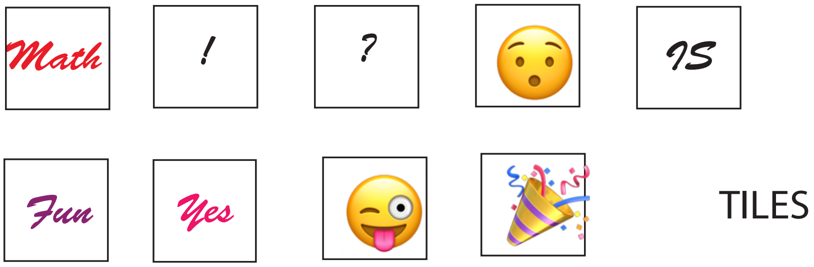


Figure 9.7: A set of tiles

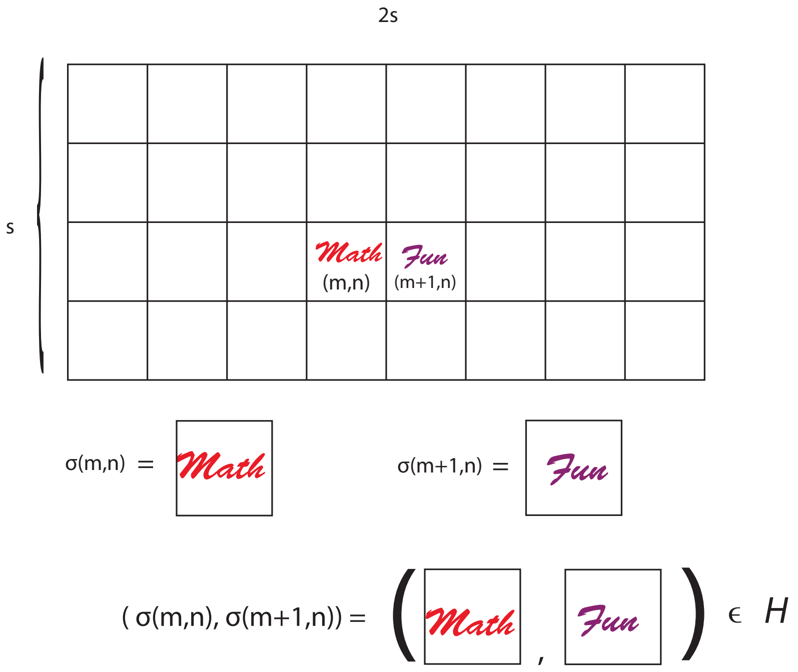


Figure 9.8: Schematic illustration of the horizontal constraints.

and the vertical constraints are are schematically illustrated in Figure 9.9. The set of

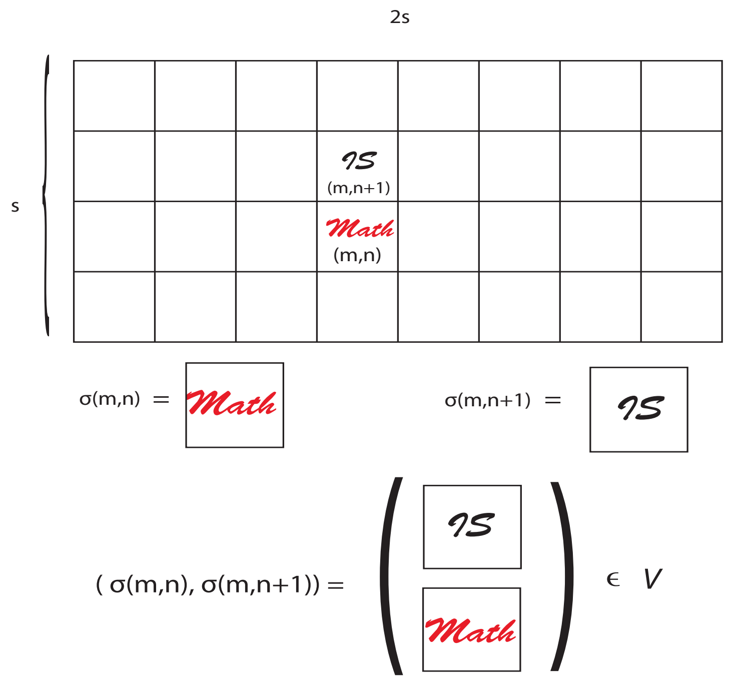


Figure 9.9: Schematic illustration of the vertical constraints.

horizontal constraints is shown in Figure 9.10

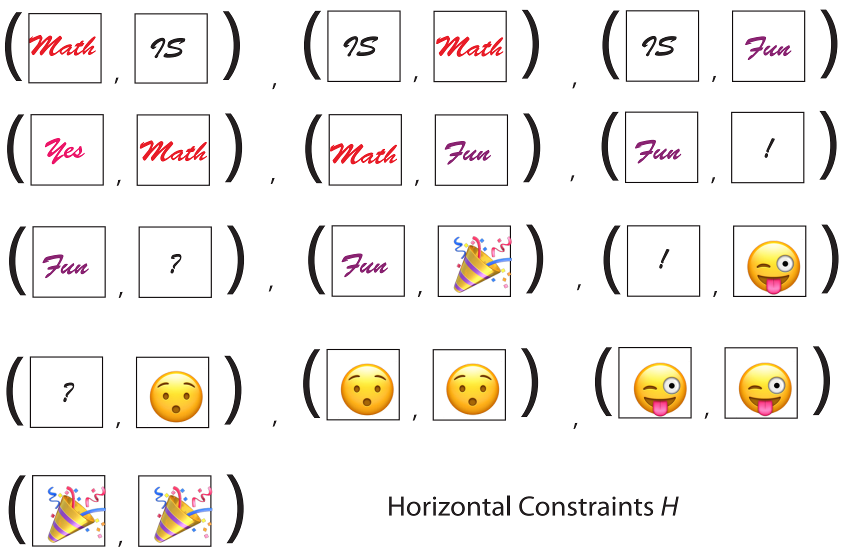


Figure 9.10: Horizontal constraints.

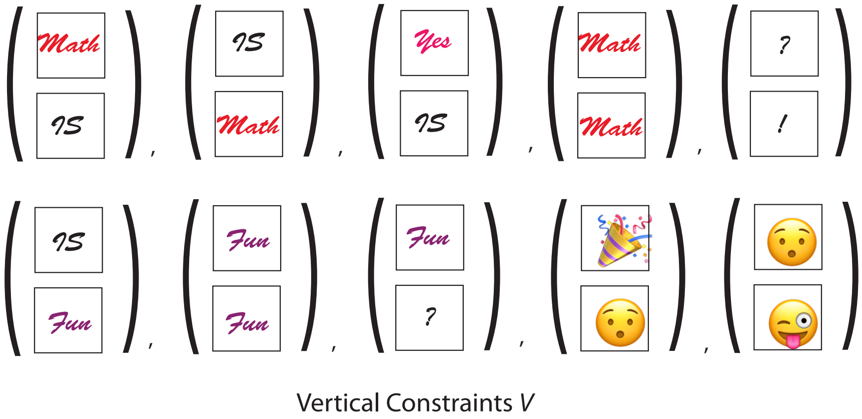
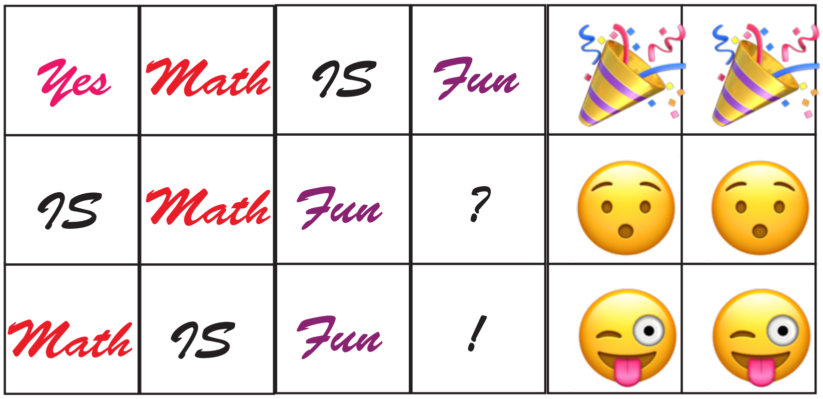


Figure 9.11: Vertical constraints.

and the set of vertical constraints is shown in Figure. 9.11 A solution to the puzzle (tiling problem) is shown in Figure 9.12, assuming that the bottom row is given as part of the input.



Yes	Math	IS	Fun	🎉	🎉
IS	Math	Fun	?	😯	😯
Math	IS	Fun	!	😜	😜

Figure 9.12: A solution to the tiling problem.

Formally, an *instance of the tiling problem* is a triple $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, where $(\mathcal{T}, V, H)$ is a tiling system, $\widehat{s}$ is the string representation of the number $s \geq 2$, in binary and σ_0 is an initial row of tiles (the bottom row).

For example, if $s = 1025$ (as a decimal number), then its binary representation is $\widehat{s} = 10000000001$. The length of $\widehat{s}$ is $\log_2 s + 1$.

Recall that the input must be a string. This is why the number s is represented by a string in binary. If we only included a *single* tile σ_0 in position $(s + 1, 1)$, then the length of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$ would be $\log_2 s + 1 + C + 1 = \log_2 s + C + 2$ for some constant C corresponding to the length of the string encoding $(\mathcal{T}, V, H)$.

However, the rectangular grid has size $2s^2$, which is *exponential* in the length $\log_2 s + C + 2$ of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$. Thus, it is impossible to check in polynomial time that a proposed solution is a tiling.

However, if we include in the input the bottom row σ_0 of length $2s$, then the length of input is $\log_2 s + 1 + C + 2s = \log_2 s + C + 2s + 1$ and the size $2s^2$ or the grid is indeed polynomial in the size of the input.

Theorem 9.7. *The tiling problem defined earlier is $\mathcal{NP}$-complete.*

Proof. Let $L \subseteq \Sigma^*$ be any language in $\mathcal{NP}$ and let u be any string in Σ^*. Assume that L is accepted in polynomial time bounded by $p(|u|)$.

We show how to construct an instance of the tiling problem, $((\mathcal{T}, V, H)_L, \widehat{s}, \sigma_0)$, where $s = p(|u|) + 2$, and where the bottom row encodes the starting ID, so that $u \in L$ iff the tiling problem $((\mathcal{T}, V, H)_L, \widehat{s}, \sigma_0)$ has a solution.

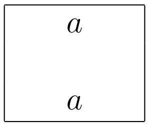
First, note that the problem is indeed in $\mathcal{NP}$, since we have to guess a rectangle of size $2s^2$, and that checking that a tiling is legal can indeed be done in $O(s^2)$, where s is *bounded by the the size of the input* $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, since the input contains the bottom row of $2s$ symbols (this is the reason for including the bottom row of $2s$ tiles in the input!).

The idea behind the definition of the tiles is that, in a solution of the tiling problem, the labels on the horizontal edges between two adjacent rows represent a legal ID, $xpay$. In a given row, the labels on vertical edges of adjacent tiles keep track of the change of state and direction.

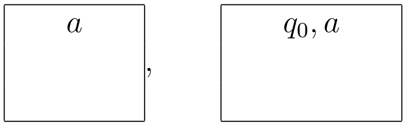
Let Γ be the tape alphabet of the TM, M. As before, we assume that M signals that it accepts u by halting with the output 1 (true).

From M, we create the following tiles:

(1) For every $a \in \Gamma$, tiles

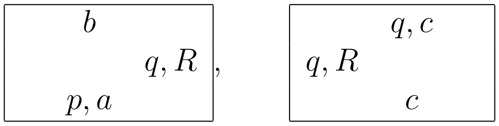


(2) For every $a \in \Gamma$, the bottom row uses tiles

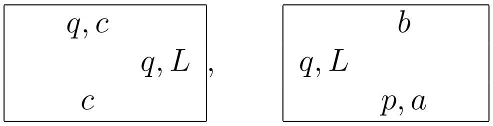


where q_0 is the start state.

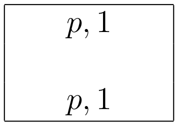
(3) For every instruction $(p, a, b, R, q) \in \delta$, for every $c \in \Gamma$, tiles



(4) For every instruction $(p, a, b, L, q) \in \delta$, for every $c \in \Gamma$, tiles



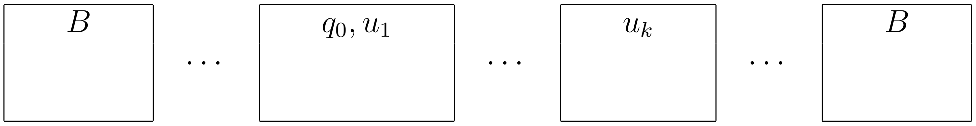
(5) For every halting state, p, tiles



The purpose of tiles of type (5) is to fill the $2s \times s$ rectangle iff M accepts u. Since $s = p(|u|) + 2$ and the machine runs for at most $p(|u|)$ steps, the $2s \times s$ rectangle can be tiled iff $u \in L$.

The vertical and the horizontal constraints are that adjacent edges have the same label (or no label).

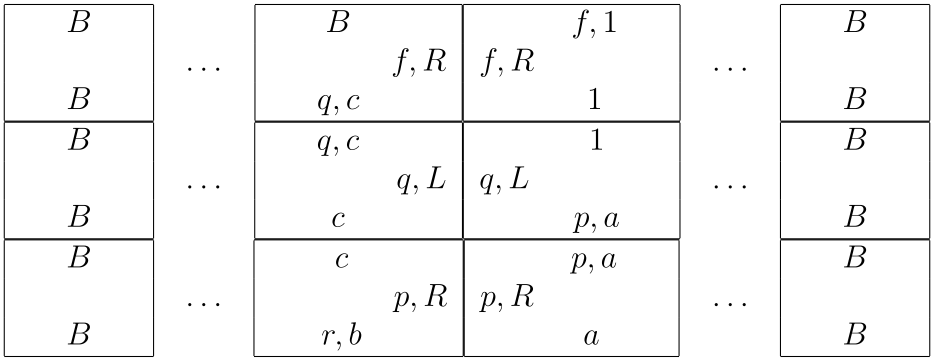
If $u = u_1 \cdots u_k$, the initial bottom row σ_0, of length $2s$, is



where the tile labeled q_0, u_1 is in position $s + 1$.

The example below illustrates the construction:

Example 9.10.



We claim that $u = u_1 \cdots u_k$ is accepted by M iff the tiling problem just constructed has a solution.

The upper horizontal edge of the first (bottom) row of tiles represents the starting configuation $B^s q_0 u B^{s-|u|}$. By induction, we see that after i ($i \leq p(|u|) = s - 2$) steps the upper horizontal edge of the $(i+1)$th row of tiles represents the current ID $xpay$ reached by the Turing machine; see Example 9.10. Since the machine runs for at most $p(|u|)$ steps and since $s = p(|u|) + 2$, when the computation stops, at most the lowest $p(|u|) + 1 = s - 1$ rows of the the $2s \times s$ rectangle have been tiled. Assume the machine M stops after $r \leq s - 2$ steps. Then the lowest $r + 1$ rows have been tiled, and since no further instruction can be executed (since the machine entered a halting state), the remaining $s - r - 1$ rows can be filled iff tiles of type (5) can be used iff the machine stopped in an ID containing a pair $p\,1$ where p is a halting state. Therefore, the machine M accepts u iff the $2s \times s$ rectangle can be tiled. □

Remark:

(1) The problem becomes harder if we only specify a *single* tile σ_0 as input, instead of a row of length $2s$. If s is specified in binary (or any other base, but not in tally notation), then the $2s^2$ grid has size exponential in the length $\log_2 s + C + 2$ of the input $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, and this tiling problem is actually $\mathcal{NEXP}$-complete! The class $\mathcal{NEXP}$ is the family of languages that can be accepted by a nondeterministic Turing machine that runs in time bounded by $2^{p(|x|)}$, for every x, where p is a polynomial; see the remark after Definition 10.5. By the time hierarchy theorem (Cook, Seiferas, Fischer, Meyer, Zak), it is known that $\mathcal{NP}$ is properly contained in $\mathcal{NEXP}$; see Papadimitriou [32] (Chapters 7 and 20) and Arora and Barak [2] (Chapter 3, Section 3.2). Then the tiling problem with a single tile as input is a computable $\mathcal{NP}$-hard problem not in $\mathcal{NP}$.

(2) If we relax the finiteness condition and require that the entire upper half-plane be tiled, i.e., for every $s > 1$, there is a solution to the $2s \times s$-tiling problem, then the problem is undecidable.

In 1972, Richard Karp published a list of twenty one $\mathcal{NP}$-complete problems.

9.8 The Cook–Levin Theorem: SAT is $\mathcal{NP}$-Complete

We finally prove the Cook-Levin theorem.

Theorem 9.8. *(Cook, 1971, Levin, 1973) The satisfiability problem* SAT *is $\mathcal{NP}$-complete.*

Proof. We reduce the tiling problem to SAT. Given a tiling problem, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, we introduce boolean variables

$$x_{mnt},$$

for all m with $1 \leq m \leq 2s$, all n with $1 \leq n \leq s$, and all tiles $t \in \mathcal{T}$.

The intuition is that $x_{mnt} = \mathbf{T}$ iff tile t occurs in some tiling σ so that $\sigma(m,n) = t$.

We define the following clauses:

(1) For all m, n in the correct range, as above,

$$(x_{mnt_1} \vee x_{mnt_2} \vee \cdots \vee x_{mnt_p}),$$

for all p tiles in $\mathcal{T}$.

This clause states that every position in σ is tiled.

(2) For any two distinct tiles $t \neq t' \in \mathcal{T}$, for all m, n in the correct range, as above,

$$(\overline{x}_{mnt} \vee \overline{x}_{mnt'}).$$

This clause states that a position may not be occupied by more than one tile.

(3) For every pair of tiles $(t, t') \in \mathcal{T} \times \mathcal{T} - H$, for all m with $1 \leq m \leq 2s - 1$, and all n, with $1 \leq n \leq s$,

$$(\overline{x}_{mnt} \vee \overline{x}_{m+1\,nt'}).$$

This clause enforces the horizontal adjacency constraints.

(4) For every pair of tiles $(t, t') \in \mathcal{T} \times \mathcal{T} - V$, for all m with $1 \leq m \leq 2s$, and all n, with $1 \leq n \leq s - 1$,

$$(\overline{x}_{mnt} \vee \overline{x}_{m\,n+1\,t'}).$$

This clause enforces the vertical adjacency constraints.

(5) For all m with $1 \leq m \leq 2s$,

$$(x_{m1\sigma_0(m)}).$$

This clause states that the bottom row is correctly tiled with σ_0.

It is easily checked that the tiling problem has a solution iff the conjunction of the clauses just defined is satisfiable. Thus, SAT is $\mathcal{NP}$-complete. □

We sharpen Theorem 9.8 to prove that 3-SAT is also $\mathcal{NP}$-complete. This is the satisfiability problem for clauses containing at most three literals.

We know that we can't go further and retain $\mathcal{NP}$-completeteness, since 2-SAT is in $\mathcal{P}$.

Theorem 9.9. *(Cook, 1971) The satisfiability problem* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. We have to break "long clauses"

$$C = (L_1 \vee \cdots \vee L_k),$$

i.e., clauses containing $k \geq 4$ literals, into clauses with at most three literals, in such a way that satisfiability is preserved.

Example 9.11. For example, consider the following clause with $k = 6$ literals:

$$C = (L_1 \vee L_2 \vee L_3 \vee L_4 \vee L_5 \vee L_6).$$

We create 3 new boolean variables y_1, y_2, y_3, and the 4 clauses

$$(L_1 \vee L_2 \vee y_1),\ (\overline{y_1} \vee L_3 \vee y_2),\ (\overline{y_2} \vee L_4 \vee y_3),\ (\overline{y}_3 \vee L_5 \vee L_6).$$

Let C' be the conjunction of these clauses. We claim that C is satisfiable iff C' is.

Assume that C' is satisfiable but C is not. If so, in any truth assigment v, $v(L_i) = \mathbf{F}$, for $i = 1, 2, \ldots, 6$. To satisfy the first clause, we must have $v(y_1) = \mathbf{T}$., Then to satisfy the second clause, we must have $v(y_2) = \mathbf{T}$, and similarly satisfy the third clause, we must have $v(y_3) = \mathbf{T}$. However, since $v(L_5) = \mathbf{F}$ and $v(L_6) = \mathbf{F}$, the only way to satisfy the fourth clause is to have $v(y_3) = \mathbf{F}$, contradicting that $v(y_3) = \mathbf{T}$. Thus, C is indeed satisfiable.

Let us now assume that C is satisfiable. This means that there is a smallest index i such that L_i is satisfied.

Say $i = 1$, so $v(L_1) = \mathbf{T}$. Then if we let $v(y_1) = v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 2$, so $v(L_1) = \mathbf{F}$ and $v(L_2) = \mathbf{T}$. Again if we let $v(y_1) = v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 3$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, and $v(L_3) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$ and $v(y_2) = v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 4$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, and $v(L_4) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{F}$, we see that C' is satisfied.

Say $i = 5$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, $v(L_4) = \mathbf{F}$, and $v(L_5) = \mathbf{T}$. If we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{T}$, we see that C' is satisfied.

Say $i = 6$, so $v(L_1) = \mathbf{F}$, $v(L_2) = \mathbf{F}$, $v(L_3) = \mathbf{F}$, $v(L_4) = \mathbf{F}$, $v(L_5) = \mathbf{F}$, and $v(L_6) = \mathbf{T}$. Again, if we let $v(y_1) = \mathbf{T}$, $v(y_2) = \mathbf{T}$ and $v(y_3) = \mathbf{T}$, we see that C' is satisfied.

Therefore if C is satisfied, then C' is satisfied in all cases.

In general, for every long clause, create $k - 3$ new boolean variables $y_1, \ldots y_{k-3}$, and the $k - 2$ clauses

$$\begin{aligned}&(L_1 \vee L_2 \vee y_1), (\overline{y_1} \vee L_3 \vee y_2), (\overline{y_2} \vee L_4 \vee y_3), \cdots,\\ &(\overline{y}_{k-4} \vee L_{k-2} \vee y_{k-3}), (\overline{y}_{k-3} \vee L_{k-1} \vee L_k).\end{aligned}$$

Let C' be the conjunction of these clauses. We claim that C is satisfiable iff C' is.

Assume that C' is satisfiable, but that C is not. Then for every truth assignment v, we have $v(L_i) = \mathbf{F}$, for $i = 1, \ldots, k$.

However, C' is satisfied by some v, and the only way this can happen is that $v(y_1) = \mathbf{T}$, to satisfy the first clause. Then $v(\overline{y_1}) = \mathbf{F}$, and we must have $v(y_2) = \mathbf{T}$, to satisfy the second clause.

By induction, we must have $v(y_{k-3}) = \mathbf{T}$, to satisfy the next to the last clause. However, the last clause is now false, a contradiction.

Thus, if C' is satisfiable, then so is C.

Conversely, assume that C is satisfiable. If so, there is some truth assignment, v, so that $v(C) = \mathbf{T}$, and thus, there is a smallest index i, with $1 \leq i \leq k$, so that $v(L_i) = \mathbf{T}$ (and so, $v(L_j) = \mathbf{F}$ for all $j < i$).

Let v' be the assignment extending v defined so that

$$v'(y_j) = \mathbf{F} \quad \text{if} \quad \max\{1, i-1\} \leq j \leq k-3,$$

and $v'(y_j) = \mathbf{T}$, otherwise.

It is easily checked that $v'(C') = \mathbf{T}$. □

Another version of 3-SAT can be considered, in which every clause has exactly three literals. We will call this the problem *exact* 3-SAT.

Theorem 9.10. *(Cook, 1971) The satisfiability problem for exact* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. A clause of the form (L) is satisfiable iff the following four clauses are satisfiable:

$$(L \vee u \vee v), (L \vee \overline{u} \vee v), (L \vee u \vee \overline{v}), (L \vee \overline{u} \vee \overline{v})$$

where u, v are new variables. A clause of the form $(L_1 \vee L_2)$ is satisfiable iff the following two clauses are satisfiable:

$$(L_1 \vee L_2 \vee u), (L_1 \vee L_2 \vee \overline{u}).$$

Thus, we have a reduction of 3-SAT to exact 3-SAT. □

We now make some remarks about the conversion of propositions to CNF and about the satisfiability and validity of arbitrary propositions.

9.9 Satisfiability of Arbitrary Propositions and CNF

The satisfiability problem for arbitrary propositions belongs to $\mathcal{NP}$ because if we can guess a truth assignment v satisfying a proposition A, then evaluating the truth value of A under v can certainly be done in polynomial time. Since a proposition in CNF is a special kind of proposition and since the satisfiability problem for propositions in CNF (SAT) is $\mathcal{NP}$-complete, the satisfiability problem for arbitrary propositions is also $\mathcal{NP}$-complete.

Since the satisfiability problem for propositions in CNF is $\mathcal{NP}$-complete, there is a polynomial-time reduction that takes an arbitrary proposition A and produces a proposition A' in CNF such that A is satisfiable iff A' is satisfiable. In general, given a proposition A, a proposition A' in CNF equivalent to A may have an exponential length in the size of A. However, using new variables, there is an algorithm to convert a proposition A to another proposition A' (containing the new variables) whose length is polynomial in the length of A and such that A is satisfiable iff A' is satisfiable.

We will explain how to convert an arbitrary proposition A to an equivalent proposition in CNF, and also how to construct in polynomial time a proposition A' such that A is satisfiable iff A' is satisfiable. We also briefly discuss the issue of uniqueness of the CNF. In short, it is not unique!

Recall the definition of arbitrary propositions.

Definition 9.10. The set of *propositions* (over the connectives $\vee$, $\wedge$, and $\neg$) is defined inductively as follows:

(1) Every propositional letter, $x \in \mathbf{PV}$, is a proposition (an *atomic* proposition).

(2) If A is a proposition, then $\neg A$ is a proposition.

(3) If A and B are propositions, then $(A \vee B)$ is a proposition.

(4) If A and B are propositions, then $(A \wedge B)$ is a proposition.

Two propositions A and B are *equivalent*, denoted $A \equiv B$, if

$$v \models A \quad \text{iff} \quad v \models B$$

for all truth assignments, v. It is easy to show that $A \equiv B$ iff the proposition

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

is valid.

Definition 9.11. A proposition P is in *conjunctive normal form* (*CNF*) if it is a conjunction $P = C_1 \wedge \cdots \wedge C_n$ of propositions C_j which are disjunctions of literals (a literal is either a variable x or the negation $\neg x$ (also denoted $\overline{x}$) of a variable x).

A proposition P is in *disjunctive normal form* (*DNF*) if it is a disjunction $P = D_1 \vee \cdots \vee D_n$ of propositions D_j which are conjunctions of literals.

There are propositions such that any equivalent proposition in CNF has size exponential in terms of the original proposition.

Example 9.12. Here is such an example:

$$A = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2n-1} \wedge x_{2n}).$$

Observe that it is in DNF. We will prove a little later that any CNF for A contains 2^n occurrences of variables.

Proposition 9.11. *Every proposition A is equivalent to a proposition A' in CNF.*

There are several ways of proving Proposition 9.11. One method is algebraic, and consists in using the algebraic laws of boolean algebra. First one may convert a proposition to *negation normal form*, or *nnf*.

Definition 9.12. A proposition is in *negation normal form* or *nnf* if all occurrences of $\neg$ only appear in front of propositional variables, but not in front of compound propositions.

Any proposition can be converted to an equivalent one in nnf by using the de Morgan laws:

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

Observe that if A has n connectives, then the equivalent formula A' in nnf has at most $2n-1$ connectives. Then a proposition in nnf can be converted to CNF,

A nice method to convert a proposition in nnf to CNF is to construct a tree whose nodes are labeled with sets of propositions using the following (Gentzen-style) rules:

$$\frac{P, \Delta \qquad Q, \Delta}{(P \wedge Q), \Delta}$$

and

$$\frac{P, Q, \Delta}{(P \vee Q), \Delta}$$

where Δ stands for any set of propositions (even empty), and the comma stands for union. Thus, it is assumed that $(P \wedge Q) \notin \Delta$ in the first case, and that $(P \vee Q) \notin \Delta$ in the second case.

Since we interpret a set, Γ, of propositions as a disjunction, a valuation, v, satisfies Γ iff it satisfies *some* proposition in Γ.

Observe that a valuation v satisfies the conclusion of a rule iff it satisfies both premises in the first case, and the single premise in the second case. Using these rules, we can build a finite tree whose leaves are labeled with sets of literals.

By the above observation, a valuation v satisfies the proposition labeling the root of the tree iff it satisfies all the propositions labeling the leaves of the tree.

But then, a CNF for the original proposition A (in nnf, at the root of the tree) is the conjunction of the clauses appearing as the leaves of the tree. We may exclude the clauses that are tautologies, and we may discover in the process that A is a tautology (when all leaves are tautologies).

Example 9.13. An illustration of the above method to convert the proposition

$$A = (x_1 \wedge y_1) \vee (x_2 \wedge y_2)$$

is shown below:

$$\dfrac{\dfrac{\dfrac{x_1, x_2 \qquad x_1, y_2}{x_1, x_2 \wedge y_2} \qquad \dfrac{y_1, x_2 \qquad y_1, y_2}{y_1, x_2 \wedge y_2}}{x_1 \wedge y_1, x_2 \wedge y_2}}{(x_1 \wedge y_1) \vee (x_2 \wedge y_2)}$$

We obtain the CNF

$$B = (x_1 \vee x_2) \wedge (x_1 \vee y_2) \wedge (y_1 \vee x_2) \wedge (y_1 \vee y_2).$$

Remark: Rules for dealing for $\neg$ can also be created. In this case, we work with pairs of sets of propositions,

$$\Gamma \to \Delta,$$

where, the propositions in Γ are interpreted conjunctively, and the propositions in Δ are interpreted disjunctively. We obtain a sound and complete proof system for propositional logic (a "Gentzen-style" proof system, see *Logic for Computer Science*, Gallier [18]).

Going back to our "bad" proposition A from Example 9.12, by induction, we see that any tree for A has 2^n leaves.

However, the following result holds.

Proposition 9.12. *For any proposition A, we can construct in polynomial time a formula A' in CNF, so that A is satisfiable iff A' is satisfiable, by creating new variables.*

Sketch of proof. First we convert A to nnf, which yields a proposition at most twice as long. Then we proceed recursively. For a conjunction $C \wedge D$, we apply recursively the procedure to C and D. The trick is that for a disjunction $C \vee D$, first we apply recursively the procedure to C and D obtain

$$(C_1 \wedge \cdots \wedge C_m) \vee (D_1 \wedge \cdots \wedge D_n)$$

where the C_i's and the D_j's are clauses. Then we create

$$(C_1 \vee y) \wedge \cdots \wedge (C_m \vee y) \wedge (D_1 \vee \overline{y}) \wedge \cdots \wedge (D_n \vee \overline{y}),$$

where y is a new variable.

It can be shown that the number of new variables required is at most quadratic in the size of A. For details on this construction see Hopcroft, Motwani and Ullman [23] (Section 10.3.3), but beware that the proof on page 455 contains a mistake. Repair the mistake. □

Example 9.14. Consider the proposition

$$A = (x_1 \land \neg x_2) \lor ((\neg x_1 \land x_2) \lor (x_2 \lor x_3)).$$

First, since x_1 and $\neg x_2$ are clauses, we get

$$A_1 = x_1 \land \neg x_2.$$

Since $\neg x_1$, x_2 and $x_2 \lor x_3$ are clauses, from $(\neg x_1 \land x_2) \lor (x_2 \lor x_3)$ we construct

$$A_2 = (\neg x_1 \lor y_1) \land (x_2 \lor y_1) \land (x_2 \lor x_3 \lor \neg y_1).$$

Next, since A_1 and A_2 are conjunctions of clauses, we construct

$$A' = (x_1 \lor y_2) \land (\neg x_2 \lor y_2) \land (\neg x_1 \lor y_1 \lor \neg y_2) \land (x_2 \lor y_1 \lor \neg y_2) \land (x_2 \lor x_3 \lor \neg y_1 \lor \neg y_2),$$

a conjunction of clauses which is satisfiable iff A is satisfiable.

Warning: In general, the proposition A' is *not* equivalent to the proposition A.

Remark: Other authors, including Hoprcoft, Motwani, and Ullman, prove that the satisfiability problem for *arbitrary* propositions is $\mathcal{NP}$-complete, by showing how the computation of a nondeterministic Turing machine (operating in polynomial time) can be simulated using propositions. For this simulation to work, it appears that propositions that are not in CNF are required. Then Proposition 9.12 is used to show that the satisfiability problem for propositions in CNF (SAT) is also $\mathcal{NP}$-complete.

In our approach, since we have already shown that the bounded tiling problem is $\mathcal{NP}$-complete, in the second step to reduce the tiling problem to SAT we only need clauses to perform the reduction. Thus we don't need Proposition 9.12 to prove that SAT is $\mathcal{NP}$-complete.

We just observed that the satisfiability problems for propositions in CNF is as hard as the satisfiability problems for arbitrary propositions. However, the situation is completely different for the validity problem. Indeed, a proposition $P = C_1 \land \cdots \land C_m$ in CNF is valid iff every conjunct C_i is valid. But each C_i is clause, namely a disjunction of literals

$$C_i = L_{i1} \lor \cdots \lor L_{in_i},$$

where $L_{i,j}$ is either a variable x or the negation $\neg x$ of a variable. But such a disjunction is valid iff some variable and its negation both occur in C_i. This is because if all the L_{ij} were variables, we could falsify C_i by assigning the truth value **F** to all of them, and if all the L_{ij} were negations of variables, we could falsify C_i by assigning the truth value **T** to all of them. Therefore, the validity problem for proposition in CNF is in $\mathcal{P}$.

This does not help to obtain a polynomial time algorithm to test the validity of arbitrary propositions because converting a proposition to a CNF may yield a proposition whose size is exponential in terms of the size of the original proposition. We can view the method using the Gentzen rules described earlier for building a tree from a proposition P in nnf as an attempt to demonstrate that the proposition P is valid. If this attempt fails, then we obtain a CNF for P, so our efforts are not wasted.

The question of uniqueness of the CNF is a bit tricky. For example, the proposition

$$A = (u \wedge (x \vee y)) \vee (\neg u \wedge (x \vee y))$$

has

$$\begin{aligned} A_1 &= (u \vee x \vee y) \wedge (\neg u \vee x \vee y) \\ A_2 &= (u \vee \neg u) \wedge (x \vee y) \\ A_3 &= x \vee y, \end{aligned}$$

as equivalent propositions in CNF!

We can get a *unique* CNF equivalent to a given proposition if we do the following:

(1) Let $\mathrm{Var}(A) = \{x_1, \ldots, x_m\}$ be the set of variables occurring in A.

(2) Define a *maxterm w.r.t.* $\mathrm{Var}(A)$ as any disjunction of m pairwise distinct literals formed from $\mathrm{Var}(A)$, and not containing both some variable x_i and its negation $\neg x_i$.

(3) Then it can be shown that for any proposition A that is not a tautology, there is a *unique* proposition in CNF *equivalent* to A, whose clauses consist of maxterms formed from $\mathrm{Var}(A)$.

The above definition can yield strange results. For instance, the CNF of any unsatisfiable proposition with m distinct variables is the conjunction of all of its 2^m maxterms! The above notion does not cope well with minimality.

For example, according to the above, the CNF of

$$A = (u \wedge (x \vee y)) \vee (\neg u \wedge (x \vee y))$$

should be

$$A_1 = (u \vee x \vee y) \wedge (\neg u \vee x \vee y).$$

Chapter 10

Some $\mathcal{NP}$-Complete Problems

10.1 Statements of the Problems

In this chapter we will show that certain classical algorithmic problems are $\mathcal{NP}$-complete. This chapter is heavily inspired by Lewis and Papadimitriou's excellent treatment [28]. In order to study the complexity of these problems in terms of resource (time or space) bounded Turing machines (or RAM programs), it is crucial to be able to encode instances of a problem P as strings in a language L_P. Then an instance of a problem P is solvable iff the corresponding string belongs to the language L_P. This implies that our problems must have a yes–no answer, which is not always the usual formulation of optimization problems where what is required is to find some *optimal* solution, that is, a solution minimizing or maximizing so objective (cost) function F. For example the standard formulation of the traveling salesman problem asks for a tour (of the cities) of minimal cost.

Fortunately, there is a trick to reformulate an optimization problem as a yes–no answer problem, which is to explicitly incorporate a *budget* (or *cost*) term B into the problem, and instead of asking whether some objective function F has a minimum or a maximum w, we ask whether there is a solution w such that $F(w) \leq B$ in the case of a minimum solution, or $F(w) \geq B$ in the case of a maximum solution.

If we are looking for a minimum of F, we try to guess the minimum value B of F and then we solve the problem of finding w such that $F(w) \leq B$. If our guess for B is too small, then we fail. In this case, we try again with a larger value of B. Otherwise, if B was not too small we find some w such that $F(w) \leq B$, but w may not correspond to a minimum of F, so we try again with a smaller value of B, and so on. This yields an approximation method to find a minimum of F.

Similarly, if we are looking for a maximum of F, we try to guess the maximum value B of F and then we solve the problem of finding w such that $F(w) \geq B$. If our guess for B is too large, then we fail. In this case, we try again with a smaller value of B. Otherwise, if B was not too large we find some w such that $F(w) \geq B$, but w may not correspond to a maximum of F, so we try again with a greater value of B, and so on. This yields an approximation method to find a maximum of F.

We will see several examples of this technique in Problems 5–8 listed below.

The problems that will consider are

(1) Exact Cover

(2) Hamiltonian Cycle for directed graphs

(3) Hamiltonian Cycle for undirected graphs

(4) The Traveling Salesman Problem

(5) Independent Set

(6) Clique

(7) Node Cover

(8) Knapsack, also called subset sum

(9) Inequivalence of $*$-free Regular Expressions

(10) The 0-1-integer programming problem

We begin by describing each of these problems.

(1) **Exact Cover**

We are given a finite nonempty set $U = \{u_1, \ldots, u_n\}$ (the universe), and a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of $m \geq 1$ nonempty subsets of U. The question is whether there is an *exact cover*, that is, a subfamily $\mathcal{C} \subseteq \mathcal{F}$ of subsets in $\mathcal{F}$ such that the sets in $\mathcal{C}$ are disjoint and their union is equal to U.

For example, let $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$, and let $\mathcal{F}$ be the family

$$\mathcal{F} = \{\{u_1, u_3\}, \{u_2, u_3, u_6\}, \{u_1, u_5\}, \{u_2, u_3, u_4\}, \{u_5, u_6\}, \{u_2, u_4\}\}.$$

The subfamily

$$\mathcal{C} = \{\{u_1, u_3\}, \{u_5, u_6\}, \{u_2, u_4\}\}$$

is an exact cover.

It is easy to see that **Exact Cover** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce the **Satisfiability Problem** to it. This means that we provide a method running in polynomial time that converts every instance of the **Satisfiability Problem** to an instance of **Exact Cover**, such that the first problem has a solution iff the converted problem has a solution.

(2) **Hamiltonian Cycle (for Directed Graphs)**

Recall that a *directed graph* G is a pair $G = (V, E)$, where $E \subseteq V \times V$. Elements of V are called *nodes* (or *vertices*). A pair $(u, v) \in E$ is called an *edge* of G. We will restrict ourselves to *simple graphs*, that is, graphs without edges of the form (u, u); equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of $n+1$ edges ($n \geq 0$)

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from u to v is simply a single edge, (u, v).)

A directed graph G is *strongly connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself. We will restrict out attention to finite graphs, i.e. graphs (V, E) where V is a finite set.

Definition 10.1. Given a directed graph G, a *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Directed Graphs): Given a directed graph G, is there an Hamiltonian cycle in G?

Is there is a Hamiltonian cycle in the directed graph D shown in Figure 10.1?

Finding a Hamiltonian cycle in this graph does not appear to be so easy! A solution is shown in Figure 10.2 below.

It is easy to see that **Hamiltonian Cycle (for Directed Graphs)** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Hamiltonian Cycle (for Directed Graphs)** such that the first problem has a solution iff the converted problem has a solution. This is perphaps the hardest reduction.

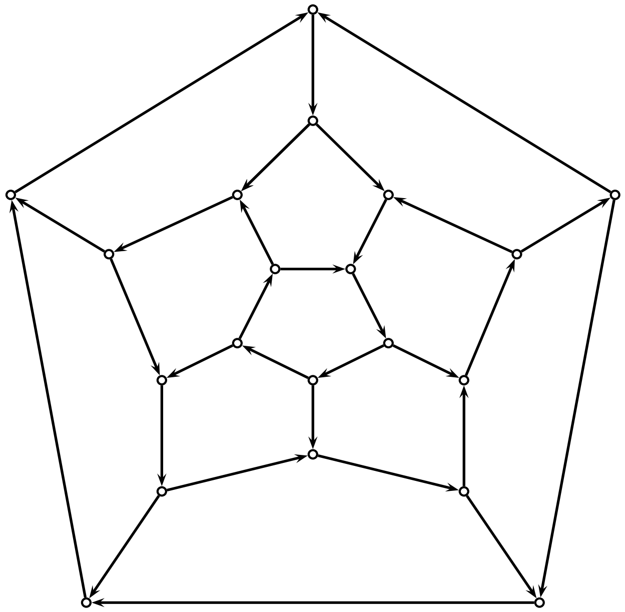


Figure 10.1: A tour "around the world."

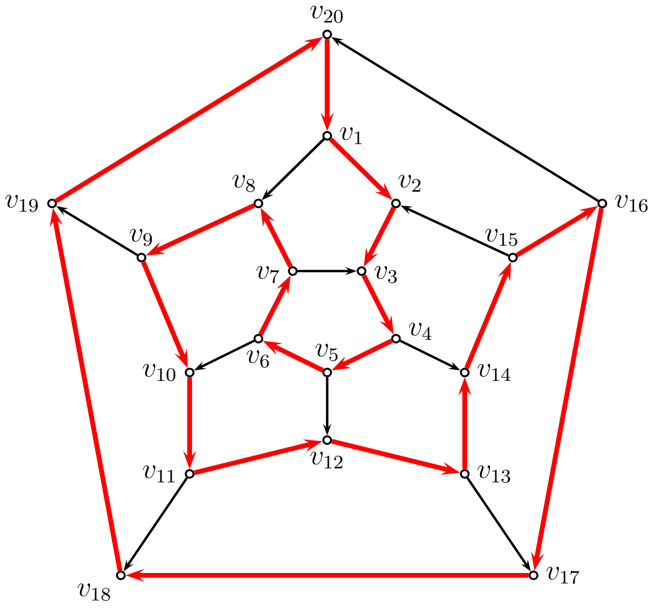


Figure 10.2: A Hamiltonian cycle in D.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

Recall that an *undirected graph* G is a pair $G = (V, E)$, where E is a set of subsets $\{u, v\}$ of V consisting of exactly two distinct elements. Elements of V are called *nodes* (or *vertices*). A pair $\{u, v\} \in E$ is called an *edge* of G.

Given any two nodes $u, v \in V$, a *path from u to v* is any sequence of n nodes ($n \geq 2$)

$$u = u_1, u_2, \ldots, u_n = v$$

such that $\{u_i, u_{i+1}\} \in E$ for $i = 1, \ldots, n-1$. (If $n = 2$, a path from u to v is simply a single edge, $\{u, v\}$.)

An undirected graph G is *connected* if for every pair $(u, v) \in V \times V$, there is a path from u to v. A *closed path, or cycle*, is a path from some node u to itself.

Definition 10.2. Given an undirected graph G, a *Hamiltonian cycle* is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Undirected Graphs): Given an undirected graph G, is there an Hamiltonian cycle in G?

An instance of this problem is obtained by changing every directed edge in the directed graph of Figure 10.1 to an undirected edge. The directed Hamiltonian cycle given in Figure 10.1 is also an undirected Hamiltonian cycle of the undirected graph of Figure 10.3.

We see immediately that **Hamiltonian Cycle (for Undirected Graphs)** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce **Hamiltonian Cycle (for Directed Graphs)** to it. This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle (for Directed Graphs)** to an instance of **Hamiltonian Cycle (for Undirected Graphs)** such that the first problem has a solution iff the converted problem has a solution. This is an easy reduction.

(4) **Traveling Salesman Problem**

We are given a set $\{c_1, c_2, \ldots, c_n\}$ of $n \geq 2$ cities, and an $n \times n$ matrix $D = (d_{ij})$ of nonnegative integers, where d_{ij} is the *distance* (or *cost*) of traveling from city c_i to city c_j. We assume that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all i, j, so that the matrix D is symmetric and has zero diagonal.

Traveling Salesman Problem: Given some $n \times n$ matrix $D = (d_{ij})$ as above and some integer $B \geq 0$ (the *budget* of the traveling salesman), find a permutation π of $\{1, 2, \ldots, n\}$ such that

$$c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)} \leq B.$$

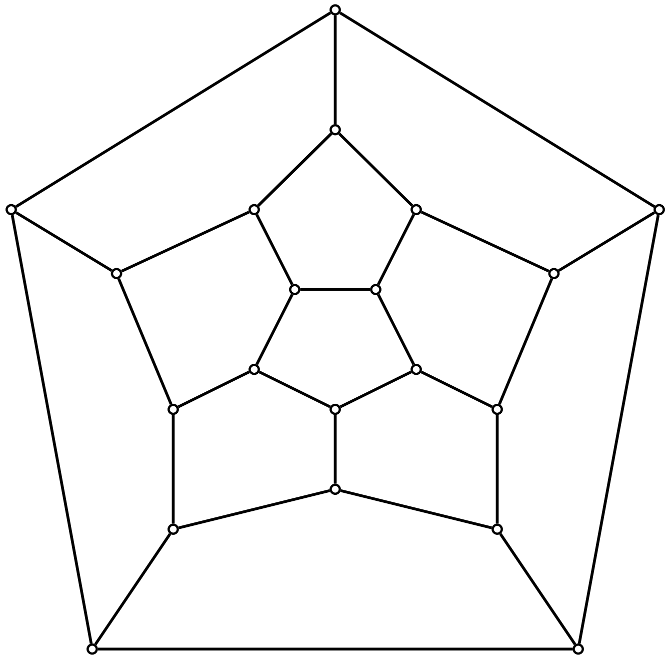


Figure 10.3: A tour "around the world," undirected version.

The quantity $c(\pi)$ is the *cost* of the trip specified by π. The Traveling Salesman Problem has been stated in terms of a budget so that it has a yes or no answer, which allows us to convert it into a language. A minimal solution corresponds to the smallest feasible value of B.

Example 10.1. Consider the 4×4 symmetric matrix given by

$$D = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 1 & 0 & 3 \\ 1 & 1 & 3 & 0 \end{pmatrix},$$

and the budget $B = 4$. The tour specified by the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix}$$

has cost 4, since

$$\begin{aligned} c(\pi) &= d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)} \\ &= d_{14} + d_{42} + d_{23} + d_{31} \\ &= 1 + 1 + 1 + 1 = 4. \end{aligned}$$

The cities in this tour are traversed in the order

$$(1, 4, 2, 3, 1).$$

It is clear that the **Traveling Salesman Problem** is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it. This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle Problem (Undirected Graphs)** to an instance of the **Traveling Salesman Problem** such that the first problem has a solution iff the converted problem has a solution.

(5) **Independent Set**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set C of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is *no* edge $\{v_i, v_j\} \in E$?

A maximal independent set with 3 nodes is shown in Figure 10.4. A maximal solution corresponds to the largest feasible value of K. The problem **Independent Set** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact 3-Satisfiability** to an instance of **Independent Set** such that the first problem has a solution iff the converted problem has a solution.

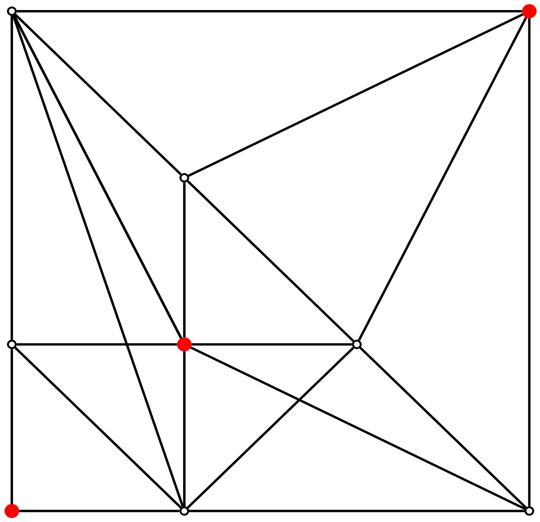


Figure 10.4: A maximal Independent Set in a graph.

(6) **Clique**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set C of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is *some* edge $\{v_i, v_j\} \in E$? Equivalently, does G contain a complete subgraph with at least K nodes?

A maximal clique with 4 nodes is shown in Figure 10.5. A maximal solution corresponds to the largest feasible value of K. The problem **Clique** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it. This means that we provide a method running in polynomial time that converts every instance of **Independent Set** to an instance of **Clique** such that the first problem has a solution iff the converted problem has a solution.

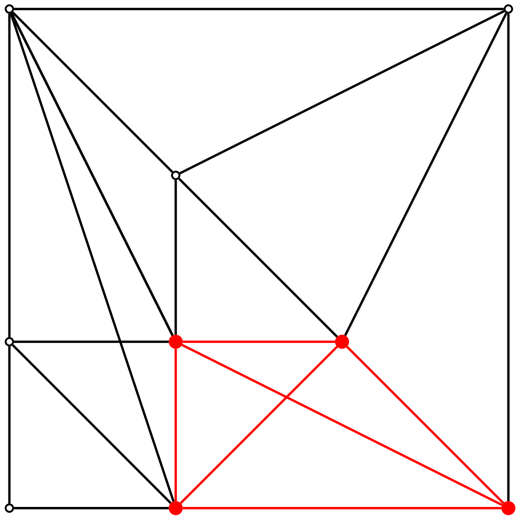


Figure 10.5: A maximal Clique in a graph.

(7) **Node Cover**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $B \geq 2$, is there a set C of nodes with $|C| \leq B$ such that C covers all edges in G, which means that for every edge $\{v_i, v_j\} \in E$, either $v_i \in C$ or $v_j \in C$?

A minimal node cover with 6 nodes is shown in Figure 10.6. A minimal solution corresponds to the smallest feasible value of B. The problem **Node Cover** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it. This means that we provide a method running in polynomial time that converts every instance of

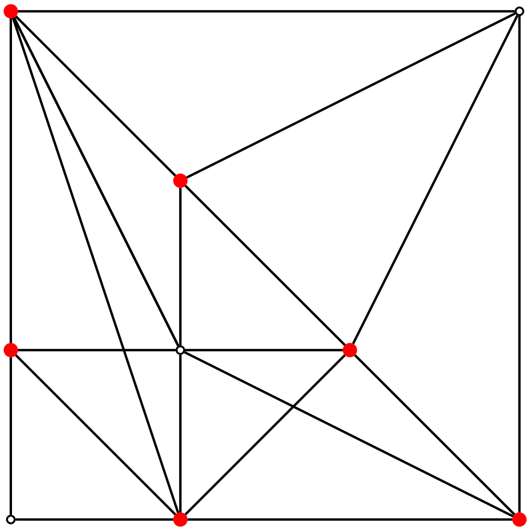


Figure 10.6: A minimal Node Cover in a graph.

Independent Set to an instance of **Node Cover** such that the first problem has a solution iff the converted problem has a solution.

The Node Cover problem has the following interesting interpretation: think of the nodes of the graph as rooms of a museum (or art gallery *etc.*), and each edge as a straight corridor that joins two rooms. Then Node Cover may be useful in assigning as few as possible guards to the rooms, so that all corridors can be seen by a guard.

(8) **Knapsack (also called Subset sum)**

The problem is this: Given a finite nonempty set $S = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers, and some integer $K \geq 0$, all represented in binary, is there a nonempty subset $I \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in I} a_i = K?$$

A "concrete" realization of this problem is that of a hiker who is trying to fill her/his backpack to its maximum capacity with items of varying weights or values.

It is easy to see that the **Knapsack** Problem is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Exact Cover** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Knapsack** Problem such that the first problem has a solution iff the converted problem has a solution.

Remark: The **0-1 Knapsack Problem** is defined as the following problem. Given a set of n items, numbered from 1 to n, each with a weight $w_i \in \mathbb{N}$ and a value $v_i \in \mathbb{N}$, given a maximum capacity $W \in \mathbb{N}$ and a budget $B \in \mathbb{N}$, is there a set of n variables $x_1, \ldots, x_n$ with $x_i \in \{0, 1\}$ such that

$$\sum_{i=1}^{n} x_i v_i \geq B,$$
$$\sum_{i=1}^{n} x_i w_i \leq W.$$

Informally, the problem is to pick items to include in the knapsack so that the sum of the values exceeds a given minimum B (the goal is to maximize this sum), and the sum of the weights is less than or equal to the capacity W of the knapsack. A maximal solution corresponds to the largest feasible value of B.

The **Knapsack** Problem as we defined it (which is how Lewis and Papadimitriou define it) is the special case where $v_i = w_i$ for $i = 1, \ldots, n$, the v_i are pairwise distinct (they form a set), and $W = B$. For this reason, it is also called the **Subset Sum** Problem. Clearly, the **Knapsack** (**Subset Sum**) Problem reduces to the **0-1 Knapsack** Problem, and thus the **0-1 Knapsack** Problem is also NP-complete.

(9) **Inequivalence of $*$-free Regular Expressions**

Recall that the problem of deciding the equivalence $R_1 \cong R_2$ of two regular expressions R_1 and R_2 is the problem of deciding whether R_1 and R_2 define the same language, that is, $\mathcal{L}[R_1] = \mathcal{L}[R_2]$. Is this problem in $\mathcal{NP}$?

In order to show that the equivalence problem for regular expressions is in $\mathcal{NP}$ we would have to be able to somehow check in polynomial time that two expressions define the same language, but this is still an open problem.

What might be easier is to decide whether two regular expressions R_1 and R_2 are *inequivalent*. For this, we just have to find a string w such that either $w \in \mathcal{L}[R_1] - \mathcal{L}[R_2]$ or $w \in \mathcal{L}[R_2] - \mathcal{L}[R_1]$. The problem is that if we can guess such a string w, we still have to check in polynomial time that $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, and this implies that there is a bound on the length of w which is polynomial in the sizes of R_1 and R_2. Again, this is an open problem.

To obtain a problem in $\mathcal{NP}$ we have to consider a restricted type of regular expressions, and it turns out that $*$-free regular expressions are the right candidate. A *$*$-free regular expression* is a regular expression which is built up from the atomic expressions using only $+$ and $\cdot$, but not $*$. For example,

$$R = ((a + b)aa(a + b) + aba(a + b)b)$$

is such an expression.

It is easy to see that if R is a $*$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. In particular, $\mathcal{L}[R]$ is finite. The above observation shows that if R_1 and R_2 are $*$-free and if there is a string $w \in (\mathcal{L}[R_1]-\mathcal{L}[R_2]) \cup (\mathcal{L}[R_2]-\mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time. It follows that the inequivalence problem for $*$ -free regular expressions is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it. This means that we provide a method running in polynomial time that converts every instance of **Satisfiability Problem** to an instance of **Inequivalence of Regular Expressions** such that the first problem has a solution iff the converted problem has a solution.

Observe that both problems of **Inequivalence of Regular Expressions** and **Equivalence of Regular Expressions** are as hard as **Inequivalence of $*$-free Regular Expressions**, since if we could solve the first two problems in polynomial time, then we we could solve **Inequivalence of $*$-free Regular Expressions** in polynomial time, but since this problem is $\mathcal{NP}$-complete, we would have $\mathcal{P} = \mathcal{NP}$. This is very unlikely, so the complexity of **Equivalence of Regular Expressions** remains open.

(10) 0-1 **integer programming problem**

Let A be any $p \times q$ matrix with integer coefficients and let $b \in \mathbb{Z}^p$ be any vector with integer coefficients. The 0-1 **integer programming problem** is to find whether a system of p linear equations in q variables

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1q}x_q &= b_1 \\ \vdots \qquad\qquad & \quad \vdots \\ a_{i1}x_1 + \cdots + a_{iq}x_q &= b_i \\ \vdots \qquad\qquad & \quad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q &= b_p \end{aligned}$$

with $a_{ij}, b_i \in \mathbb{Z}$ has any solution $x \in \{0,1\}^q$, that is, with $x_i \in \{0,1\}$. In matrix form, if we let

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1q} \\ \vdots & \ddots & \vdots \\ a_{p1} & \cdots & a_{pq} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_q \end{pmatrix},$$

then we write the above system as

$$Ax = b.$$

Example 10.2. Is there a solution $x = (x_1, x_2, x_3, x_4, x_5, x_6)$ of the linear system

$$\begin{pmatrix} 1 & -2 & 1 & 3 & -1 & 4 \\ 2 & 2 & -1 & 0 & 1 & -1 \\ -1 & 1 & 2 & 3 & -2 & 3 \\ 3 & 1 & -1 & 2 & -1 & 4 \\ 0 & 1 & -1 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} 9 \\ 0 \\ 7 \\ 8 \\ 2 \end{pmatrix}$$

with $x_i \in \{0, 1\}$?

Indeed, $x = (1, 0, 1, 1, 0, 1)$ is a solution.

It is immediate that 0-1 **integer programming problem** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete we reduce the **bounded tiling** problem to it. This means that we provide a method running in polynomial time that converts every instance of the **bounded tiling** problem to an instance of the 0-1 **integer programming problem** such that the first problem has a solution iff the converted problem has a solution.

10.2 Proofs of $\mathcal{NP}$-Completeness

(1) **Exact Cover**

To prove that **Exact Cover** is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it:

Satisfiability Problem $\leq_P$ **Exact Cover**

Given a set $F = \{C_1, \ldots, C_\ell\}$ of ℓ clauses constructed from n propositional variables $x_1, \ldots, x_n$, we must construct in polynomial time (in the sum of the lengths of the clauses) an instance $\tau(F) = (U, \mathcal{F})$ of **Exact Cover** such that F is satisfiable iff $\tau(F)$ has a solution.

Example 10.3. If

$$F = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \vee \overline{x_3})\},$$

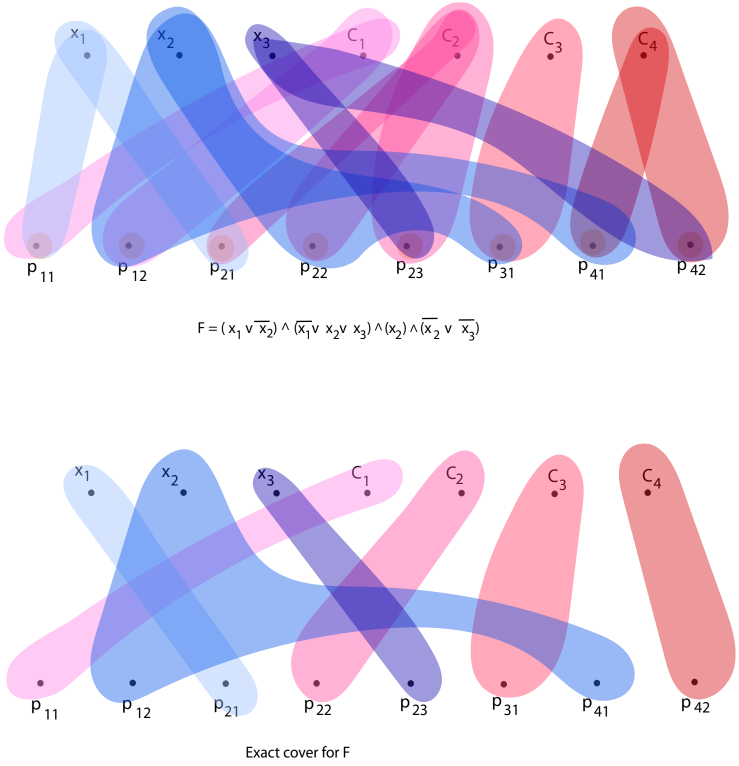
then the universe U is given by

$$U = \{x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31}, p_{41}, p_{42}\},$$

and the family $\mathcal{F}$ consists of the subsets

$$\begin{aligned}
&\{p_{11}\}, \{p_{12}\}, \{p_{21}\}, \{p_{22}\}, \{p_{23}\}, \{p_{31}\}, \{p_{41}\}, \{p_{42}\}\\
&T_{1,\mathbf{F}} = \{x_1, p_{11}\}\\
&T_{1,\mathbf{T}} = \{x_1, p_{21}\}\\
&T_{2,\mathbf{F}} = \{x_2, p_{22}, p_{31}\}\\
&T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}\\
&T_{3,\mathbf{F}} = \{x_3, p_{23}\}\\
&T_{3,\mathbf{T}} = \{x_3, p_{42}\}\\
&\{C_1, p_{11}\}, \{C_1, p_{12}\}, \{C_2, p_{21}\}, \{C_2, p_{22}\}, \{C_2, p_{23}\},\\
&\{C_3, p_{31}\}, \{C_4, p_{41}\}, \{C_4, p_{42}\}.
\end{aligned}$$

The above construction is illustrated in Figure 10.7.



$F = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2) \wedge (\overline{x_2} \vee \overline{x_3})$

Exact cover for F

Figure 10.7: Construction of an exact cover from the set of clauses in Example 10.3.

It is easy to check that the set $\mathcal{C}$ consisting of the following subsets is an exact cover:

$$\begin{aligned}
&T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\},\\
&\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}.
\end{aligned}$$

The general method to construct $(U, \mathcal{F})$ from $F = \{C_1, \ldots, C_\ell\}$ proceeds as follows. The size n of the input is the sum of the lengths of the clauses C_i as strings, Say

$$C_j = (L_{j1} \vee \cdots \vee L_{jm_j})$$

is the jth clause in F, where L_{jk} denotes the kth literal in C_j and $m_j \geq 1$. The universe of $\tau(F)$ is the set

$$U = \{x_i \mid 1 \leq i \leq n\} \cup \{C_j \mid 1 \leq j \leq \ell\} \cup \{p_{jk} \mid 1 \leq j \leq \ell,\ 1 \leq k \leq m_j\}$$

where in the third set p_{jk} corresponds to the kth literal in C_j. The universe U can be constructed in time $O(n^2)$.

The following subsets are included in $\mathcal{F}$:

(a) There is a set $\{p_{jk}\}$ for every p_{jk}.

(b) For every boolean variable x_i, the following two sets are in $\mathcal{F}$:

$$T_{i,\mathbf{T}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = \overline{x_i}\}$$

which contains x_i and all negative occurrences of x_i, and

$$T_{i,\mathbf{F}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = x_i\}$$

which contains x_i and all its positive occurrences. Note carefully that $T_{i,\mathbf{T}}$ involves negative occurrences of x_i whereas $T_{i,\mathbf{F}}$ involves positive occurrences of x_i.

(c) For every clause C_j, the m_j sets $\{C_j, p_{jk}\}$ are in $\mathcal{F}$.

The subsets in (a), (b), (c) can be constructed in time $O(n^3)$. It remains to prove that F is satisfiable iff $\tau(F)$ has a solution. We claim that if v is a truth assignement that satisfies F, then we can make an exact cover $\mathcal{C}$ as follows:

For each x_i, we put the subset $T_{i,\mathbf{T}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{T}$, else we we put the subset $T_{i,\mathbf{F}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{F}$. Also, for every clause C_j, we put some subset $\{C_j, p_{jk}\}$ in $\mathcal{C}$ for a literal L_{jk} which is made true by v. By construction of $T_{i,\mathbf{T}}$ and $T_{i,\mathbf{F}}$, this p_{jk} is not in any set in $\mathcal{C}$ selected so far. Since by hypothesis F is satisfiable, such a literal exists for every clause. Having covered all x_i and C_j, we put a set $\{p_{jk}\}$ in $\mathcal{C}$ for every remaining p_{jk} which has not yet been covered by the sets already in $\mathcal{C}$.

Going back to Example 10.3, the truth assignment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$ satisfies F, so we put

$$\begin{aligned}&T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\},\\ &\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}\end{aligned}$$

in $\mathcal{C}$.

We leave as an exercise to check that the above procedure works.

Conversely, if $\mathcal{C}$ is an exact cover of $\tau(F)$, we define a truth assigment as follows:

For every x_i, if $T_{i,\mathbf{T}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{T}$, else if $T_{i,\mathbf{F}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{F}$. We leave it as an exercise to check that this procedure works.

Example 10.4. Given the exact cover

$$\begin{aligned}&T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\},\\ &\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\},\end{aligned}$$

we get the satisfying assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$.

If we now consider the proposition is CNF given by

$$F_2 = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \vee \overline{x_3} \vee x_4)\}$$

where we have added the boolean variable x_4 to clause C_4, then U also contains x_4 and p_{43} so we need to add the following subsets to $\mathcal{F}$:

$$T_{4,\mathbf{F}} = \{x_4, p_{43}\},\ T_{4,\mathbf{T}} = \{x_4\},\ \{C_4, p_{43}\},\ \{p_{43}\}.$$

The truth assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}, v(x_4) = \mathbf{T}$ satisfies F_2, so an exact cover $\mathcal{C}$ is

$$\begin{aligned}&T_{1,\mathbf{T}} = \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}, T_{4,\mathbf{T}} = \{x_4\},\\ &\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}, \{p_{43}\}.\end{aligned}$$

The above construction is illustrated in Figure 10.8.

Observe that this time, because the truth assignment v makes both literals corresponding to p_{42} and p_{43} true and since we picked p_{42} to form the subset $\{C_4, p_{42}\}$, we need to add the singleton $\{p_{43}\}$ to $\mathcal{C}$ to cover all elements of U.

(2) **Hamiltonian Cycle (for Directed Graphs)**

To prove that **Hamiltonian Cycle (for Directed Graphs)** is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it:

Exact Cover $\leq_P$ Hamiltonian Cycle (for Directed Graphs)

We need to find an algorithm working in polynomial time that converts an instance $(U, \mathcal{F})$ of **Exact Cover** to a directed graph $G = \tau(U, \mathcal{F})$ such that G has a Hamiltonian cycle iff $(U, \mathcal{F})$ has an exact cover. The size n of the input $(U, \mathcal{F})$ is $|U| + |\mathcal{F}|$.

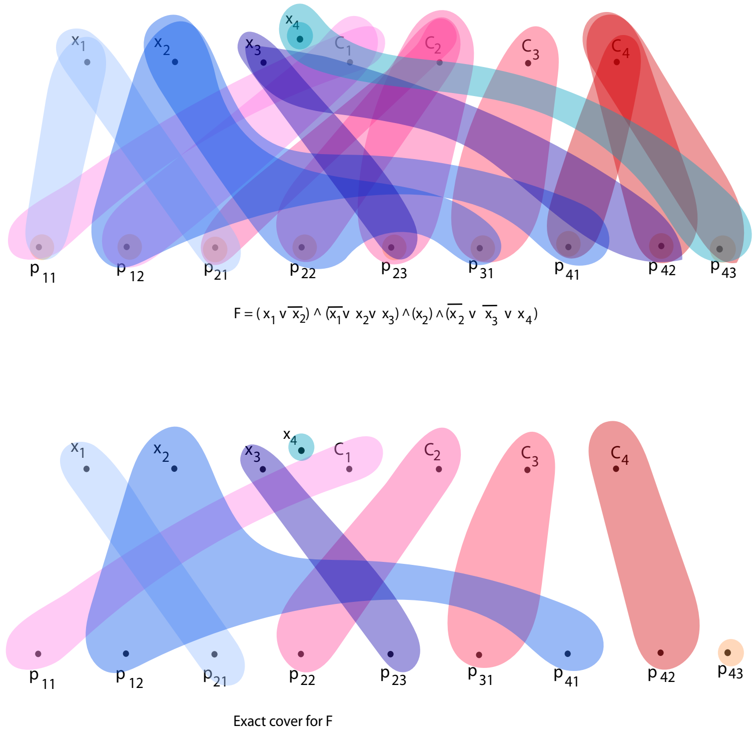


Figure 10.8: Construction of an exact cover from the set of clauses in Example 10.4.

The construction of the graph G uses a trick involving a small subgraph Gad with 7 (distinct) nodes known as a *gadget* shown in Figure 10.9.

The crucial property of the graph Gad is that if Gad is a subgraph of a bigger graph G in such a way that no edge of G is incident to any of the nodes u, v, w unless it is one of the eight edges of Gad incident to the nodes u, v, w, then for any Hamiltonian cycle in G, either the path $(a, u), (u, v), (v, w), (w, b)$ is traversed or the path $(c, w), (w, v), (v, u), (u, d)$ is traversed, but not both.

The reader should convince herself/himself that indeed, any Hamiltonian cycle that does not traverse either the subpath $(a, u), (u, v), (v, w), (w, b)$ from a to b or the subpath $(c, w), (w, v), (v, u), (u, d)$ from c to d will not traverse one of the nodes u, v, w. Also, the fact that node v is traversed exactly once forces only one of the two paths to be traversed but not both. The reader should also convince herself/himself that a smaller graph does not guarantee the desired property.

It is convenient to use the simplified notation with a special type of edge labeled with the exclusive or sign $\oplus$ between the "edges" between a and b and between d and c, as shown in Figure 10.10.

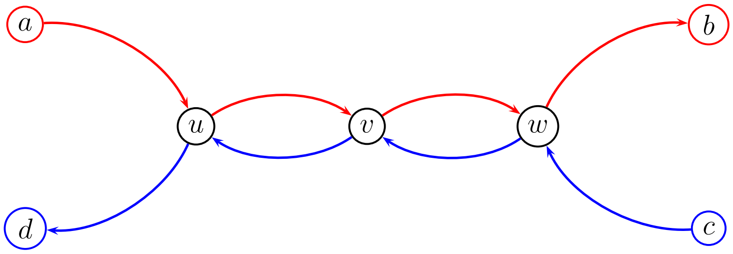


Figure 10.9: A gadget *Gad*.

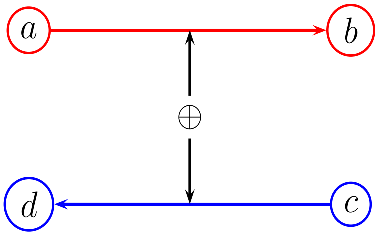


Figure 10.10: A shorthand notation for a gadget.

Whenever such a figure occurs, the actual graph is obtained by substituting a copy of the graph *Gad* (the four nodes a, b, c, d must be distinct). This abbreviating device can be extended to the situation where we build gadgets between a given pair (a, b) and several other pairs $(c_1, d_1), \ldots, (c_m, d_m)$, all nodes being distinct, as illustrated in Figure 10.11.

Either all three edges $(c_1, d_1), (c_2, d_2), (c_3, d_3)$ are traversed or the edge (a, b) is traversed, and these possibilities are mutually exclusive.

The graph $G = \tau(U, \mathcal{F})$ where $U = \{u_1, \ldots, u_n\}$ (with $n \geq 1$) and $\mathcal{F} = \{S_1, \ldots, S_m\}$ (with $m \geq 1$) is constructed as follows:

The graph G has $m + n + 2$ nodes $\{u_0, u_1, \ldots, u_n, S_0, S_1, \ldots, S_m\}$. Note that we have added two extra nodes u_0 and S_0. For $i = 1, \ldots, m$, there are *two* edges $(S_{i-1}, S_i)_1$ and $(S_{i-1}, S_i)_2$ from S_{i-1} to S_i. For $j = 1, \ldots, n$, from u_{j-1} to u_j, there are as many edges as there are sets $S_i \in \mathcal{F}$ containing the element u_j. We can think of each edge between u_{j-1} and u_j as an occurrence of u_j in a uniquely determined set $S_i \in \mathcal{F}$; we denote this edge by $(u_{j-1}, u_j)_i$. We also have an edge from u_n to S_0 and an edge from S_m to u_0, thus "closing the cycle."

What we have constructed so far is not a legal graph since it may have many parallel

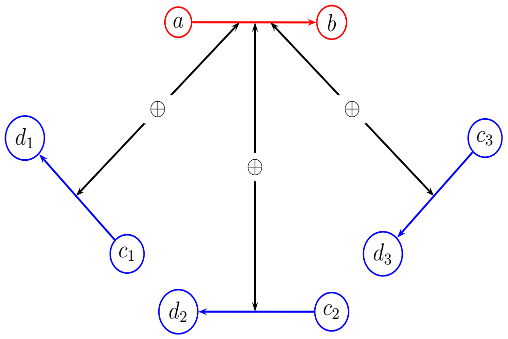


Figure 10.11: A shorthand notation for several gadgets.

edges, but are going to turn it into a legal graph by pairing edges between the u_j's and edges between the S_i's. Indeed, since each edge $(u_{j-1}, u_j)_i$ between u_{j-1} and u_j corresponds to an occurrence of u_j in some uniquely determined set $S_i \in \mathcal{F}$ (that is, $u_j \in S_i$), we put an exclusive-or edge between the edge $(u_{j-1}, u_j)_i$ and the edge $(S_{i-1}, S_i)_2$ between S_{i-1} and S_i, which we call the *long edge*. The other edge $(S_{i-1}, S_i)_1$ between S_{i-1} and S_i (not paired with any other edge) is called the *short edge*. Effectively, we put a copy of the gadget graph *Gad* with $a = u_{j-1}, b = u_j, c = S_{i-1}, d = S_i$ for any pair (u_j, S_i) such that $u_j \in S_i$. The resulting object is indeed a directed graph with no parallel edges. The graph G can be constructed from $(U, \mathcal{F})$ in time $O(n^2)$.

Example 10.5. The above construction is illustrated in Figure 10.12 for the instance of the exact cover problem given by

$$U = \{u_1, u_2, u_3, u_4\},\ \mathcal{F} = \{S_1 = \{u_3, u_4\},\ S_2 = \{u_2, u_3, u_4\},\ S_3 = \{u_1, u_2\}\}.$$

It remains to prove that $(U, \mathcal{F})$ has an exact cover iff the graph $G = \tau(U, \mathcal{F})$ has a Hamiltonian cycle. First, assume that G has a Hamiltonian cycle. If so, for every j some unique "edge" $(u_{j-1}, u_j)_i$ is traversed once (since every u_j is traversed once), and by the exclusive-or nature of the gadget graphs, the corresponding long edge $(S_{i-1}, S_i)_2$ can't be traversed, which means that the short edge $(S_{i-1}, S_i)_1$ is traversed. Consequently, if $\mathcal{C}$ consists of those subsets S_i such that the short edge $(S_{i-1}, S_i)_1$ is traversed, then $\mathcal{C}$ consists of pairwise disjoint subsets whose union is U, namely $\mathcal{C}$ is an exact cover.

In our example, there is a Hamiltonian where the blue edges are traversed between the S_i nodes, and the red edges are traversed between the u_j nodes, namely

$$\begin{aligned}&\text{short } (S_0, S_1),\ \text{long } (S_1, S_2),\ \text{short } (S_2, S_3),\ (S_3, u_0),\\ &(u_0, u_1)_3,\ (u_1, u_2)_3,\ (u_2, u_3)_1,\ (u_3, u_4)_1,\ (u_4, S_0).\end{aligned}$$

The subsets corresponding to the short (S_{i-1}, S_i) edges are S_1 and S_3, and indeed $\mathcal{C} = \{S_1, S_3\}$ is an exact cover.

Note that the exclusive-or property of the gadgets implies the following: since the edge $(u_0, u_1)_3$ must be chosen to obtain a Hamiltonian, the long edge (S_2, S_3) can't be chosen, so the edge $(u_1, u_2)_3$ must be chosen, but then the edge $(u_1, u_2)_2$ is not chosen so the long edge (S_1, S_2) must be chosen, so the edges $(u_2, u_3)_2$ and $(u_3, u_4)_2$ can't be chosen, and thus edges $(u_2, u_3)_1$ and $(u_3, u_4)_1$ must be chosen.

Conversely, if $\mathcal{C}$ is an exact cover for $(U, \mathcal{F})$, then consider the path in G obtained by traversing each short edge $(S_{i-1}, S_i)_1$ for which $S_i \in \mathcal{C}$, each edge $(u_{j-1}, u_j)_i$ such that $u_j \in S_i$, which means that this edge is connected by a $\oplus$-sign to the long edge $(S_{i-1}, S_i)_2$ (by construction, for each u_j there is a unique such S_i), and the edges (u_n, S_0) and (S_m, u_0), then we obtain a Hamiltonian cycle. Observe that the long edges are the inside edges joining the S_i.

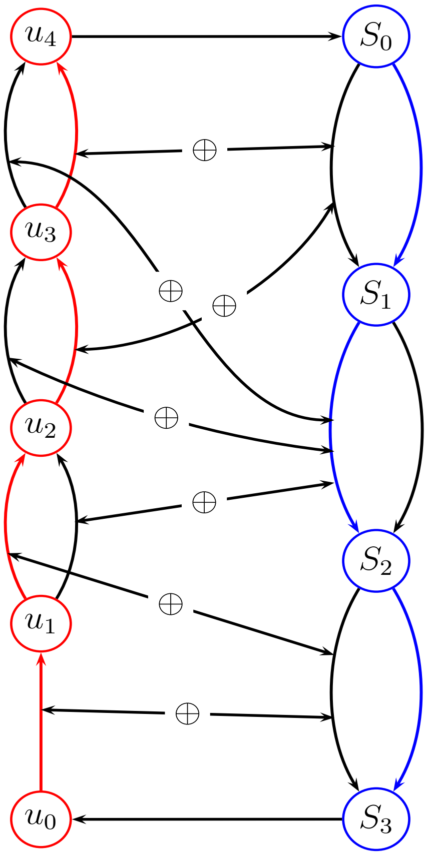


Figure 10.12: The directed graph constructed from the data $(U, \mathcal{F})$ of Example 10.5.

In our example, the exact cover $\mathcal{C} = \{S_1, S_3\}$ yields the Hamiltonian

$$\begin{aligned}&\text{short } (S_0, S_1),\ \text{long } (S_1, S_2),\ \text{short } (S_2, S_3),\ (S_3, u_0),\\ &(u_0, u_1)_3,\ (u_1, u_2)_3,\ (u_2, u_3)_1,\ (u_3, u_4)_1,\ (u_4, S_0)\end{aligned}$$

that we encountered earlier.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

To show that **Hamiltonian Cycle (for Undirected Graphs)** is $\mathcal{NP}$-complete we reduce **Hamiltonian Cycle (for Directed Graphs)** to it:

Hamiltonian Cycle (for Directed Graphs) $\leq_P$ **Hamiltonian Cycle (for Undirected Graphs)**

Given any directed graph $G = (V, E)$ we need to construct in polynomial time an undirected graph $\tau(G) = G' = (V', E')$ such that G has a (directed) Hamiltonian cycle iff G' has a (undirected) Hamiltonian cycle. This is easy. We make three distinct copies v_0, v_1, v_2 of every node $v \in V$ which we put in V', and for every edge $(u, v) \in E$ we create five edges $\{u_0, u_1\}, \{u_1, u_2\}, \{u_2, v_0\}, \{v_0, v_1\}, \{v_1, v_2\}$ which we put in E', as illustrated in the diagram shown in Figure 10.13.

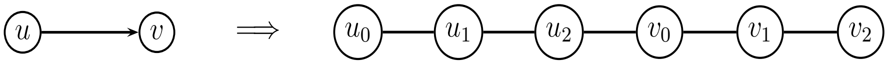


Figure 10.13: Conversion of a directed graph into an undirected graph.

If the size n of the input is $|V| + |E|$, then G' is constructed in time $O(n)$. The crucial point about the graph G' is that although there may be several edges adjacent to a node u_0 or a node u_2, the only way to reach u_1 from u_0 is through the edge $\{u_0, u_1\}$ and the only way to reach u_1 from u_2 is through the edge $\{u_1, u_2\}$.

Suppose there is a Hamiltonian cycle in G'. If this cycle arrives at a node u_0 from the node u_1, then by the above remark, the previous node in the cycle must be u_2. Then the predecessor of u_2 in the cycle must be a node v_0 such that there is an edge $\{u_2, v_0\}$ in G' arising from an edge (u, v) in G. The nodes in the cycle in G' are traversed in the order (v_0, u_2, u_1, u_0) where v_0 and u_2 are traversed in the opposite order in which they occur as the endpoints of the edge (u, v) in G. If so, consider the reverse of our Hamiltonian cycle in G', which is also a Hamiltonian cycle since G' is unoriented. In this cycle, we go from u_0 to u_1, then to u_2, and finally to v_0. In G, we traverse the edge from u to v. In order for the cycle in G' to be Hamiltonian, we must continue by visiting v_1 and v_2, since otherwise v_1 is never traversed. Now the next node w_0 in the Hamiltonian cycle in G' corresponds to an edge (v, w) in G, and by repeating our reasoning we see that our Hamiltonian cycle in G' determines a Hamiltonian cycle in G. We leave it as an easy exercise to check that a Hamiltonian cycle in G yields a Hamiltonian cycle in G'. The process of expanding a directed graph into an undirected graph and the inverse process are illustrated in Figure 10.14 and Figure 10.15.

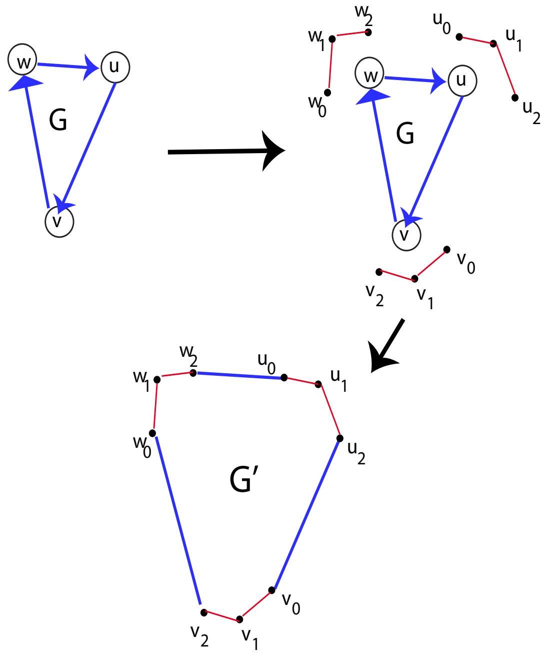


Figure 10.14: Expanding the directed graph into an undirected graph.

(4) **Traveling Salesman Problem**

To show that the **Traveling Salesman Problem** is $\mathcal{NP}$-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it:

Hamiltonian Cycle Problem (Undirected Graphs) $\leq_P$ **Traveling Salesman Problem**

This is a fairly easy reduction.

Given an undirected graph $G = (V, E)$, we construct an instance $\tau(G) = (D, B)$ of the Traveling Salesman Problem so that G has a Hamiltonian cycle iff the traveling salesman problem has a solution. If we let $n = |V|$, we have n cities and the matrix

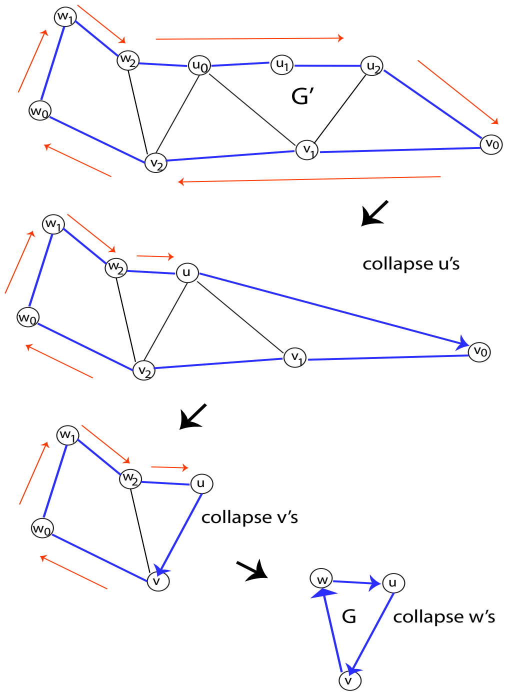


Figure 10.15: Collapsing the undirected graph onto a directed graph.

$D = (d_{ij})$ is defined as follows:

$$d_{ij} = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } \{v_i, v_j\} \in E \\ 2 & \text{otherwise.} \end{cases}$$

We also set the budget B as $B = n$. The construction of (D, B) from G can be done in time $O(n^2)$.

Any tour of the cities has cost equal to n plus the number of pairs (v_i, v_j) such that $i \neq j$ and $\{v_i, v_j\}$ is *not* an edge of G. It follows that a tour of cost n exists iff there are no pairs (v_i, v_j) of the second kind iff the tour is a Hamiltonian cycle.

The reduction from **Hamiltonian Cycle Problem (Undirected Graphs)** to the **Traveling Salesman Problem** is quite simple, but a direct reduction of say **Satisfiability** to the **Traveling Salesman Problem** is hard. By breaking this reduction into several steps made it simpler to achieve.

(5) **Independent Set**

To show that **Independent Set** is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it:

Exact 3-Satisfiability $\leq_P$ Independent Set

Recall that in **Exact 3-Satisfiability** every clause C_i has exactly three literals L_{i1}, L_{i2}, L_{i3}.

Given a set $F = \{C_1, \ldots, C_m\}$ of $m \geq 2$ such clauses, we construct in polynomial time an undirected graph $G = (V, E)$ such that F is satisfiable iff G has an independent set C with at least $K = m$ nodes.

For every i ($1 \leq i \leq m$), we have three nodes c_{i1}, c_{i2}, c_{i3} corresponding to the three literals L_{i1}, L_{i2}, L_{i3} in clause C_i, so there are $3m$ nodes in V. The "core" of G consists of m triangles, one for each set $\{c_{i1}, c_{i2}, c_{i3}\}$. We also have an edge $\{c_{ik}, c_{j\ell}\}$ iff L_{ik} and $L_{j\ell}$ are complementary literals. If the size n of the input is the sum of the lengths of the clauses, then the construction of G can be done in time $O(n^2)$.

Example 10.6. Let F be the set of clauses

$$F = \{C_1 = (x_1 \vee \overline{x_2} \vee x_3),\ C_2 = (\overline{x_1} \vee \overline{x_2} \vee x_3),\ C_3 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}),\ C_4 = (x_1 \vee x_2 \vee x_3)\}.$$

The graph G associated with F is shown in Figure 10.16.

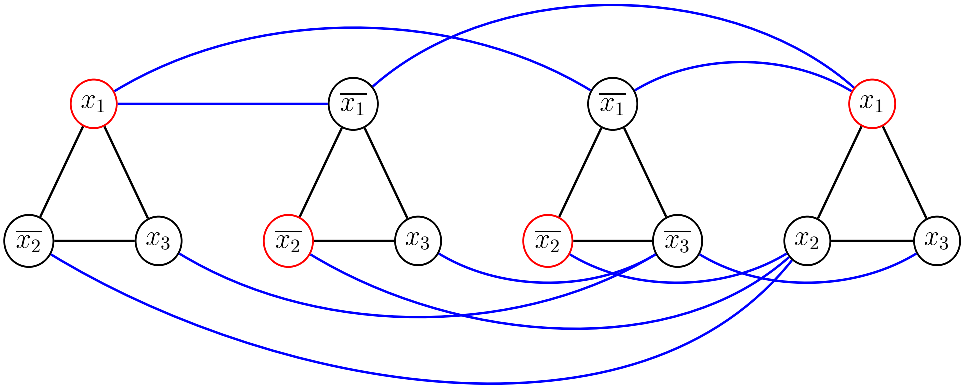


Figure 10.16: The graph constructed from the clauses of Example 10.6.

It remains to show that the construction works. Since any three nodes in a triangle are connected, an independent set C can have at most one node per triangle and thus has at most m nodes. Since the budget is $K = m$, we may assume that there is an independent set with m nodes. Define a (partial) truth assignment by

$$v(x_i) = \begin{cases} \mathbf{T} & \text{if } L_{jk} = x_i \text{ and } c_{jk} \in C \\ \mathbf{F} & \text{if } L_{jk} = \overline{x_i} \text{ and } c_{jk} \in C. \end{cases}$$

Since the non-triangle edges in G link nodes corresponding to complementary literals and nodes in C are not connected, our truth assigment does not assign clashing truth values to the variables x_i. Not all variables may receive a truth value, in which case we assign an arbitrary truth value to the unassigned variables. This yields a satisfying assignment for F.

In Example 10.6, the set $C = \{c_{11}, c_{22}, c_{32}, c_{41}\}$ corresponding to the nodes shown in red in Figure 10.16 form an independent set, and they induce the partial truth assignment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{F}$. The variable x_3 can be assigned an arbitrary value, say $v(x_3) = \mathbf{F}$, and v is indeed a satisfying truth assignment for F.

Conversely, if v is a truth assignment for F, then we obtain an independent set C of size m by picking for each clause C_i a node c_{ik} corresponding to a literal L_{ik} whose value under v is $\mathbf{T}$.

(6) **Clique**

To show that **Clique** is $\mathcal{NP}$-complete, we reduce **Independent Set** to it:

Independent Set $\leq_P$ Clique

The key to the reduction is the notion of the complement of an undirected graph $G = (V, E)$. The *complement* $G^c = (V, E^c)$ of the graph $G = (V, E)$ is the graph with the same set of nodes V as G but there is an edge $\{u, v\}$ (with $u \neq v$) in E^c iff $\{u, v\} \notin E$. Then it is not hard to check that there is a bijection between maximum independent sets in G and maximum cliques in G^c. The reduction consists in constructing from a graph G its complement G^c, and then G has an independent set iff G^c has a clique. Obviously, the reduction can be done in linear time.

This construction is illustrated in Figure 10.17, where a maximum independent set in the graph G is shown in blue and a maximum clique in the graph G^c is shown in red.

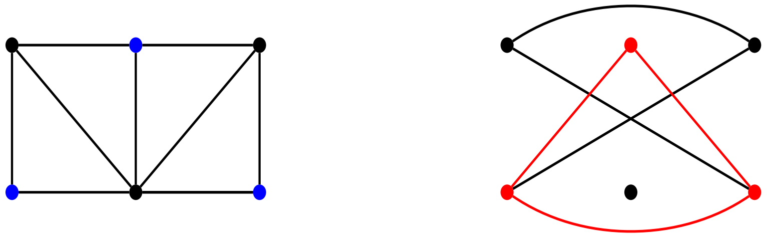


Figure 10.17: A graph (left) and its complement (right).

(7) **Node Cover**

To show that **Node Cover** is $\mathcal{NP}$-complete, we reduce **Independent Set** to it:

Independent Set $\leq_P$ Node Cover

This time the crucial observation is that if N is an independent set in G, then the complement $C = V - N$ of N in V is a node cover in G. Thus there is an independent set of size at least K iff there is a node cover of size at most $n - K$ where $n = |V|$ is the number of nodes in V. The reduction leaves the graph unchanged and replaces K by $n - K$. Obviously, the reduction can be done in linear time. An example is shown in Figure 10.18 where an independent set is shown in blue and a node cover is shown in red.

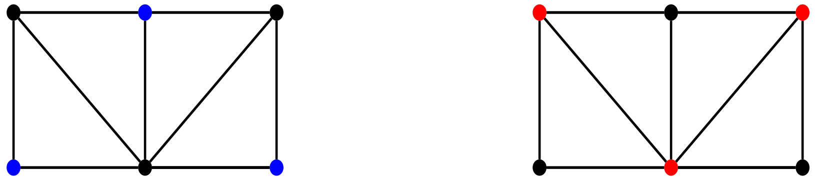


Figure 10.18: An inpendent set (left) and a node cover (right).

(8) **Knapsack (also called Subset sum)**

To show that **Knapsack** is $\mathcal{NP}$-complete, we reduce **Exact Cover** to it:

Exact Cover $\leq_P$ Knapsack

Given an instance $(U, \mathcal{F})$ of set cover with $U = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$, a family of subsets of U, we need to produce in polynomial time an instance $\tau(U, \mathcal{F})$ of the Knapsack Problem consisting of k nonnegative integers $a_1, \ldots, a_k$ and another integer $K > 0$ such that there is a subset $I \subseteq \{1, \ldots, k\}$ such that $\sum_{i \in I} a_i = K$ iff there is an exact cover of U using subsets in $\mathcal{F}$.

The trick here is the relationship between *set union* and *integer addition*.

Example 10.7. Consider the exact cover problem given by $U = \{u_1, u_2, u_3, u_4\}$ and

$$\mathcal{F} = \{S_1 = \{u_3, u_4\},\ S_2 = \{u_2, u_3, u_4\},\ S_3 = \{u_1, u_2\}\}.$$

We can represent each subset S_j by a binary string a_j of length 4, where the ith bit from the left is 1 iff $u_i \in S_j$, and 0 otherwise. In our example

$$\begin{aligned} a_1 &= 0011 \\ a_2 &= 0111 \\ a_3 &= 1100. \end{aligned}$$

Then the trick is that some family $\mathcal{C}$ of subsets S_j is an exact cover if the sum of the corresponding numbers a_j adds up to $1111 = 2^4 - 1 = K$. For example,

$$\mathcal{C} = \{S_1 = \{u_3, u_4\},\ S_3 = \{u_1, u_2\}\}$$

is an exact cover and

$$a_1 + a_3 = 0011 + 1100 = 1111.$$

Unfortunately, there is a problem with this encoding which has to do with the fact that addition may involve carry. For example, assuming four subsets and the universe $U = \{u_1, \ldots, u_6\}$,

$$11 + 13 + 15 + 24 = 63,$$

in binary

$$001011 + 001101 + 001111 + 011000 = 111111,$$

but if we convert these binary strings to the corresponding subsets we get the subsets

$$\begin{aligned} S_1 &= \{u_3, u_5, u_6\} \\ S_2 &= \{u_3, u_4, u_6\} \\ S_3 &= \{u_3, u_4, u_5, u_6\} \\ S_4 &= \{u_2, u_3\}, \end{aligned}$$

which are not disjoint and do not cover U.

The fix is surprisingly simple: use base m (where m is the number of subsets in $\mathcal{F}$) instead of base 2.

Example 10.8. Consider the exact cover problem given by $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$ and $\mathcal{F}$ given by

$$\begin{aligned} S_1 &= \{u_3, u_5, u_6\} \\ S_2 &= \{u_3, u_4, u_6\} \\ S_3 &= \{u_3, u_4, u_5, u_6\} \\ S_4 &= \{u_2, u_3\}, \\ S_5 &= \{u_1, u_2, u_4\}. \end{aligned}$$

In base $m = 5$, the numbers corresponding to $S_1, \ldots, S_5$ are

$$\begin{aligned} a_1 &= 001011\\ a_2 &= 001101\\ a_3 &= 001111\\ a_4 &= 011000\\ a_5 &= 110100. \end{aligned}$$

This time,

$$a_1 + a_2 + a_3 + a_4 = 001011 + 001101 + 001111 + 011000 = 014223 \neq 111111,$$

so $\{S_1, S_2, S_3, S_4\}$ is not a solution. However

$$a_1 + a_5 = 001011 + 110100 = 111111,$$

and $\mathcal{C} = \{S_1, S_5\}$ is an exact cover.

Thus, given an instance $(U, \mathcal{F})$ of **Exact Cover** where $U = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{S_1, \ldots, S_m\}$ the reduction to **Knapsack** consists in forming the m numbers $a_1, \ldots, a_m$ (each of n bits) encoding the subsets S_j, namely $a_{ji} = 1$ iff $u_i \in S_j$, else 0, and to let $K = 1 + m^2 + \cdots + m^{n-1}$, which is represented in base m by the string $\underbrace{11 \cdots 11}_{n}$. In testing whether $\sum_{i \in I} a_i = K$ for some subset $I \subseteq \{1, \ldots, m\}$, we use arithmetic in base m.

If a candidate solution $\mathcal{C}$ involves at most $m - 1$ subsets, then since the corresponding numbers are added in base m, a carry can never happen. If the candidate solution involves all m subsets, then $a_1 + \cdots + a_m = K$ iff $\mathcal{F}$ is a partition of U, since otherwise some bit in the result of adding up these m numbers in base m is not equal to 1, even if a carry occurs. Since the number K is written in binary, it takes time $O(mn)$ to produce $((a_1, \ldots, a_m), K)$ from $(U, \mathcal{F})$.

(9) **Inequivalence of $*$-free Regular Expressions**

To show that **Inequivalence of $*$-free Regular Expressions** is $\mathcal{NP}$-complete, we reduce the **Satisfiability Problem** to it:

Satisfiability Problem $\leq_P$ Inequivalence of $*$-free Regular Expressions

We already argued that **Inequivalence of $*$-free Regular Expressions** is in $\mathcal{NP}$ because if R is a $*$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. The above observation shows that if R_1 and R_2 are $*$-free and if there is a string $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time. It follows that the inequivalence problem for $*$ -free regular expressions is in $\mathcal{NP}$.

We reduce the **Satisfiability Problem** to the **Inequivalence of $*$-free Regular Expressions** as follows. For any set of clauses $P = C_1 \wedge \cdots \wedge C_p$, if the propositional variables occurring in P are $x_1, \ldots, x_n$, we produce two $*$-free regular expressions R, S over $\Sigma = \{0, 1\}$, such that P is satisfiable iff $L_R \neq L_S$. The expression S is actually

$$S = \underbrace{(0+1)(0+1)\cdots(0+1)}_{n}.$$

The expression R is of the form

$$R = R_1 + \cdots + R_p,$$

where R_i is constructed from the clause C_i in such a way that L_{R_i} corresponds precisely to the set of truth assignments that falsify C_i; see below.

Given any clause C_i, let R_i be the $*$-free regular expression defined such that, if x_j and $\overline{x}_j$ both belong to C_i (for some j), then $R_i = \emptyset$, else

$$R_i = R_i^1 \cdot R_i^2 \cdots R_i^n,$$

where R_i^j is defined by

$$R_i^j = \begin{cases} 0 & \text{if } x_j \text{ is a literal of } C_i \\ 1 & \text{if } \overline{x}_j \text{ is a literal of } C_i \\ (0+1) & \text{if } x_j \text{ does not occur in } C_i. \end{cases}$$

The construction of R from P takes linear time.

Example 10.9. If we apply the above conversion to the clauses of Example 10.3, namely

$$F = \{C_1 = (x_1 \vee \overline{x_2}),\ C_2 = (\overline{x_1} \vee x_2 \vee x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \vee \overline{x_3})\},$$

we get

$$R_1 = 0 \cdot 1 \cdot (0+1), \quad R_2 = 1 \cdot 0 \cdot 0, \quad R_3 = (0+1) \cdot 0 \cdot (0+1), \quad R_4 = (0+1) \cdot 1 \cdot 1.$$

Clearly, all truth assignments that falsify C_i must assign **F** to x_j if $x_j \in C_i$ or assign **T** to x_j if $\overline{x}_j \in C_i$. Therefore, L_{R_i} corresponds to the set of truth assignments that falsify C_i (where 1 stands for **T** and 0 stands for **F**) and thus, if we let

$$R = R_1 + \cdots + R_p,$$

then L_R corresponds to the set of truth assignments that falsify $P = C_1 \wedge \cdots \wedge C_p$. Since $L_S = \{0,1\}^n$ (all binary strings of length n), we conclude that $L_R \neq L_S$ iff P is satisfiable. Therefore, we have reduced the **Satisfiability Problem** to our problem and the reduction clearly runs in polynomial time. This proves that the problem of deciding whether $L_R \neq L_S$, for any two $*$-free regular expressions R and S is $\mathcal{NP}$-complete.

(10) 0-1 **integer programming problem**

It is easy to check that the problem is in $\mathcal{NP}$.

To prove that the is $\mathcal{NP}$-complete we reduce the **bounded-tiling problem** to it:

bounded-tiling problem $\leq_P$ 0-1 **integer programming problem**

Given a tiling problem, $((\mathcal{T}, V, H), \widehat{s}, \sigma_0)$, we create a 0-1-valued variable x_{mnt}, such that $x_{mnt} = 1$ iff tile t occurs in position (m, n) in some tiling. Write equations or inequalities expressing that a tiling exists and then use "slack variables" to convert inequalities to equations. For example, to express the fact that every position is tiled by a single tile, use the equation

$$\sum_{t \in \mathcal{T}} x_{mnt} = 1,$$

for all m, n with $1 \leq m \leq 2s$ and $1 \leq n \leq s$. We leave the rest as as exercise.

10.3 Succinct Certificates, co$\mathcal{NP}$, and $\mathcal{EXP}$

All the problems considered in Section 10.1 share a common feature, which is that for each problem, a solution is produced nondeterministically (an exact cover, a directed Hamiltonian cycle, a tour of cities, an independent set, a node cover, a clique *etc.*), and then this candidate solution is checked deterministically and in polynomial time. The candidate solution is a string called a *certificate* (or *witness*).

It turns out that membership on $\mathcal{NP}$ can be defined in terms of certificates. To be a certificate, a string must satisfy two conditions:

1. It must be *polynomially succinct*, which means that its length is at most a polynomial in the length of the input.

2. It must be *checkable* in polynomial time.

All "yes" inputs to a problem in $\mathcal{NP}$ must have at least one certificate, while all "no" inputs must have none.

The notion of certificate can be formalized using the notion of a polynomially balanced language.

Definition 10.3. Let Σ be an alphabet, and let ";" be a symbol not in Σ. A language $L' \subseteq \Sigma^*; \Sigma^*$ is said to be *polynomially balanced* if there exists a polynomial $p(X)$ such that for all $x, y \in \Sigma^*$, if $x; y \in L'$ then $|y| \leq p(|x|)$.

Suppose L' is a polynomially balanced language and that $L' \in \mathcal{P}$. Then we can consider the language

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

The intuition is that for each $x \in L$, the set

$$\{y \in \Sigma^* \mid x; y \in L'\}$$

is the set of certificates of x. For every $x \in L$, a Turing machine can nondeterministically guess one of its certificates y, and then use the deterministic Turing machine for L' to check in polynomial time that $x; y \in L'$. Note that, by definition, strings not in L have no certificate. It follows that $L \in \mathcal{NP}$.

Conversely, if $L \in \mathcal{NP}$ and the alphabet Σ has at least two symbols, we can encode the paths in the computation tree for every input $x \in L$, and we obtain a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ with L' in $\mathcal{P}$ such that

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

The details of this construction are left as an exercise. In summary, we obtain the following theorem.

Theorem 10.1. *Let $L \subseteq \Sigma^*$ be a language over an alphabet Σ with at least two symbols, and let ";" be a symbol not in Σ. Then $L \in \mathcal{NP}$ iff there is a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ such that $L' \in \mathcal{P}$ and*

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)(x; y \in L')\}.$$

Theorem 10.1 shows that the introduction of non-determinstic Turing machines is not really needed to define the class $\mathcal{NP}$, but this extreme point of view is not fruitful.

A striking illustration of the notion of succint certificate is illustrated by the set of *composite* integers, namely those natural numbers $n \in \mathbb{N}$ that can be written as the product pq of two numbers $p, q \geq 2$ with $p, q \in \mathbb{N}$. For example, the number

$$4,294,967,297$$

is a composite!

This is far from obvious, but if an oracle gives us the certificate $\{6,700,417, \; 641\}$, it is easy to carry out in polynomial time the multiplication of these two numbers and check that it is equal to $4,294,967,297$. Finding a certificate is usually (very) hard, but checking that it works is easy. This is the point of certificates.

We conclude this section with a brief discussion of the complexity classes co$\mathcal{NP}$ and $\mathcal{EXP}$.

By definition,

$$\text{co}\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\},$$

that is, co$\mathcal{NP}$ consists of all complements of languages in $\mathcal{NP}$. Since $\mathcal{P} \subseteq \mathcal{NP}$ and $\mathcal{P}$ is closed under complementation,

$$\mathcal{P} \subseteq \text{co}\mathcal{NP},$$

so $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co}\mathcal{NP}$, but nobody knows whether this inclusion is proper or whether $\mathcal{NP}$ is closed under complementation, that is, nobody knows whether $\mathcal{NP} = \text{co}\mathcal{NP}$.

A language L is *co$\mathcal{NP}$-hard* if every language in co$\mathcal{NP}$ is polynomial-time reducible to L, and *co$\mathcal{NP}$-complete* if $L \in \text{co}\mathcal{NP}$ and L is co$\mathcal{NP}$-hard.

What can be shown is that if $\mathcal{NP} \neq \text{co}\mathcal{NP}$, then $\mathcal{P} \neq \mathcal{NP}$. However it is possible that $\mathcal{P} \neq \mathcal{NP}$ and yet $\mathcal{NP} = \text{co}\mathcal{NP}$, although this is considered unlikely.

We have $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co}\mathcal{NP}$, but there are problems in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ not known to be in $\mathcal{P}$. One of the most famous in the following problem:

Integer factorization problem:

Given an integer $N \geq 3$, and another integer M (a budget) such that $1 < M < N$, does N have a factor d with $1 < d \leq M$?

Proposition 10.2. *The problem* **Integer factorization** *is in $\mathcal{NP} \cap \text{co}\mathcal{NP}$.*

Proof. That **Integer factorization** is in $\mathcal{NP}$ is clear. To show that **Integer factorization** is in co$\mathcal{NP}$, we can guess a factorization of N into distinct factors all greater than M, check that they are prime using the results of Chapter 11 showing that testing primality is in $\mathcal{NP}$ (even in $\mathcal{P}$, but that's much harder to prove), and then check that the product of these factors is N. □

It is widely believed that **Integer factorization** *does not* belong to $\mathcal{P}$, which is the technical justification for saying that this problem is hard. Most cryptographic algorithms rely on this unproven fact. If **Integer factorization** was either $\mathcal{NP}$-complete or co$\mathcal{NP}$-complete, then we would have $\mathcal{NP} = \text{co}\mathcal{NP}$, which is considered very unlikely.

Remark: If $\sqrt{N} \leq M < N$, the above problem is equivalent to asking whether N is prime.

A natural instance of a problem in co$\mathcal{NP}$ is the *unsatisfiability problem* for propositions UNSAT $= \neg$SAT, namely deciding that a proposition P has no satisfying assignment.

Definition 10.4. A proposition P (in CNF) is *falsifiable* if there is some truth assigment v such that $\widehat{v}(P) = \mathbf{F}$.

It is obvious that the set of falsifiable propositions is in $\mathcal{NP}$. Since a proposition P is valid iff P is not falsifiable, the *validity (or tautology) problem* TAUT for propositions is in co$\mathcal{NP}$. In fact, the follolwing result holds.

Proposition 10.3. *The problem* TAUT *is* co$\mathcal{NP}$*-complete.*

Proof. See Papadimitriou [32]. Since SAT is $\mathcal{NP}$-complete, for every language $L \in \mathcal{NP}$, there is a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ such that $x \in L$ iff $f(x) \in \text{SAT}$. Then $x \notin L$ iff $f(x) \notin \text{SAT}$, that is, $x \in \overline{L}$ iff $f(x) \in \neg\text{SAT}$, which means that every language $\overline{L} \in \text{co}\mathcal{NP}$ is polynomial-time reducible to $\neg\text{SAT} = \text{UNSAT}$. But $\text{TAUT} = \{\neg P \mid P \in \text{UNSAT}\}$, so we have the polynomial-time computable function g given by $g(x) = \neg f(x)$ which gives us the reduction $x \in \overline{L}$ iff $g(x) \in \text{TAUT}$, which shows that TAUT is co$\mathcal{NP}$-complete. □

Despite the fact that this problem has been extensively studied, not much is known about its exact complexity.

The reasoning used to show that TAUT is co$\mathcal{NP}$-complete can also be used to show the following interesting result.

Proposition 10.4. *If a language L is $\mathcal{NP}$-complete, then its complement $\overline{L}$ is* co$\mathcal{NP}$*-complete.*

Proof. By definition, since $L \in \mathcal{NP}$, we have $\overline{L} \in \text{co}\mathcal{NP}$. Since L is $\mathcal{NP}$-complete, for every language $L_2 \in \mathcal{NP}$, there is a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ such that $x \in L_2$ iff $f(x) \in L$. Then $x \notin L_2$ iff $f(x) \notin L$, that is, $x \in \overline{L_2}$ iff $f(x) \in \overline{L}$, which means that $\overline{L}$ is co$\mathcal{NP}$-hard as well, thus co$\mathcal{NP}$-complete. □

The class $\mathcal{EXP}$ is defined as follows.

Definition 10.5. A deterministic Turing machine M is said to be *exponentially bounded* if there is a polynomial $p(X)$ such that for every input $x \in \Sigma^*$, there is no ID ID_n such that

$$ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > 2^{p(|x|)}.$$

The class $\mathcal{EXP}$ is the class of all languages that are accepted by some exponentially bounded deterministic Turing machine.

Remark: We can also define the class $\mathcal{NEXP}$ as in Definition 10.5, except that we allow nondeterministic Turing machines.

One of the interesting features of $\mathcal{EXP}$ is that it contains $\mathcal{NP}$.

Theorem 10.5. *We have the inclusion $\mathcal{NP} \subseteq \mathcal{EXP}$.*

Sketch of proof. Let M be some nondeterministic Turing machine accepting L in polynomial time bounded by $p(X)$. We can construct a deterministic Turing machine M' that operates as follows: for every input x, M' simulates M on all computations of length 1, then on all possible computations of length 2, and so on, up to all possible computations of length $p(|x|)+1$. At this point, either an accepting computation has been discovered or all computations have halted rejecting. We claim that M' operates in time bounded by $2^{q(|x|)}$ for some polynomial $q(X)$. First, let r be the degree of nondeterminism of M, that is, the maximum number of triples (b, m, q) such that a quintuple (p, q, b, m, q) is an instructions of M. Then to simulate a computation of M of length ℓ, M' needs $O(\ell)$ steps—to copy the input, to produce a string c in $\{1, \ldots, r\}^\ell$, and so simulate M according to the choices specified by c. It follows that M' can carry out the simulation of M on an input x in

$$\sum_{\ell=1}^{p(|x|)+1} r^\ell \leq (r+1)^{p(|x|)+1}$$

steps. Including the $O(\ell)$ extra steps for each ℓ, we obtain the bound $(r+2)^{p(|x|)+1}$. Then we can pick a constant k such that $2^k > r+2$, and with $q(X) = k(p(X)+1)$, we see that M' operates in time bounded by $2^{q(|x|)}$. □

It is also immediate to see that $\mathcal{EXP}$ is closed under complementation. Furthermore the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$ holds.

Theorem 10.6. *We have the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$.*

Sketch of proof. We use a diagonalization argument to produce a language E such that $E \notin \mathcal{P}$, yet $E \in \mathcal{EXP}$. We need to code a Turing machine as a string, but this can certainly be done using the techniques of Chapter 4. Let $\#(M)$ be the code of Turing machine M and let $\#(x)$ be the code of x. Define E as

$$E = \{\#(M)\#(x) \mid M \text{ accepts input } x \text{ after at most } 2^{|x|} \text{ steps}\}.$$

We claim that $E \notin \mathcal{P}$. We proceed by contradiction. If $E \in \mathcal{P}$, then so is the language E_1 given by

$$E_1 = \{\#(M) \mid M \text{ accepts } \#(M) \text{ after at most } 2^{|\#(M)|} \text{ steps}\}.$$

Since $\mathcal{P}$ is closed under complementation, we also have $\overline{E_1} \in \mathcal{P}$. Let M^* be a deterministic Turing machine accepting $\overline{E_1}$ in time $p(X)$, for some polynomial $p(X)$. Since $p(X)$ is a polynomial, there is some n_0 such that $p(n) \leq 2^n$ for all all $n \geq n_0$. We may also assume that $|\#(M^*)| \geq n_0$, since if not we can add n_0 "dead states" to M^*.

Now what happens if we run M^* on its own code $\#(M^*)$?

It is easy to see that we get a contradiction, namely M^* accepts $\#(M^*)$ iff M^* rejects $\#(M^*)$. We leave this verification as an exercise.

In conclusion, $\overline{E_1} \notin \mathcal{P}$, which in turn implies that $E \notin \mathcal{P}$.

It remains to prove that $E \in \mathcal{EXP}$. This is because we can construct a Turing machine that can in exponential time simulate any Turing machine M on input x for $2^{|x|}$ steps. $\square$

In summary, we have the chain of inclusions

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP},$$

where the inclusions $\mathcal{P} \subset \mathcal{EXP}$ is strict (by Theorem 10.6), but the left inclusion and the right inclusion are both open problems, and we know that at least one of these two inclusions is strict.

We also have the inclusions

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP} \subseteq \mathcal{NEXP},$$

where the inclusions $\mathcal{P} \subset \mathcal{EXP}$ and $\mathcal{NP} \subset \mathcal{NEXP}$ are strict. The strict inclusion $\mathcal{NP} \subset \mathcal{NEXP}$ is a consequence of the time hierarchy theorem (Cook, Seiferas, Fischer, Meyer, Zak); see Papadimitriou [32] (Chapters 7 and 20) and Arora and Barak [2] (Chapter 3, Section 3.2). The left inclusion and the right inclusion in $\mathcal{NP} \subseteq \mathcal{EXP} \subseteq \mathcal{NEXP}$ are both open problems, but we know that at least one of these two inclusions is strict. It can be shown that if $\mathcal{EXP} \neq \mathcal{NEXP}$, then $\mathcal{P} \neq \mathcal{NP}$; see Papadimitriou [32].

Chapter 11

Primality Testing is in $\mathcal{NP}$

11.1 Prime Numbers and Composite Numbers

Prime numbers have fascinated mathematicians and more generally curious minds for thousands of years. What is a prime number? Well, $2, 3, 5, 7, 11, 13, \ldots, 9973$ are prime numbers.

Definition 11.1. A positive integer p is *prime* if $p \geq 2$ and if p is only divisible by 1 and p. Equivalently, p is prime if and only if p is a positive integer $p \geq 2$ that is not divisible by any integer m such that $2 \leq m < p$. A positive integer $n \geq 2$ which is not prime is called *composite*.

Observe that the number 1 is considered neither a prime nor a composite. For example, $6 = 2 \cdot 3$ is composite. Is $3\,215\,031\,751$ composite? Yes, because

$$3\,215\,031\,751 = 151 \cdot 751 \cdot 28351.$$

Even though the definition of primality is very simple, the structure of the set of prime numbers is highly nontrivial. The prime numbers are the basic building blocks of the natural numbers because of the following theorem bearing the impressive name of *fundamental theorem of arithmetic*.

Theorem 11.1. *Every natural number $n \geq 2$ has a unique factorization*

$$n = p_1^{i_1} p_2^{i_2} \cdots p_k^{i_k},$$

where the exponents $i_1, \ldots, i_k$ are positive integers and $p_1 < p_2 < \cdots < p_k$ are primes.

Every book on number theory has a proof of Theorem 11.1. The proof is not difficult and uses induction. It has two parts. The first part shows the existence of a factorization. The second part shows its uniqueness. For example, see Apostol [1] (Chapter 1, Theorem 1.10).

How many prime numbers are there? Many! In fact, infinitely many.

Theorem 11.2. *The set of prime numbers is infinite.*

Proof. The following proof attributed to Hermite only use the fact that every integer greater than 1 has some prime divisor. We prove that for every natural number $n \geq 2$, there is some prime $p > n$. Consider $N = n! + 1$. The number N must be divisible by some prime p ($p = N$ is possible). Any prime p dividing N is distinct from $2, 3, \ldots, n$, since otherwise p would divide $N - n! = 1$, a contradiction. □

The problem of determining whether a given integer is prime is one of the better known and most easily understood problems of pure mathematics. This problem has caught the interest of mathematicians again and again for centuries. However, it was not until the 20th century that questions about primality testing and factoring were recognized as problems of practical importance and a central part of applied mathematics. The advent of cryptographic systems that use large primes, such as RSA, was the main driving force for the development of fast and reliable methods for primality testing. Indeed, in order to create RSA keys, one needs to produce large prime numbers.

11.2 Methods for Primality Testing

The general strategy to test whether an integer $n > 2$ is prime or composite is to choose some property, say A, implied by primality, and to search for a counterexample a to this property for the number n, namely some a for which property A fails. We look for properties for which checking that a candidate a is indeed a countexample can be done quickly.

A simple property that is the basis of several primality testing algorithms is the *Fermat test*, namely

$$a^{n-1} \equiv 1 \pmod{n},$$

which means that $a^{n-1} - 1$ is divisible by n (see Definition 11.2 for the meaning of the notation $a \equiv b \pmod{n}$). If n is prime, and if $\gcd(a, n) = 1$, then the above test is indeed satisfied; this is Fermat's little theorem, Theorem 11.7.

Typically, together with the number n being tested for primality, some candidate counterexample a is supplied to an algorithm which runs a test to determine whether a is really a counterexample to property A for n. If the test says that a is a counterexample, also called a *witness*, then we know for sure that n is composite.

For example, using the Fermat test, if $n = 10$ and $a = 3$, we check that

$$3^9 = 19683 = 10 \cdot 1968 + 3,$$

so $3^9 - 1$ is not divisible by 10, which means that

$$a^{n-1} = 3^9 \not\equiv 1 \pmod{10},$$

and the Fermat test fails. This shows that 10 is not prime and that $a = 3$ is a witness of this fact.

If the algorithm reports that a is not a witness to the fact that n is composite, does this imply that n is prime? Unfortunately, no. This is because, there may be some composite number n and some candidate counterexample a for which the test says that a is not a countexample. Such a number a is called a *liar*.

For example, using the Fermat test for $n = 91 = 7 \cdot 13$ and $a = 3$, we can check that

$$a^{n-1} = 3^{90} \equiv 1 \pmod{91},$$

so the Fermat test succeeds even though 91 is not prime. The number $a = 3$ is a liar.

The other reason is that we haven't tested all the candidate counterexamples a for n. In the case where $n = 91$, it is shown in Section 11.6 that $2^{90} - 64$ is divisible by 91, so the Fermat test fails for $a = 2$, which confirms that 91 is not prime, and $a = 2$ is a witness of this fact.

Unfortunately, the Fermat test has the property that it may succeed for all candidate counterexamples, even though n is composite. The number $n = 561 = 3 \cdot 11 \cdot 17$ is such a devious number. It can be shown that for all $a \in \{2, \ldots, 560\}$ such that $\gcd(a, 561) = 1$, we have

$$a^{560} \equiv 1 \pmod{561},$$

so *all* these a are liars.

Such composite numbers for which the Fermat test succeeds for all candidate counterexamples are called *Carmichael numbers*, and unfortunately there are infinitely many of them. Thus the Fermat test is doomed. There are various ways of strengthening the Fermat test, but we will not discuss this here. We refer the interested reader to Crandall and Pomerance [5] and Gallier and Quaintance [15].

The remedy is to make sure that we pick a property A such that if n is composite, then at least some candidate a is not a liar, and to test all potential countexamples a. The difficulty is that trying all candidate countexamples can be too expensive to be practical.

There are two classes of primality testing algorithms:

(1) Algorithms that try all possible countexamples and for which the test does not lie. These algorithms give a definite answer: n is prime or n is composite. Until 2002, no algorithms running in polynomial time were known. The situation changed in 2002 when a paper with the title "PRIMES is in **P**," by Agrawal, Kayal and Saxena, appeared on the website of the Indian Institute of Technology at Kanpur, India. In this paper, it was shown that testing for primality has a deterministic (nonrandomized) algorithm that runs in polynomial time.

    We will not discuss algorithms of this type here, and instead refer the reader to Crandall and Pomerance [5] and Ribenboim [36].

(2) Randomized algorithms. To avoid having problems with infinite events, we assume that we are testing numbers in some large finite interval $\mathcal{I}$. Given any positive integer $m \in \mathcal{I}$, some candidate witness a is chosen at random. We have a test which, given m and a potential witness a, determines whether or not a is indeed a witness to the fact that m is composite. Such an algorithm is a *Monte Carlo* algorithm, which means the following:

(1) *If the test is positive, then $m \in \mathcal{I}$ is composite.* In terms of probabilities, this is expressed by saying that the conditional probability that $m \in \mathcal{I}$ is composite given that the test is positive is equal to 1. If we denote the event that some positive integer $m \in \mathcal{I}$ is composite by C, then we can express the above as

$$\mathsf{Pr}(C \mid \text{test is positive}) = 1.$$

(2) *If $m \in \mathcal{I}$ is composite, then the test is positive for at least* 50% *of the choices for a.* We can express the above as

$$\mathsf{Pr}(\text{test is positive} \mid C) \geq \frac{1}{2}.$$

This gives us a degree of confidence in the test.

The contrapositive of (1) says that if $m \in \mathcal{I}$ is prime, then the test is negative. If we denote by P the event that some positive integer $m \in \mathcal{I}$ is prime, then this is expressed as

$$\mathsf{Pr}(\text{test is negative} \mid P) = 1.$$

If we repeat the test ℓ times by picking independent potential witnesses, then the conditional probability that the test is negative ℓ times given that n is composite, written $\mathsf{Pr}(\text{test is negative } \ell \text{ times} \mid C)$, is given by

$$\begin{aligned}\mathsf{Pr}(\text{test is negative } \ell \text{ times} \mid C) &= \mathsf{Pr}(\text{test is negative} \mid C)^{\ell}\\ &= (1 - \mathsf{Pr}(\text{test is positive} \mid C))^{\ell}\\ &\leq \left(1 - \frac{1}{2}\right)^{\ell}\\ &= \left(\frac{1}{2}\right)^{\ell},\end{aligned}$$

where we used Property (2) of a Monte Carlo algorithm that

$$\mathsf{Pr}(\text{test is positive} \mid C) \geq \frac{1}{2}$$

and the independence of the trials. *This confirms that if we run the algorithm ℓ times, then* $\mathsf{Pr}(\textit{test is negative } \ell \textit{ times} \mid C)$ *is very small.* In other words, it is very unlikely that the test will lie ℓ times (is negative) given that the number $m \in \mathcal{I}$ is composite.

If the probabilty $\mathsf{Pr}(P)$ of the event P is known, which requires knowledge of the distribution of the primes in the interval $\mathcal{I}$, then the conditional probability

$$\mathsf{Pr}(P \mid \text{test is negative } \ell \text{ times})$$

can be determined using Bayes's rule.

A Monte Carlo algorithm does not give a definite answer. However, if ℓ is large enough (say $\ell = 100$), then the conditional probability that the number n being tested is prime given that the test is negative ℓ times, is very close to 1.

Two of the best known randomized algorithms for primality testing are the *Miller–Rabin test* and the *Solovay–Strassen test*. We will not discuss these methods here, and we refer the reader to Gallier and Quaintance [15].

However, what we will discuss is a nondeterministic algorithm that checks that a number n is prime by guessing a certain kind of tree that we call a Lucas tree (because this algorithm is based on a method due to E. Lucas), and then verifies in polynomial time (in the length $\log_2 n$ of the input given in binary) that this tree constitutes a "proof" that n is indeed prime. This shows that primality testing is in $\mathcal{NP}$, a fact that is not obvious at all. Of course, this is a much weaker result than the AKS algorithm, but the proof that the AKS works in polynomial time (in $\log_2 n$) is much harder.

The Lucas test, and basically all of the primality-testing algorithms, use modular arithmetic and some elementary facts of number theory such as the Euler-Fermat theorem, so we proceed with a review of these concepts.

11.3 Modular Arithmetic, the Groups $\mathbb{Z}/n\mathbb{Z}$, $(\mathbb{Z}/n\mathbb{Z})^*$

Recall the fundamental notion of congruence modulo n and its notation due to Gauss (circa 1802).

Definition 11.2. For any $a, b \in \mathbb{Z}$, we write $a \equiv b \pmod{m}$ iff $a - b = km$, for some $k \in \mathbb{Z}$ (in other words, $a - b$ is divisible by m), and we say that *a and b are congruent modulo m*.

For example, $37 \equiv 1 \pmod{9}$, since $37 - 1 = 36 = 4 \cdot 9$. It can also be shown that $200^{250} \equiv 1 \pmod{251}$, but this is impossible to do by brute force, so we will develop some tools to either avoid such computations, or to make them tractable.

It is easy to check that congruence is an equivalence relation but it also satisfies the following properties.

Proposition 11.3. *For any positive integer m, for all $a_1, a_2, b_1, b_2 \in \mathbb{Z}$, the following properties hold. If $a_1 \equiv b_1 \pmod{m}$ and $a_2 \equiv b_2 \pmod{m}$, then*

(1) $a_1 + a_2 \equiv b_1 + b_2 \pmod{m}$.

(2) $a_1 - a_2 \equiv b_1 - b_2 \pmod{m}$.

(3) $a_1a_2 \equiv b_1b_2 \pmod{m}$.

Proof. We only check (3), leaving (1) and (2) as easy exercises. Because $a_1 \equiv b_1 \pmod{m}$ and $a_2 \equiv b_2 \pmod{m}$, we have $a_1 = b_1 + k_1m$ and $a_2 = b_2 + k_2m$, for some $k_1, k_2 \in \mathbb{Z}$, so we obtain

$$\begin{aligned} a_1a_2 - b_1b_2 &= a_1(a_2 - b_2) + (a_1 - b_1)b_2 \\ &= (a_1k_2 + k_1b_2)m. \end{aligned}$$ □

Proposition 11.3 allows us to define addition, subtraction, and multiplication on equivalence classes modulo m.

Definition 11.3. Given any positive integer m, we denote by $\mathbb{Z}/m\mathbb{Z}$ the set of equivalence classes modulo m. If we write $\overline{a}$ for the equivalence class of $a \in \mathbb{Z}$, then we define addition, subtraction, and multiplication on residue classes as follows:

$$\begin{aligned} \overline{a} + \overline{b} &= \overline{a + b} \\ \overline{a} - \overline{b} &= \overline{a - b} \\ \overline{a} \cdot \overline{b} &= \overline{ab}. \end{aligned}$$

The above operations make sense because $\overline{a + b}$ does not depend on the representatives chosen in the equivalence classes $\overline{a}$ and $\overline{b}$, and similarly for $\overline{a - b}$ and $\overline{ab}$. Each equivalence class $\overline{a}$ contains a unique representative from the set of remainders $\{0, 1, \ldots, m-1\}$, modulo m, so the above operations are completely determined by $m \times m$ tables. Using the arithmetic operations of $\mathbb{Z}/m\mathbb{Z}$ is called *modular arithmetic*.

The addition tables of $\mathbb{Z}/n\mathbb{Z}$ for $n = 2, 3, 4, 5, 6, 7$ are shown below.

$n = 2$

+	0	1
0	0	1
1	1	0

$n = 3$

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$n = 4$

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$n = 5$

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$n = 6$

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

$n = 7$

+	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

It is easy to check that the addition operation $+$ is commutative (abelian), associative, that 0 is an identity element for $+$, and that every element a has $-a$ as additive inverse, which means that

$$a + (-a) = (-a) + a = 0.$$

The set $\mathbb{Z}/n\mathbb{Z}$ of residue classes modulo n is a group under addition, a notion defined formally in Definition 11.4

It is easy to check that the multiplication operation $\cdot$ is commutative (abelian), associative, that 1 is an identity element for $\cdot$, and that $\cdot$ is distributive on the left and on the right with respect to addition. We usually suppress the dot and write $\overline{a}\,\overline{b}$ instead of $\overline{a} \cdot \overline{b}$. The multiplication tables of $\mathbb{Z}/n\mathbb{Z}$ for $n = 2, 3, \ldots, 9$ are shown below. Since $0 \cdot m = m \cdot 0 = 0$ for all m, these tables are only given for nonzero arguments.

$n = 2$

·	1
1	1

$n = 3$

·	1	2
1	1	2
2	2	1

$n = 4$

·	**1**	2	**3**
1	**1**	2	**3**
2	2	0	2
3	**3**	2	**1**

$n = 5$

·	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

$n = 6$

·	**1**	2	3	4	**5**
1	**1**	2	3	4	**5**
2	2	4	0	2	4
3	3	0	3	0	3
4	4	2	0	4	2
5	**5**	4	3	2	**1**

$n = 7$

·	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

$n = 8$

·	**1**	2	**3**	4	**5**	6	**7**
1	**1**	2	**3**	4	**5**	6	**7**
2	2	4	6	0	2	4	6
3	**3**	6	**1**	4	**7**	2	**5**
4	4	0	4	0	4	0	4
5	**5**	2	**7**	4	**1**	6	**3**
6	6	4	2	0	6	4	2
7	**7**	6	**5**	4	**3**	2	**1**

$n = 9$

·	**1**	**2**	3	**4**	**5**	6	**7**	**8**
1	**1**	**2**	3	**4**	**5**	6	**7**	**8**
2	**2**	**4**	6	**8**	**1**	3	**5**	**7**
3	3	6	0	3	6	0	3	6
4	**4**	**8**	3	**7**	**2**	6	**1**	**5**
5	**5**	**1**	6	**2**	**7**	3	**8**	**4**
6	6	3	0	6	3	0	6	3
7	**7**	**5**	3	**1**	**8**	6	**4**	**2**
8	**8**	**7**	6	**5**	**4**	3	**2**	**1**

Examining the above tables, we observe that for $n = 2, 3, 5, 7$, which are primes, every element has an inverse, which means that for every nonzero element a, there is some (actually, unique) element b such that

$$a \cdot b = b \cdot a = 1.$$

For $n = 2, 3, 5, 7$, the set $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is an abelian group under multiplication (see Definition 11.4). When n is composite, there exist nonzero elements whose product is zero. For example, when $n = 6$, we have $3 \cdot 2 = 0$, when $n = 8$, we have $4 \cdot 4 = 0$, when $n = 9$, we have $6 \cdot 6 = 0$.

For $n = 4, 6, 8, 9$, the elements a that have an inverse are precisely those that are relatively prime to the modulus n (that is, $\gcd(a, n) = 1$).

These observations hold in general. Recall the Bezout criterion (Proposition 7.3): two nonzero integers $m, n \in \mathbb{Z}$ are relatively prime ($\gcd(m, n) = 1$) iff there are integers $a, b \in \mathbb{Z}$ such that

$$am + bn = 1.$$

Proposition 11.4. *Given any integer $n \geq 1$, for any $a \in \mathbb{Z}$, the residue class $\overline{a} \in \mathbb{Z}/n\mathbb{Z}$ is invertible with respect to multiplication iff* $\gcd(a, n) = 1$.

Proof. If $\overline{a}$ has inverse $\overline{b}$ in $\mathbb{Z}/n\mathbb{Z}$, then $\overline{a}\,\overline{b} = 1$, which means that

$$ab \equiv 1 \pmod{n},$$

that is $ab = 1 + nk$ for some $k \in \mathbb{Z}$, which is the Bezout identity

$$ab - nk = 1$$

and implies that $\gcd(a, n) = 1$. Conversely, if $\gcd(a, n) = 1$, then by Bezout's identity there exist $u, v \in \mathbb{Z}$ such that

$$au + nv = 1,$$

so $au = 1 - nv$, that is,

$$au \equiv 1 \pmod{n},$$

which means that $\overline{a}\,\overline{u} = 1$, so $\overline{a}$ is invertible in $\mathbb{Z}/n\mathbb{Z}$. □

We have alluded to the notion of a group. Here is the formal definition.

Definition 11.4. A *group* is a set G equipped with a binary operation $\cdot \colon G \times G \to G$ that associates an element $a \cdot b \in G$ to every pair of elements $a, b \in G$, and having the following properties: $\cdot$ is associative, has an identity element $e \in G$, and every element in G is invertible (w.r.t. $\cdot$). More explicitly, this means that the following equations hold for all $a, b, c \in G$:

(G1) $a \cdot (b \cdot c) = (a \cdot b) \cdot c.$ (associativity);

(G2) $a \cdot e = e \cdot a = a.$ (identity);

(G3) For every $a \in G$, there is some $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e$. (inverse).

A group G is *abelian* (or *commutative*) if

$$a \cdot b = b \cdot a \quad \text{for all } a, b \in G.$$

It is easy to show that the element e satisfying property (G2) is unique, and for any $a \in G$, the element $a^{-1} \in G$ satisfying $a \cdot a^{-1} = a^{-1} \cdot a = e$ required to exist by (G3) is actually unique. This element is called *the* inverse of a.

The set of integers $\mathbb{Z}$ with the addition operation is an abelian group with identity element 0. The set $\mathbb{Z}/n\mathbb{Z}$ of residues modulo m is an abelian group under addition with identity element 0. In general, $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is *not* a group under multiplication, because some nonzero elements may not have an inverse. However, by Proposition 11.4, if p is prime, then $\mathbb{Z}/n\mathbb{Z} - \{0\}$ *is* an abelian group under multiplication.

When p is not prime, the subset of elements, shown in boldface in the multiplication tables, forms an abelian group under multiplication.

Definition 11.5. The group (under multiplication) of invertible elements of the ring $\mathbb{Z}/n\mathbb{Z}$ is denoted by $(\mathbb{Z}/n\mathbb{Z})^*$. Note that this group is abelian and only defined if $n \geq 2$.

Definition 11.6. If G is a finite group, the number of elements in G is called the *the order* of G.

Given a group G with identity element e, and any element $g \in G$, we often need to consider the powers of g defined as follows.

Definition 11.7. Given a group G with identity element e, for any nonnegative integer n, it is natural to define the *power* g^n of g as follows:

$$\begin{aligned} g^0 &= e \\ g^{n+1} &= g \cdot g^n. \end{aligned}$$

Using induction, it is easy to show that

$$g^m g^n = g^{n+m}$$

for all $m, n \in \mathbb{N}$.

Since g has an inverse g^{-1}, we can extend the definition of g^n to negative powers. For $n \in \mathbb{Z}$, with $n < 0$, let

$$g^n = (g^{-1})^{-n}.$$

Then it is easy to prove that

$$\begin{aligned}
g^i \cdot g^j &= g^{i+j} \\
(g^i)^{-1} &= g^{-i} \\
g^i \cdot g^j &= g^j \cdot g^i
\end{aligned}$$

for all $i, j \in \mathbb{Z}$.

Given a finite group G of order n, for any element $a \in G$, it is natural to consider the set of powers $\{e, a^1, a^2, \ldots, a^k, \ldots\}$. A crucial fact is that there is a smallest positive $s \in \mathbb{N}$ such that $a^s = e$, and that s divides n.

Proposition 11.5. *Let G be a finite group of order n. For every element $a \in G$, the following facts hold:*

(1) There is a smallest positive integer $s \leq n$ such that $a^s = e$.

(2) The set $\{e, a, \ldots, a^{s-1}\}$ is an abelian group denoted $\langle a \rangle$.

(3) We have $a^n = e$, and the positive integer s divides n, More generally, for any positive integer m, if $a^m = e$, then s divides m.

Proof. (1) Consider the sequence of $n+1$ elements

$$(e, a^1, a^2, \ldots, a^n).$$

Since G only has n distinct elements, by the pigeonhole principle, there exist i, j such that $0 \leq i < j \leq n$ such that

$$a^i = a^j.$$

By multiplying both sides by $(a^i)^{-1} = a^{-i}$, we get

$$e = a^i (a^i)^{-1} = a^j (a^i)^{-1} = a^j a^{-i} = a^{j-i}.$$

Since $0 \leq i < j \leq n$, we have $0 \leq j - i \leq n$ with $a^{j-i} = e$. Thus there is some s with $0 < s \leq n$ such that $a^s = e$, and thus a smallest such s.

(2) Since $a^s = e$, for any $i, j \in \{0, \ldots, s-1\}$ if we write $i + j = sq + r$ with $0 \leq r \leq s-1$, we have

$$a^i a^j = a^{i+j} = a^{sq+r} = a^{sq} a^r = (a^s)^q a^r = e^q a^r = a^r,$$

so $\langle a \rangle$ is closed under multiplication. We have $e \in \langle a \rangle$ and the inverse of a^i is a^{s-i}, so $\langle a \rangle$ is a group. This group is obviously abelian.

(3) For any element $g \in G$, let $g\langle a \rangle = \{g a^k \mid 0 \leq k \leq s-1\}$. Observe that for any $i \in \mathbb{N}$, we have

$$a^i \langle a \rangle = \langle a \rangle.$$

We claim that for any two elements $g_1, g_2 \in G$, if $g_1\langle a \rangle \cap g_2\langle a \rangle \neq \emptyset$, then $g_1\langle a \rangle = g_2\langle a \rangle$.

Proof of the claim. If $g \in g_1\langle a\rangle \cap g_2\langle a\rangle$, then there exist $i, j \in \{0, \ldots, s-1\}$ such that

$$g_1 a^i = g_2 a^j.$$

Without loss of generality, we may assume that $i \geq j$. By multipliying both sides by $(a^j)^{-1}$, we get

$$g_2 = g_1 a^{i-j}.$$

Consequently

$$g_2\langle a\rangle = g_1 a^{i-j}\langle a\rangle = g_1\langle a\rangle,$$

as claimed. □

It follows that the pairwise disjoint nonempty subsets of the form $g\langle a\rangle$, for $g \in G$, form a partition of G. However, the map φ_g from $\langle a\rangle$ to $g\langle a\rangle$ given by $\varphi_g(a^i) = ga^i$ has for inverse the map $\varphi_{g^{-1}}$, so φ_g is a bijection, and thus the subsets $g\langle a\rangle$ all have the same number of elements s. Since these subsets form a partition of G, we must have $n = sq$ for some $q \in \mathbb{N}$, which implies that $a^n = e$.

If $g^m = 1$, then writing $m = sq + r$, with $0 \leq r < s$, we get

$$1 = g^m = g^{sq+r} = (g^s)^q \cdot g^r = g^r,$$

so $g^r = 1$ with $0 \leq r < s$, contradicting the minimality of s, so $r = 0$ and s divides m. □

Definition 11.8. Given a finite group G of order n, for any $a \in G$, the smallest positive integer $s \leq n$ such that $a^s = e$ in (1) of Proposition 11.5 is called the *order* of a.

The *Euler φ-function* plays an important role in the theory of the groups $(\mathbb{Z}/n\mathbb{Z})^*$.

Definition 11.9. Given any positive integer $n \geq 1$, the *Euler φ-function* (or Euler *totient function*) is defined such that $\varphi(n)$ is the number of integers a, with $1 \leq a \leq n$, which are relatively prime to n; that is, with $\gcd(a, n) = 1$.[1]

If p is prime, then by definition

$$\varphi(p) = p - 1.$$

We leave it as an exercise to show that if p is prime and if $k \geq 1$, then

$$\varphi(p^k) = p^{k-1}(p-1).$$

It can also be shown that if $\gcd(m, n) = 1$, then

$$\varphi(mn) = \varphi(m)\varphi(n).$$

[1] We allow $a = n$ to accomodate the special case $n = 1$.

The above properties yield a method for computing $\varphi(n)$, based on its prime factorization. If $n = p_1^{i_1} \cdots p_k^{i_k}$, then

$$\varphi(n) = p_1^{i_1-1} \cdots p_k^{i_k-1}(p_1 - 1) \cdots (p_k - 1).$$

For example, $\varphi(17) = 16$, $\varphi(49) = 7 \cdot 6 = 42$,

$$\varphi(900) = \varphi(2^2 \cdot 3^2 \cdot 5^2) = 2 \cdot 3 \cdot 5 \cdot 1 \cdot 2 \cdot 4 = 240.$$

Proposition 11.4 shows that $(\mathbb{Z}/n\mathbb{Z})^*$ has $\varphi(n)$ elements. It also shows that $\mathbb{Z}/n\mathbb{Z} - \{0\}$ is a group (under multiplication) iff n is prime.

For any integer $n \geq 2$, let $(\mathbb{Z}/n\mathbb{Z})^*$ be the group of invertible elements of the ring $\mathbb{Z}/n\mathbb{Z}$. This is a group of order $\varphi(n)$. Then Proposition 11.5 yields the following result.

Theorem 11.6. *(Euler) For any integer $n \geq 2$ and any $a \in \{1, \ldots, n-1\}$ such that* $\gcd(a, n) = 1$*, we have*

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

In particular, if n is a prime, then $\varphi(n) = n - 1$, and we get Fermat's little theorem.

Theorem 11.7. *(Fermat's little theorem) For any prime p and any $a \in \{1, \ldots, p-1\}$, we have*

$$a^{p-1} \equiv 1 \pmod{p}.$$

Since 251 is prime, and since $\gcd(200, 252) = 1$, Fermat's little theorem implies our earlier claim that $200^{250} \equiv 1 \pmod{251}$, without making any computations.

Proposition 11.5 suggests considering groups of the form $\langle g \rangle$.

Definition 11.10. A finite group G is *cyclic* iff there is some element $g \in G$ such that $G = \langle g \rangle$. An element $g \in G$ with this property is called a *generator* of G.

Even though, in principle, a finite cyclic group has a very simple structure, finding a generator for a finite cyclic group is generally hard. For example, it turns out that the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$ is a cyclic group when p is prime, but no efficient method for finding a generator for $(\mathbb{Z}/p\mathbb{Z})^*$ is known (besides a brute-force search).

Examining the multiplication tables for $(\mathbb{Z}/n\mathbb{Z})^*$ for $n = 3, 4, \ldots, 9$, we can check the following facts:

1. 2 is a generator for $(\mathbb{Z}/3\mathbb{Z})^*$.

2. 3 is a generator for $(\mathbb{Z}/4\mathbb{Z})^*$.

3. 2 is a generator for $(\mathbb{Z}/5\mathbb{Z})^*$.

4. 5 is a generator for $(\mathbb{Z}/6\mathbb{Z})^*$.

5. 3 is a generator for $(\mathbb{Z}/7\mathbb{Z})^*$.

6. Every element of $(\mathbb{Z}/8\mathbb{Z})^*$ satisfies the equation $a^2 = 1 \pmod 8$, thus $(\mathbb{Z}/8\mathbb{Z})^*$ has no generators.

7. 2 is a generator for $(\mathbb{Z}/9\mathbb{Z})^*$.

More generally, it can be shown that the multiplicative groups $(\mathbb{Z}/p^k\mathbb{Z})^*$ and $(\mathbb{Z}/2p^k\mathbb{Z})^*$ are cyclic groups when p is an odd prime and $k \geq 1$.

Definition 11.11. A generator of the group $(\mathbb{Z}/n\mathbb{Z})^*$ (when there is one), is called a *primitive root modulo n*.

As an exercise, the reader should check that the next value of n for which $(\mathbb{Z}/n\mathbb{Z})^*$ has no generator is $n = 12$.

The following theorem due to Gauss can be shown. For a proof, see Apostol [1] or Gallier and Quaintance [15].

Theorem 11.8. *(Gauss) For every odd prime p, the group $(\mathbb{Z}/p\mathbb{Z})^*$ is cyclic of order $p-1$. It has $\varphi(p-1)$ generators.*

According to Definition 11.11, the generators of $(\mathbb{Z}/p\mathbb{Z})^*$ are the *primitive roots modulo p*.

11.4 The Lucas Theorem

In this section we discuss an application of the existence of primitive roots in $(\mathbb{Z}/p\mathbb{Z})^*$ where p is an odd prime, known an the *$n-1$ test.* This test due to E. Lucas determines whether a positive odd integer n is prime or not by examining the prime factors of $n-1$ and checking some congruences.

The $n-1$ test can be described as the construction of a certain kind of tree rooted with n, and it turns out that the number of nodes in this tree is bounded by $2\log_2 n$, and that the number of modular multiplications involved in checking the congruences is bounded by $2\log_2^2 n$.

When we talk about the complexity of algorithms dealing with numbers, we assume that all inputs (to a Turing machine) are strings representing these numbers, typically in base 2. Since the length of the binary representation of a natural number $n \geq 1$ is $\lfloor \log_2 n \rfloor + 1$ (or $\lceil \log_2(n+1) \rceil$, which allows $n = 0$), the complexity of algorithms dealing with (nonzero) numbers $m, n,$ *etc.* is expressed in terms of $\log_2 m, \log_2 n,$ *etc.* Recall that for any real number $x \in \mathbb{R}$, the *floor of* x is the greatest integer $\lfloor x \rfloor$ that is less that or equal to x, and the *ceiling of* x is the least integer $\lceil x \rceil$ that is greater that or equal to x.

If we choose to represent numbers in base 10, since for any base b we have $\log_b x = \ln x / \ln b$, we have

$$\log_2 x = \frac{\ln 10}{\ln 2} \log_{10} x.$$

Since $(\ln 10)/(\ln 2) \approx 3.322 \approx 10/3$, we see that the number of decimal digits needed to represent the integer n in base 10 is approximately 30% of the number of bits needed to represent n in base 2.

Since the Lucas test yields a tree such that the number of modular multiplications involved in checking the congruences is bounded by $2\log_2^2 n$, it is not hard to show that testing whether or not a positive integer n is prime, a problem denoted PRIMES, belongs to the complexity class $\mathcal{NP}$. This result was shown by V. Pratt [34] (1975), but Peter Freyd told me that it was "folklore." Since 2002, thanks to the AKS algorithm, we know that PRIMES actually belongs to the class $\mathcal{P}$, but this is a much harder result.

Here is Lehmer's version of the Lucas result, from 1876.

Theorem 11.9. *(Lucas theorem) Let n be a positive integer with $n \geq 2$. Then n is prime iff there is some integer $a \in \{1, 2, \ldots, n-1\}$ such that the following two conditions hold:*

(1) $a^{n-1} \equiv 1 \pmod{n}$.

(2) If $n > 2$, then $a^{(n-1)/q} \not\equiv 1 \pmod{n}$ for all prime divisors q of $n - 1$.

Proof. First assume that Conditions (1) and (2) hold. If $n = 2$, since 2 is prime, we are done. Thus assume that $n \geq 3$, and let r be the order of a (we are working in the abelian group $(\mathbb{Z}/n\mathbb{Z})^*$). We claim that $r = n - 1$. The condition $a^{n-1} \equiv 1 \pmod{n}$ implies that r divides $n - 1$. Suppose that $r < n - 1$, and let q be a prime divisor of $(n-1)/r$ (so q divides $n-1$). Since r is the order of a we have $a^r \equiv 1 \pmod{n}$, so we get

$$a^{(n-1)/q} \equiv a^{r(n-1)/(rq)} \equiv (a^r)^{(n-1)/(rq)} \equiv 1^{(n-1)/(rq)} \equiv 1 \pmod{n},$$

contradicting Condition (2). Therefore, $r = n - 1$, as claimed.

We now show that n must be prime. Now $a^{n-1} \equiv 1 \pmod{n}$ implies that a and n are relatively prime so by Euler's theorem (Theorem 11.6),

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

Since the order of a is $n - 1$, we have $n - 1 \leq \varphi(n)$. If $n \geq 3$ is not prime, then n has some prime divisor p, but n and p are integers in $\{1, 2, \ldots, n\}$ that are not relatively prime to n, so by definition of $\varphi(n)$, we have $\varphi(n) \leq n - 2$, contradicting the fact that $n - 1 \leq \varphi(n)$. Therefore, n must be prime.

Conversely, assume that n is prime. If $n = 2$, then we set $a = 1$. Otherwise, pick a to be any primitive root modulo p. □

Clearly, if $n > 2$ then we may assume that $a \geq 2$. The main difficulty with the $n-1$ test is not so much guessing the primitive root a, but finding a *complete prime factorization* of $n-1$. However, as a nondeterministic algorithm, the $n-1$ test yields a "proof" that a number n is indeed prime which can be represented as a tree, and the number of operations needed to check the required conditions (the congruences) is bounded by $c \log_2^2 n$ for some positive constant c, and this implies that testing primality is in $\mathcal{NP}$.

Before explaining the details of this method, we sharpen slightly Lucas theorem to deal only with odd prime divisors.

Theorem 11.10. *Let n be a positive odd integer with $n \geq 3$. Then n is prime iff there is some integer $a \in \{2, \ldots, n-1\}$ (a guess for a primitive root modulo n) such that the following two conditions hold:*

(1b) $a^{(n-1)/2} \equiv -1 \pmod{n}$.

(2b) If $n-1$ is not a power of 2, then $a^{(n-1)/2q} \not\equiv -1 \pmod{n}$ for all odd prime divisors q of $n-1$.

Proof. Assume that Conditions (1b) and (2b) of Theorem 11.10 hold. Then we claim that Conditions (1) and (2) of Theorem 11.9 hold. By squaring the congruence $a^{(n-1)/2} \equiv -1 \pmod{n}$, we get $a^{n-1} \equiv 1 \pmod{n}$, which is Condition (1) of Theorem 11.9. Since $a^{(n-1)/2} \equiv -1 \pmod{n}$, Condition (2) of Theorem 11.9 holds for $q = 2$. Next, if q is an odd prime divisor of $n-1$, let $m = a^{(n-1)/2q}$. Condition (1b) means that

$$m^q \equiv a^{(n-1)/2} \equiv -1 \pmod{n}.$$

Now if $m^2 \equiv a^{(n-1)/q} \equiv 1 \pmod{n}$, since q is an odd prime, we can write $q = 2k+1$ for some $k \geq 1$, and then

$$m^q \equiv m^{2k+1} \equiv (m^2)^k m \equiv 1^k m \equiv m \pmod{n},$$

and since $m^q \equiv -1 \pmod{n}$, we get

$$m \equiv -1 \pmod{n}$$

(regardless of whether n is prime or not). Thus we proved that if $m^q \equiv -1 \pmod{n}$ and $m^2 \equiv 1 \pmod{n}$, then $m \equiv -1 \pmod{n}$. By contrapositive, we see that if $m \not\equiv -1 \pmod{n}$, then $m^2 \not\equiv 1 \pmod{n}$ or $m^q \not\equiv -1 \pmod{n}$, but since $m^q \equiv a^{(n-1)/2} \equiv -1 \pmod{n}$ by Condition (1a), we conclude that $m^2 \equiv a^{(n-1)/q} \not\equiv 1 \pmod{n}$, which is Condition (2) of Theorem 11.9. But then Theorem 11.9 implies that n is prime.

Conversely, assume that n is an odd prime, and let a be any primitive root modulo n. Then by little Fermat we know that

$$a^{n-1} \equiv 1 \pmod{n},$$

so

$$(a^{(n-1)/2} - 1)(a^{(n-1)/2} + 1) \equiv 0 \pmod{n}.$$

Since n is prime, either $a^{(n-1)/2} \equiv 1 \pmod{n}$ or $a^{(n-1)/2} \equiv -1 \pmod{n}$, but since a generates $(\mathbb{Z}/n\mathbb{Z})^*$, it has order $n-1$, so the congruence $a^{(n-1)/2} \equiv 1 \pmod{n}$ is impossible, and Condition (1b) must hold. Similarly, if we had $a^{(n-1)/2q} \equiv -1 \pmod{n}$ for some odd prime divisor q of $n-1$, then by squaring we would have

$$a^{(n-1)/q} \equiv 1 \pmod{n},$$

and a would have order at most $(n-1)/q < n-1$, which is absurd. □

11.5 Lucas Trees

If n is an odd prime, we can use Theorem 11.10 to build recursively a tree which is a proof, or certificate, of the fact that n is indeed prime. We first illustrate this process with the prime $n = 1279$.

Example 11.1. If $n = 1279$, then we easily check that $n - 1 = 1278 = 2 \cdot 3^2 \cdot 71$. We build a tree whose root node contains the triple $(1279, ((2, 1), (3, 2), (71, 1)), 3)$, where $a = 3$ is the guess for a primitive root modulo 1279. In this simple example, it is clear that 3 and 71 are prime, but we must supply proofs that these number are prime, so we recursively apply the process to the odd divisors 3 and 71.

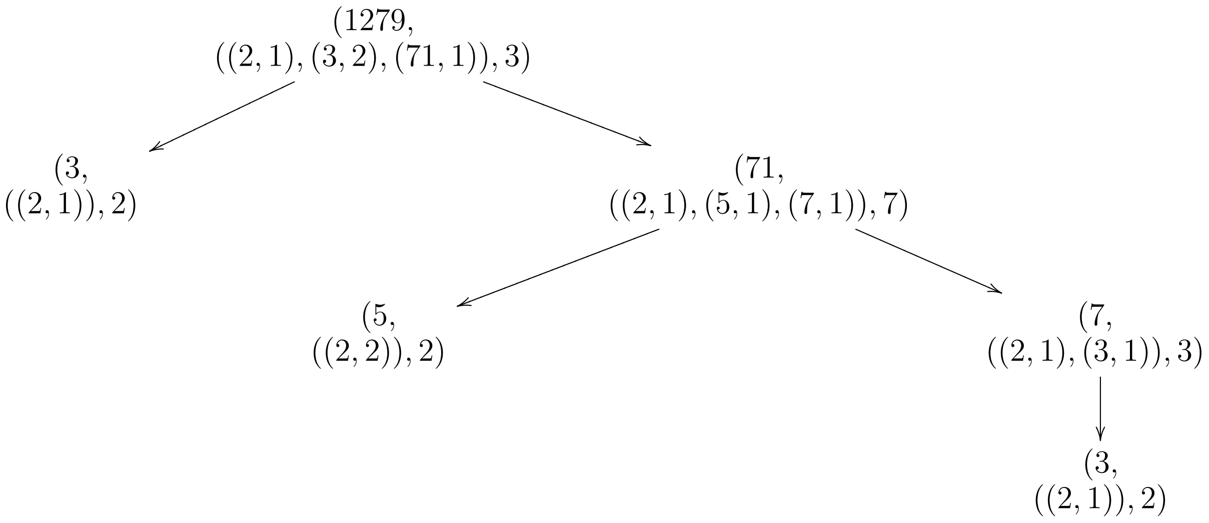
Since $3 - 1 = 2^1$ is a power of 2, we create a one-node tree $(3, ((2, 1)), 2)$, where $a = 2$ is a guess for a primitive root modulo 3. This is a leaf node.

Since $71-1 = 70 = 2\cdot5\cdot7$, we create a tree whose root node is $(71, ((2, 1), (5, 1), (7, 1)), 7)$, where $a = 7$ is the guess for a primitive root modulo 71. Since $5 - 1 = 4 = 2^2$, and $7 - 1 = 6 = 2 \cdot 3$, this node has two successors $(5, ((2, 2)), 2)$ and $(7, ((2, 1), (3, 1)), 3)$, where 2 is the guess for a primitive root modulo 5, and 3 is the guess for a primitive root modulo 7.

Since $4 = 2^2$ is a power of 2, the node $(5, ((2, 2)), 2)$ is a leaf node.

Since $3 - 1 = 2^1$, the node $(7, ((2, 1), (3, 1)), 3)$ has a single successor, $(3, ((2, 1)), 2)$, where $a = 2$ is a guess for a primitive root modulo 3. Since $2 = 2^1$ is a power of 2, the node $(3, ((2, 1)), 2)$ is a leaf node.

To recap, we obtain the following tree:



We still have to check that the relevant congruences hold at every node. For the root node $(1279, ((2, 1), (3, 2), (71, 1)), 3)$, we check that

$$3^{1278/2} \equiv 3^{864} \equiv -1 \pmod{1279} \tag{1b}$$

$$3^{1278/(2\cdot 3)} \equiv 3^{213} \equiv 775 \pmod{1279} \tag{2b}$$

$$3^{1278/(2\cdot 71)} \equiv 3^{9} \equiv 498 \pmod{1279}. \tag{2b}$$

Assuming that 3 and 71 are prime, the above congruences check that Conditions (1a) and (2b) are satisfied, and by Theorem 11.10 this proves that 1279 is prime. We still have to certify that 3 and 71 are prime, and we do this recursively.

For the leaf node $(3, ((2, 1)), 2)$, we check that

$$2^{2/2} \equiv -1 \pmod{3}. \tag{1b}$$

For the node $(71, ((2, 1), (5, 1), (7, 1)), 7)$, we check that

$$7^{70/2} \equiv 7^{35} \equiv -1 \pmod{71} \tag{1b}$$

$$7^{70/(2\cdot 5)} \equiv 7^{7} \equiv 14 \pmod{71} \tag{2b}$$

$$7^{70/(2\cdot 7)} \equiv 7^{5} \equiv 51 \pmod{71}. \tag{2b}$$

Now we certified that 3 and 71 are prime, assuming that 5 and 7 are prime, which we now establish.

For the leaf node $(5, ((2, 2)), 2)$, we check that

$$2^{4/2} \equiv 2^{2} \equiv -1 \pmod{5}. \tag{1b}$$

For the node $(7, ((2, 1), (3, 1)), 3)$, we check that

$$3^{6/2} \equiv 3^3 \equiv -1 \pmod{7} \tag{1b}$$

$$3^{6/(2\cdot 3)} \equiv 3^1 \equiv 3 \pmod{7}. \tag{2b}$$

We have certified that 5 and 7 are prime, given that 3 is prime, which we finally verify.

At last, for the leaf node $(3, ((2, 1)), 2)$, we check that

$$2^{2/2} \equiv -1 \pmod{3}. \tag{1b}$$

The above example suggests the following definition.

Definition 11.12. Given any odd integer $n \geq 3$, a *pre-Lucas tree for* n is defined inductively as follows:

(1) It is a one-node tree labeled with $(n, ((2, i_0)), a)$, such that $n - 1 = 2^{i_0}$, for some $i_0 \geq 1$ and some $a \in \{2, \ldots, n - 1\}$.

(2) If $L_1, \ldots, L_k$ are k pre-Lucas (with $k \geq 1$), where the tree L_j is a pre-Lucas tree for some odd integer $q_j \geq 3$, then the tree L whose root is labeled with $(n, ((2, i_0), ((q_1, i_1), \ldots, (q_k, i_k)), a)$ and whose jth subtree is L_j is a *pre-Lucas tree* for n if

$$n - 1 = 2^{i_0} q_1^{i_1} \cdots q_k^{i_k},$$

for some $i_0, i_1, \ldots, i_k \geq 1$, and some $a \in \{2, \ldots, n - 1\}$.

Both in (1) and (2), the number a is a guess for a primitive root modulo n.

A pre-Lucas tree for n is a *Lucas tree for* n if the following conditions are satisfied:

(3) If L is a one-node tree labeled with $(n, ((2, i_0)), a)$, then

$$a^{(n-1)/2} \equiv -1 \pmod{n}.$$

(4) If L is a pre-Lucas tree whose root is labeled with $(n, ((2, i_0), ((q_1, i_1), \ldots, (q_k, i_k)), a)$, and whose jth subtree L_j is a pre-Lucas tree for q_j, then L_j is a Lucas tree for q_j for $j = 1, \ldots, k$, and

(a) $a^{(n-1)/2} \equiv -1 \pmod{n}$.

(b) $a^{(n-1)/2q_j} \not\equiv -1 \pmod{n}$ for $j = 1, \ldots, k$.

Since Conditions (3) and (4) of Definition 11.12 are Conditions (1b) and (2b) of Theorem, 11.10, we see that Definition 11.12 has been designed in such a way that Theorem 11.10 yields the following result.

Theorem 11.11. *An odd integer $n \geq 3$ is prime iff it has some Lucas tree.*

The issue is now to see how long it takes to check that a pre-Lucas tree is a Lucas tree. For this, we need a method for computing $x^n \bmod n$ in polynomial time in $\log_2 n$. This is the object of the next section.

11.6 Algorithms for Computing Powers Modulo m

Let us first consider computing the nth power x^n of some positive integer. The idea is to look at the parity of n and to proceed recursively. If n is even, say $n = 2k$, then

$$x^n = x^{2k} = (x^k)^2,$$

so, compute x^k recursively and then square the result. If n is odd, say $n = 2k + 1$, then

$$x^n = x^{2k+1} = (x^k)^2 \cdot x,$$

so, compute x^k recursively, square it, and multiply the result by x.

What this suggests is to write $n \geq 1$ in binary, say

$$n = b_\ell \cdot 2^\ell + b_{\ell-1} \cdot 2^{\ell-1} + \cdots + b_1 \cdot 2^1 + b_0,$$

where $b_i \in \{0, 1\}$ with $b_\ell = 1$ or, if we let $J = \{j \mid b_j = 1\}$, as

$$n = \sum_{j \in J} 2^j.$$

Then we have

$$x^n \equiv x^{\sum_{j \in J} 2^j} = \prod_{j \in J} x^{2^j} \bmod m.$$

This suggests computing the residues r_j such that

$$x^{2^j} \equiv r_j \pmod{m},$$

because then,

$$x^n \equiv \prod_{j \in J} r_j \pmod{m},$$

where we can compute this latter product modulo m two terms at a time.

For example, let us compute $2^{90} \bmod 91$ (recall that $91 = 7 \cdot 13$). We have

$$91 = 2^6 + 2^4 + 2^3 + 2^1,$$

and we compute the following powers modulo 91:

$$\begin{aligned}
2^{2^1} &\equiv 4 \pmod{91} \\
2^{2^2} &\equiv 16 \pmod{91} \\
2^{2^3} &\equiv 16^2 \equiv 74 \pmod{91} \\
2^{2^4} &\equiv 74^2 \equiv 16 \pmod{91} \\
2^{2^5} &\equiv 16^2 \equiv 74 \pmod{91} \\
2^{2^6} &\equiv 74^2 \equiv 16 \pmod{91}.
\end{aligned}$$

Consequently,

$$\begin{aligned}2^{90} &\equiv 4 \cdot 74 \cdot 16 \cdot 16 \pmod{91}\\ &\equiv 23 \cdot 16 \cdot 16 \pmod{91}\\ &\equiv 4 \cdot 16 = 64 \pmod{91},\end{aligned}$$

so

$$2^{90} \equiv 64 \pmod{91},$$

confirming what we claimed in Section 11.2.

As a second example, let us compute $3^{90} \bmod 91$. We have

$$91 = 2^6 + 2^4 + 2^3 + 2^1,$$

and we compute the following powers modulo 91:

$$\begin{aligned}3^{2^1} &\equiv 9 \pmod{91}\\ 3^{2^2} &\equiv 81 \pmod{91}\\ 3^{2^3} &\equiv 81^2 \equiv 9 \pmod{91}\\ 3^{2^4} &\equiv 81 \pmod{91}\\ 3^{2^5} &\equiv 9 \pmod{91}\\ 3^{2^6} &\equiv 81 \pmod{91}.\end{aligned}$$

Consequently,

$$\begin{aligned}3^{90} &\equiv 9 \cdot 9 \cdot 81 \cdot 81 \pmod{91}\\ &\equiv 81 \cdot 81 \cdot 81 \pmod{91}\\ &\equiv 9 \cdot 81 \pmod{91}\\ &\equiv 1 \pmod{91},\end{aligned}$$

so

$$3^{90} \equiv 1 \pmod{91},$$

confirming what we claimed in Section 11.2.

As a third example, say we want to compute $999^{179} \bmod 1763$. First, we observe that

$$179 = 2^7 + 2^5 + 2^4 + 2^1 + 1,$$

and we compute the following powers modulo 1763:

$$\begin{aligned}
999^{2^1} &\equiv 143 \pmod{1763}\\
999^{2^2} &\equiv 143^2 \equiv 1056 \pmod{1763}\\
999^{2^3} &\equiv 1056^2 \equiv 920 \pmod{1763}\\
999^{2^4} &\equiv 920^2 \equiv 160 \pmod{1763}\\
999^{2^5} &\equiv 160^2 \equiv 918 \pmod{1763}\\
999^{2^6} &\equiv 918^2 \equiv 10 \pmod{1763}\\
999^{2^7} &\equiv 10^2 \equiv 100 \pmod{1763}.
\end{aligned}$$

Consequently,

$$\begin{aligned}
999^{179} &\equiv 999 \cdot 143 \cdot 160 \cdot 918 \cdot 100 \pmod{1763}\\
&\equiv 54 \cdot 160 \cdot 918 \cdot 100 \pmod{1763}\\
&\equiv 1588 \cdot 918 \cdot 100 \pmod{1763}\\
&\equiv 1546 \cdot 100 \pmod{1763}\\
&\equiv 1219 \pmod{1763},
\end{aligned}$$

and we find that

$$999^{179} \equiv 1219 \pmod{1763}.$$

Of course, it would be impossible to exponentiate 999^{179} first and then reduce modulo 1763. As we can see, the number of multiplications needed is bounded by $2\log_2 n$, which is quite good.

The above method can be implemented without actually converting n to base 2. If n is even, say $n = 2k$, then $n/2 = k$, and if n is odd, say $n = 2k + 1$, then $(n-1)/2 = k$, so we have a way of dropping the unit digit in the binary expansion of n and shifting the remaining digits one place to the right without explicitly computing this binary expansion. Here is an algorithm for computing $x^n \bmod m$, with $n \geq 1$, using the *repeated squaring* method.

An Algorithm to Compute $x^n \bmod m$ Using Repeated Squaring

```
begin
  $u := 1$; $a := x$;
  while $n > 1$ do
    if even($n$) then $e := 0$ else $e := 1$;
    if $e = 1$ then $u := a \cdot u \bmod m$;
    $a := a^2 \bmod m$; $n := (n - e)/2$
  endwhile;
  $u := a \cdot u \bmod m$
end
```

The final value of u is the result. The reason why the algorithm is correct is that after j rounds through the while loop, $a = x^{2^j} \bmod m$ and

$$u = \prod_{i \in J \mid i < j} x^{2^i} \bmod m,$$

with this product interpreted as 1 when $j = 0$.

Observe that the while loop is only executed $n - 1$ times to avoid squaring once more unnecessarily and the last multiplication $a \cdot u$ is performed outside of the while loop. Also, if we delete the reductions modulo m, the above algorithm is a fast method for computing the nth power of an integer x and the time speed-up of not performing the last squaring step is more significant. We leave the details of the proof that the above algorithm is correct as an exercise.

11.7 PRIMES is in $\mathcal{NP}$

Exponentiation modulo n can performed by repeated squaring, as explained in Section 11.6. In that section, we observed that computing $x^m \bmod n$ requires at most $2\log_2 m$ modular multiplications. Using this fact, we obtain the following result adapted from Crandall and Pomerance [5].

Proposition 11.12. *If p is any odd prime, then any pre-Lucas tree L for p has at most $\log_2 p$ nodes, and the number $M(p)$ of modular multiplications required to check that the pre-Lucas tree L is a Lucas tree is less than $2\log_2^2 p$.*

Proof. Let $N(p)$ be the number of nodes in a pre-Lucas tree for p. We proceed by complete induction. If $p = 3$, then $p - 1 = 2^1$, any pre-Lucas tree has a single node, and $1 < \log_2 3$.

Suppose the results holds for any odd prime less than p. If $p - 1 = 2^{i_0}$, then any Lucas tree has a single node, and $1 < \log_2 3 < \log_2 p$. If $p - 1$ has the prime factorization

$$p - 1 = 2^{i_0} q_1^{i_1} \cdots q_k^{i_k},$$

then by the induction hypothesis, each pre-Lucas tree L_j for q_j has less than $\log_2 q_j$ nodes, so

$$N(p) = 1 + \sum_{j=1}^{k} N(q_j) < 1 + \sum_{j=1}^{k} \log_2 q_j = 1 + \log_2(q_1 \cdots q_k) \leq 1 + \log_2\left(\frac{p-1}{2}\right) < \log_2 p,$$

establishing the induction hypothesis.

If r is one of the odd primes in the pre-Lucas tree for p, and $r < p$, then there is some other odd prime q in this pre-Lucas tree such that r divides $q - 1$ and $q \leq p$. We also have to show that at some point, $a^{(q-1)/2r} \not\equiv -1 \pmod{q}$ for some a, and at another point, that $b^{(r-1)/2} \equiv -1 \pmod{r}$ for some b. Using the fact that the number of modular multiplications required to exponentiate to the power m is at most $2\log_2 m$, we see that the number of multiplications required by the above two exponentiations does not exceed

$$2\log_2\left(\frac{q-1}{2r}\right) + 2\log_2\left(\frac{r-1}{2}\right) = 2\log_2\left(\frac{(q-1)(r-1)}{4r}\right) < 2\log_2 q - 4 < 2\log_2 p.$$

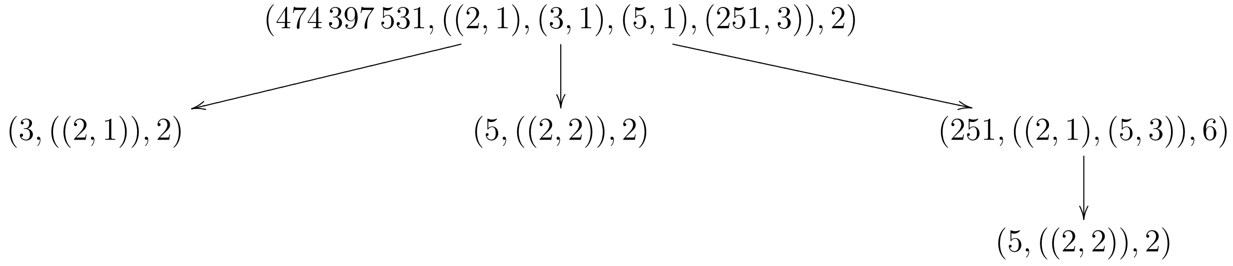
As a consequence, we have

$$M(p) < 2\log_2\left(\frac{p-1}{2}\right) + (N(p)-1)2\log_2 p < 2\log_2 p + (\log_2 p - 1)2\log_2 p = 2\log_2^2 p,$$

as claimed. □

The following impressive example is from Pratt [34].

Example 11.2. Let $n = 474\,397\,531$. It is easy to check that $n - 1 = 474\,397\,531 - 1 = 474\,397\,530 = 2 \cdot 3 \cdot 5 \cdot 251^3$. We claim that the following is a Lucas tree for $n = 474\,397\,531$:



To verify that the above pre-Lucas tree is a Lucas tree, we check that 2 is indeed a primitive root modulo 474 397 531 by computing (using `Mathematica`) that

$$\begin{aligned}
2^{474\,397\,530/2} &\equiv 2^{237\,198\,765} \equiv -1 \pmod{474\,397\,531} &(1)\\
2^{474\,397\,530/(2\cdot 3)} &\equiv 2^{79\,066\,255} \equiv 9\,583\,569 \pmod{474\,397\,531} &(2)\\
2^{474\,397\,530/(2\cdot 5)} &\equiv 2^{47\,439\,753} \equiv 91\,151\,207 \pmod{474\,397\,531} &(3)\\
2^{474\,397\,530/(2\cdot 251)} &\equiv 2^{945\,015} \equiv 282\,211\,150 \pmod{474\,397\,531}. &(4)
\end{aligned}$$

The number of modular multiplications is: 27 in (1), 26 in (2), 25 in (3) and 19 in (4).

We have $251 - 1 = 250 = 2 \cdot 5^3$, and we verify that 6 is a primitive root modulo 251 by computing:

$$\begin{aligned}
6^{250/2} &\equiv 6^{125} \equiv -1 \pmod{251} &(5)\\
6^{250/(2\cdot 5)} &\equiv 6^{10} \equiv 175 \pmod{251}. &(6)
\end{aligned}$$

The number of modular multiplications is: 6 in (5), and 3 in (6).

We have $5 - 1 = 4 = 2^2$, and 2 is a primitive root modulo 5, since

$$2^{4/2} \equiv 2^2 \equiv -1 \pmod{5}. \tag{7}$$

This takes one multiplication.

We have $3 - 1 = 2 = 2^1$, and 2 is a primitive root modulo 3, since

$$2^{2/2} \equiv 2^1 \equiv -1 \pmod{3}. \tag{8}$$

This takes 0 multiplications.

Therefore, 474 397 531 is prime.

As nice as it is, Proposition 11.12 is deceiving, because *finding* a Lucas tree is hard.

Remark: Pratt [34] presents his method for finding a certificate of primality in terms of a proof system. Although quite elegant, we feel that this method is not as transparent as the method using Lucas trees, which we adapted from Crandall and Pomerance [5]. Pratt's proofs can be represented as trees, as Pratt sketches in Section 3 of his paper. However, Pratt uses the basic version of Lucas' theorem, Theorem 11.9, instead of the improved version, Theorem 11.10, so his proof trees have at least twice as many nodes as ours.

As nice as it is, Proposition 11.12 is deceiving, because *finding* a Lucas tree is hard.

The following nice result was first shown by V. Pratt in 1975 [34].

Theorem 11.13. *The problem* PRIMES *(testing whether an integer is prime) is in* $\mathcal{NP}$*.*

Proof. Since all even integers besides 2 are composite, we can restrict out attention to odd integers $n \geq 3$. By Theorem 11.11, an odd integer $n \geq 3$ is prime iff it has a Lucas tree. Given any odd integer $n \geq 3$, since all the numbers involved in the definition of a pre-Lucas tree are less than n, there is a finite (very large) number of pre-Lucas trees for n. Given a guess of a Lucas tree for n, checking that this tree is a pre-Lucas tree can be performed in $O(\log_2 n)$, and by Proposition 11.12, checking that it is a Lucas tree can be done in $O(\log_2^2 n)$. Therefore PRIMES is in $\mathcal{NP}$. □

Of course, checking whether a number n is composite is in $\mathcal{NP}$, since it suffices to guess to factors n_1, n_2 and to check that $n = n_1 n_2$, which can be done in polynomial time in $\log_2 n$. Therefore, PRIMES $\in \mathcal{NP} \cap \text{co}\mathcal{NP}$. As we said earlier, this was the situation until the discovery of the AKS algorithm, which places PRIMES in $\mathcal{P}$.

Remark: Altough finding a primitive root modulo p is hard, we know that the number of primitive roots modulo p is $\varphi(\varphi(p))$. If p is large enough, this number is actually quite large. According to Crandal and Pomerance [5] (Chapter 4, Section 4.1.1), if p is a prime and if $p > 200560490131$, then p has more than $p/(2 \ln \ln p)$ primitive roots.

Chapter 12

Polynomial- Space Complexity; $\mathcal{PS}$ and $\mathcal{NPS}$

12.1 The Classes $\mathcal{PS}$ (or PSPACE) and $\mathcal{NPS}$ (NPSPACE)

In this chapter we consider complexity classes based on restricting the amount of *space* used by the Turing machine rather than the amount of time.

Definition 12.1. A deterministic or nondeterminitic Turing machine M is *polynomial-space bounded* if there is a polynomial $p(X)$ such that for every input $x \in \Sigma^*$, no matter how much time it uses, the machine M never visits more than $p(|x|)$ tape cells (symbols). Equivalently, for every ID $upav$ arising during the computation, we have $|uav| \leq p(|x|)$.

The class of languages $L \subseteq \Sigma^*$ accepted by some *deterministic* polynomial-space bounded Turing machine is denoted by $\mathcal{PS}$ or PSPACE. Similarly, the class of languages $L \subseteq \Sigma^*$ accepted by some *nondeterministic* polynomial-space bounded Turing machine is denoted by $\mathcal{NPS}$ or NPSPACE.

Obviously $\mathcal{PS} \subseteq \mathcal{NPS}$. Since a (time) polynomially bounded Turing machine can't visit more tape cells (symbols) than one plus the number of moves it makes, we have

$$\mathcal{P} \subseteq \mathcal{PS} \quad \text{and} \quad \mathcal{NP} \subseteq \mathcal{NPS}.$$

Nobody knows whether these inclusions are strict, but these are the most likely assumptions. Unlike the situation for time-bounded Turing machines where the big open problem is whether $\mathcal{P} \neq \mathcal{NP}$, for *time-bounded* Turing machines, we have

$$\mathcal{PS} = \mathcal{NPS}.$$

Walter Savitch proved this result in 1970 (and it is known as *Savitch's theorem*).

Now Definition 12.1 does not say anything about the time-complexity of the Turing machine, so such a machine could even run forever. However, the number of ID's that a polynomial-space bounded Turing machine can visit started on input x is a function of $|x|$ of the form $sp(|x|)t^{p(|x|)}$ for some constants $s > 0$ and $t > 0$, so by the pigeonhole principle, it the number of moves is larger than a certain constant ($c^{1+p(|x|)}$ with $c = s+t$), then some ID must repeat. This fact can be used to show that there is a shorter computation accepting x of length at most $c^{1+p(|x|)}$.

Proposition 12.1. *For any deterministic or nondeterministic polynomial-space bouned Turing machine M with polynomal space bound $p(X)$, there is a constant $c > 1$ such that for every input $x \in \Sigma^*$, if M accepts x, then M accepts x in at most $c^{1+p(|x|)}$ steps.*

Proof. Suppose there are t symbols in the tape alphabet and s states. Then the number of distinct ID's when only $p(|x|)$ tape cells are used is at most $sp(|x|)t^{p(|x|)}$, because we can choose one of s states, place the reading head in any of $p(|x|)$ distinct positions, and there are $t^{p(|x|)}$ strings of tape symbols of length $p(|x|)$. If we let $c = s+t$, by the binomial formula we have

$$\begin{aligned} c^{1+p(|x|)} = (s+t)^{1+p(|x|)} &= \sum_{k=0}^{1+p(|x|)} \binom{1+p(|x|)}{k} s^k t^{1+p(|x|)-k)} \\ &= t^{1+p(|x|)} + (1+p(|x|))st^{p(|x|)} + \cdots \end{aligned}$$

Obviously $(1+p(|x|))st^{p(|x|)} > sp(|x|)t^{p(|x|)}$, so if the number of ID's in the computation is greater than $c^{1+p(|x|)}$, by the pigeonhole principle, two ID's must be identical. By considering a shortest accepting sequence of ID's with n steps, we deduce that $n \leq c^{1+p(|x|)}$, since otherwise the preceding argument shows that the computation would be of the form

$$ID_0 \vdash^* \cdots \vdash^* ID_h \vdash^+ ID_k \vdash^* ID_n$$

with $ID_h = ID_k$, so we would have an even shorter computation

$$ID_0 \vdash^* \cdots \vdash^* ID_h \vdash^* ID_n,$$

contradicting the minimality of the original computation. □

Proposition 12.1 implies that languages in $\mathcal{NPS}$ are computable (in fact, primitive recursive, and even in $\mathcal{EXP}$). This still does not show that languages in $\mathcal{NPS}$ are accepted by polynomial-space Turing machines that *always halt* within some time $c^{q(|x|)}$ for some polynomial $q(X)$. Such a result can be shown using a simulation involving a Turing machine with two tapes.

Proposition 12.2. *For any language $L \in \mathcal{PS}$ (resp. $L \in \mathcal{NPS}$), there is deterministic (resp. nondeterministic) polynomial-space bounded Turing machine M, a polynomal $q(X)$ and a constant $c > 1$, such that for every input $x \in \Sigma^*$, M accepts x in at most $c^{q(|x|)}$ steps.*

A proof of Proposition 12.2 can be found in Hopcroft, Motwani and Ullman [23] (Section 11.2.2, Theorem 11.4).

We now turn to Savitch's theorem.

12.2 Savitch's Theorem: $\mathcal{PS} = \mathcal{NPS}$

The key to the fact that $\mathcal{PS} = \mathcal{NPS}$ is that given a polynomial-space bounded nondeterministic Turing machine M, there is a recursive method to check whether $I \vdash^k J$ with $0 \leq k \leq m$ using at *most* $\log_2 m$ recursive calls, for any two ID's I and J and any natural number $m \geq 1$, (that is, whether there is some computation of $k \leq m$ steps from I to J).

The idea is reminiscent of binary search, namely, to recursively find some intermediate ID K such that $I \vdash^{m_1} K$ and $K \vdash^{m_2} J$ with $m_1 \leq m/2$ and $m_2 \leq m/2$ (here $m/2$ is the integer quotient obtained by dividing m by 2). Because the Turing machine M is polynomial-space bounded, for a given input x, we know from Proposition 12.1 that there are at most $c^{1+p(|x|)}$ distinct ID's, so the search is finite. We will intitially set $m = c^{1+p(|x|)}$, so at most $\log_2 c^{1+p(|x|)} = O(p(|x|)$ recursive calls will be made. We will show that each stack frame takes $O(p(|x|)$ space, so altogether the search uses $O(p(|x|)^2)$ amount of space. This is the crux of Savitch's argument.

The recursive procedure that deals with stack frames of the form $[I, J, m]$ is shown below.

```
function reach(I, J, m): boolean
begin
  if $m = 1$ then
    if $I = J = K$ or $I \vdash^1 J$ then
         reach = true
    else
         reach = false
    endif
  else
    for each possible ID $K$ do
      if reach$(I, K, m/2)$ and reach$(K, J, m/2)$ then
        reach = true
      else
        reach = false
      endif
    endfor
  endif
end
```

Even though the above procedure makes two recursive calls, they are performed sequentially, so the maximum number of stack frames that may arise corresponds to the sequence

$$[I_1, J_1, m],\ [I_2, J_2, m/2],\ [I_3, J_3, m/4],\ [I_4, J_4, m/8],\ \cdots, [I_k, J_k, m/2^{k-1}], \cdots$$

which has length at most $\log_2 m$. Using the procedure *search*, we obtain Savitch's theorem.

Theorem 12.3. *(Savitch, 1970) The complexity classes $\mathcal{PS}$ and $\mathcal{NPS}$ are identical. In fact, if L is accepted by the polynomial-space bounded nondeterministic Turing machine M with space bound $p(X)$, then there is a polynomial-space bounded deterministic Turing machine D accepting L with space bound $O(p(X)^2)$.*

Sketch of proof. Assume that L is accepted by the polynomial-space bounded nondeterministic Turing machine M with space bound $p(X)$. By Proposition 12.1 we may assume that M accepts any input $x \in L$ in at most $c^{1+p(|x|)}$ steps (for some $c > 1$). Set $m = c^{1+p(|x|)}$.

We can design a deterministic Turing machine D which determines (using the function *search*) whether $I_0 \vdash^k J$ with $k \leq m$ where $I_0 = q_0x$ is the starting ID, for all accepting ID's J, by enumerate all accepting ID's J using at most $p(|x|)$ tape cells, using a scratch tape.

As we explained above, the function *search* makes no more than $\log_2 c^{1+p(|x|)} = O(p(|x|)$ recursive calls, Each stack frame takes $O(p(|x|)$ space. The reason is that every ID has at most $1 + p(|x|)$ tape cells and that if we write $m = c^{1+p(|x|)}$ in binary, this takes $\log_2 m = O(p(|x|)$ tape cells. Since at most $O(p(|x|)$ stack frames may arise and since each stack frame has size at most $O(p(|x|)$, the deterministic TM D uses at most $O(p(|x|)^2$ space. For more details, see Hopcroft, Motwani and Ullman [23] (Section 11.2.3, Theorem 11.5). □

Savitch's theorem and Proposition 12.1 show that $\mathcal{PS} = \mathcal{NPS} \subseteq \mathcal{EXP}$. Whether this inclusion is strict is an open problem. The present status of the relative containments of the complexity classes that we have discussed so far is illustrated in Figure 12.1

Savitch's theorem shows that nondeterminism does not help as far as polynomial *space* is concerned, but we still don't have a good example of a language in $\mathcal{PS} = \mathcal{NPS}$ which is not known to be in $\mathcal{NP}$. The next section is devoted to such a problem. This problem also turns out to be $\mathcal{PS}$-complete, so we discuss this notion as well.

12.3 A Complete Problem for $\mathcal{PS}$: QBF

Logic is a natural source of problems complete with respect to a number of complexity classes: SAT is $\mathcal{NP}$-complete (see Theorem 9.8), TAUT is co$\mathcal{NP}$-complete (see Proposition 10.3). It turns out that the validity problem for quantified boolean formulae is $\mathcal{PS}$-complete. We will describe this problem shortly, but first we define $\mathcal{PS}$-completeness.

Definition 12.2. A language $L \subseteq \Sigma^*$ is $\mathcal{PS}$-complete if:

(1) $L \in \mathcal{PS}$.

(2) For every language $L_2 \in \mathcal{PS}$, there is a *polynomial-time computable* function $f\colon \Sigma^* \to \Sigma^*$ such that $x \in L_2$ iff $f(x) \in L$, for all $x \in \Sigma^*$.

Observe that we require the reduction function f to be *polynomial-time computable* rather than *polynomial-space computable*. The reason for this is that with this stronger form of reduction we can prove the following proposition whose simple proof is left as an exercise.

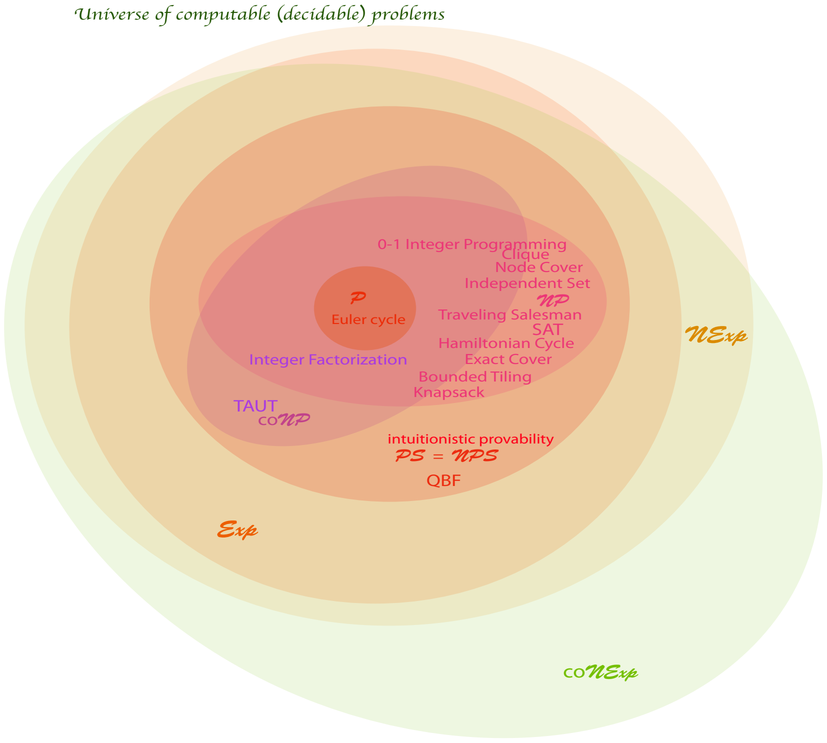


Figure 12.1: Relative containments of the complexity classes.

Proposition 12.4. *Suppose L is a $\mathcal{PS}$-complete language. Then the following facts hold:*

(1) If $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{PS}$.

(2) If $L \in \mathcal{NP}$, then $\mathcal{NP} = \mathcal{PS}$.

The premises in Proposition 12.4 are very unlikely, but we never know!

We now define the class of quantified boolean formulae. These are actually second-order formulae because we are allowed to quantify over propositional variables, which are 0-ary (constant) predicate symbols. As we will see, validity is still decidable, but the fact that we allow alternation of the quantifiers $\forall$ and $\exists$ makes the problem harder, in the sense that testing validity or nonvalidity no longer appears to be doable in $\mathcal{NP}$ (so far, nobody knows how to do this!).

Recall from Section 9.5 that we have a countable set **PV** of *propositional (or boolean)* *variables*,

$$\mathbf{PV} = \{x_1, x_2, \ldots, \}.$$

Definition 12.3. A *quantified boolean formula* (for short *QBF*) is an expression A defined inductively as follows:

(1) The constants $\top$ and $\perp$ and every propositional variable x_i are QBF's called *atomic* QBF's.

(2) If B is a QBF, then $\neg B$ is a QBF.

(3) If B and C are QBF's, then $(B \vee C)$ is a QBF.

(4) If B and C are QBF's, then $(B \wedge C)$ is a QBF.

(5) If B is a QBF and if x is a propositional variable, then $\forall x B$ is a QBF. The variable x is said to be *universally bound by* $\forall$.

(6) If B is a QBF and if x is a propositional variable, then $\exists x B$ is a QBF. The variable x is said to be *existentially bound by* $\exists$.

(7) If allow the connective $\Rightarrow$, and if B and C are QBF's, then $(B \Rightarrow C)$ is a QBF.

Example 12.1. The following formula is a QBF:

$$A = \forall x\big(\exists y(x \wedge y) \vee \forall z(\neg x \vee z)\big).$$

As usual, we can define inductively the notion of free and bound variable as follows.

Definition 12.4. Given any QBF A, we define the set $FV(A)$ of *variables free in* A and the set $BV(A)$ of *variables bound in* A as follows:

$$\begin{aligned}
FV(\perp) &= FV(\top) = \emptyset\\
FV(x_i) &= \{x_i\}\\
FV(\neg B) &= FV(B)\\
FV((B * C)) &= FV(B) \cup FV(C), \qquad * \in \{\vee, \wedge, \Rightarrow\}\\
FV(\forall x B) &= FV(B) - \{x\}\\
FV(\exists x B) &= FV(B) - \{x\},
\end{aligned}$$

and

$$\begin{aligned}
BV(\perp) &= BV(\top) = \emptyset\\
BV(x_i) &= \emptyset\\
BV(\neg B) &= BV(B)\\
BV((B * C)) &= BV(B) \cup BV(C), \qquad * \in \{\vee, \wedge, \Rightarrow\}\\
BV(\forall x B) &= BV(B) \cup \{x\}\\
BV(\exists x B) &= BV(B) \cup \{x\}.
\end{aligned}$$

A QBF A such that $FV(A) = \emptyset$ (A has no free variables) is said to be *closed* or a *sentence*.

It should be noted that $FV(A)$ and $BV(A)$ may not be disjoint! For example, if

$$A = x_1 \vee \forall x_1(\neg x_1 \vee x_2),$$

then $FV(A) = \{x_1, x_2\}$ and $BV(A) = \{x_1\}$. This situation is somewhat undesirable. Intuitively, A is "equivalent" to the QBF

$$A' = x_1 \vee \forall x_3(\neg x_3 \vee x_2),$$

with $FV(A') = \{x_1, x_2\}$ and $BV(A') = \{x_3\}$. Here equivalent means that A and A' have the same truth value for all truth assignments. To make all this precise we proceed as follows.

Definition 12.5. A *substitution* is a set of pairs $\varphi = \{(y_1, A_1), \ldots, (y_m, A_m)\}$ where the variables $y_1, \ldots, y_m$ are *distinct* and $A_1, \ldots, A_m$ are arbitrary QBF's. We write $\varphi = [y_1 := A_1, \ldots, y_m := A_m]$. For any QBF B, we also denote by $\varphi[y_i := B]$ the substitution such that $y_i := A_i$ is replaced by $y_i := B$. In particular, $\varphi[y_i := y_i]$ leaves y_i unchanged.

Given a QBF A, the result of applying the substitution $\varphi = [y_1 := A_1, \ldots, y_m := A_m]$ to A, denoted $A[\varphi]$, is defined inductively as follows:

$$\begin{aligned}
\perp[\varphi] &= \perp \\
\top[\varphi] &= \top \\
x[\varphi] &= A_i \quad \text{if } x = y_i,\ 1 \leq i \leq m \\
x[\varphi] &= x \quad \text{if } x \notin \{y_1, \ldots, y_m\} \\
(\neg B)[\varphi] &= (\neg B)[\varphi] \\
(B * C)[\varphi] &= (B[\varphi] * C[\varphi]), \qquad * \in \{\vee, \wedge, \Rightarrow\} \\
(\forall x B)[\varphi] &= \forall x B[\varphi[y_i := y_i]] \quad \text{if } x = y_i,\ 1 \leq i \leq m \\
(\forall x B)[\varphi] &= \forall x B[\varphi] \quad \text{if } x \notin \{y_1, \ldots, y_m\} \\
(\exists x B)[\varphi] &= \exists x B[\varphi[y_i := y_i]] \quad \text{if } x = y_i,\ 1 \leq i \leq m \\
(\exists x B)[\varphi] &= \exists x B[\varphi] \quad \text{if } x \notin \{y_1, \ldots, y_m\}.
\end{aligned}$$

Definition 12.6. A QBF A is *rectified* if distinct quantifiers bind distinct variables and if $BV(A) \cap FV(A) = \emptyset$.

Given a QBF A and any finite set V of variables, we can define recursively a new rectified QBF A' such that $BV(A') \cap V = \emptyset$.

(1) If $A = \top$, or $A = \perp$, or $A = x_i$, then $A' = A$.

(2) If $A = \neg B$, then $A' = A$.

(3) If $A = (B \vee C)$, then first we find recursively some rectified QBF B_1 such that $BV(B_1) \cap V = \emptyset$, then we find recursively some rectified QBF C_1 such that $BV(C_1) \cap (FV(B_1) \cup BV(B_1) \cup V) = \emptyset$, and we set $A' = (B_1 \vee C_1)$. We proceed similarly if $A = (B \wedge C)$ or $A = (B \Rightarrow C)$, with $\vee$ replaced by $\wedge$ or $\Rightarrow$.

(4) If $A = \forall x B$, first we find recursively some rectified QBF B_1 such that $BV(B_1) \cap V = \emptyset$, and then we let $A' = \forall z B_1[x := z]$ for some new variable z such that $z \notin FV(B_1) \cup BV(B_1) \cup V$. Note that in this step it is possible that $x \notin FV(B)$.

(5) If $A = \exists x B$, first we find recursively some rectified QBF B_1 such that $BV(B_1) \cap V = \emptyset$, and then we let $A' = \exists z B_1[x := z]$ for some new variable z such that $z \notin FV(B_1) \cup BV(B_1) \cup V$. Note that in this step it is possible that $x \notin FV(B)$.

Given any QBF A, we find a rectified QBF A' by applying the above procedure recursively starting with A and $V = \emptyset$.

Recall that a *truth assignment* or *valuation* is a function $v\colon \mathbf{PV} \to \{\mathbf{T}, \mathbf{F}\}$. We also let $\overline{\mathbf{T}} = \mathbf{F}$ and $\overline{\mathbf{T}} = \mathbf{T}$.

Definition 12.7. Given a valuation $v\colon \mathbf{PV} \to \{\mathbf{T}, \mathbf{F}\}$, we define *truth value* $A[v]$ of a QBF A inductively as follows.

$$\begin{aligned}
\bot\,[v] &= \mathbf{F} && (1)\\
\top[v] &= \mathbf{T} && (2)\\
x[v] &= v(x) && (3)\\
(\neg B)[v] &= \overline{B[v]} = \mathbf{F} \text{ if } B[v] = \mathbf{T} \text{ else } \mathbf{T} \text{ if } B[v] = \mathbf{F} && (4)\\
(B \vee C)[v] &= B[v] \;\mathbf{or}\; C[v] && (5)\\
(B \wedge C)[v] &= B[v] \;\mathbf{and}\; C[v] && (6)\\
(B \Rightarrow C)[v] &= \overline{B[v]} \;\mathbf{or}\; C[v] && (7)\\
(\forall x B)[v] &= B[v[x := \mathbf{T}]] \;\mathbf{and}\; B[v[x := \mathbf{F}]] && (8)\\
(\exists x B)[v] &= B[v[x := \mathbf{T}]] \;\mathbf{or}\; B[v[x := \mathbf{F}]]. && (9)
\end{aligned}$$

If $A[v] = \mathbf{T}$, we write say that v *satisfies* A and we write $v \models A$. If $A[v] = \mathbf{T}$ for *all valuations* v , we say that A *is valid* and we write $\models A$.

As usual, we write $A \equiv B$ iff $(A \Rightarrow B) \wedge (B \Rightarrow A)$ is valid.

In Clause (5) when evaluating $(B \vee C)[v]$, if $B[v] = \mathbf{T}$, then we don't need to evaluate $C[v]$, since $\mathbf{T}\,\mathbf{or}\,b = \mathbf{T}$ independently of $b \in \{\mathbf{T}, \mathbf{F}\}$, and so $(B \vee C)[v] = \mathbf{T}$. If $B[v] = \mathbf{F}$, then we need to evaluate $C[v]$, and $(B \vee C)[v] = \mathbf{T}$ iff $C[v] = \mathbf{T}$. Even though the above method is more economical, we usually evaluate both $B[v]$ and $C[v]$ and then compute $B[v]$ **or** $C[v]$.

A similar discussion applies to evaluating $(\exists x B)[v]$ in Clause (9). If $B[v[x := \mathbf{T}]] = \mathbf{T}$, then we don't need to evaluate $B[v[x := \mathbf{F}]]$ and $(\exists x B)[v] = \mathbf{T}$. If $B[v[x := \mathbf{T}]] = \mathbf{F}$, then we need to evaluate $B[v[x := \mathbf{F}]]$, and $(\exists x B)[v] = \mathbf{T}$ iff $B[v[x := \mathbf{F}]] = \mathbf{T}$. Even though the above method is more economical, we usually evaluate both $B[v[x := \mathbf{T}]]$ and $B[v[x := \mathbf{F}]]$ and then compute $B[v[x := \mathbf{T}]]$ **or** $B[v[x := \mathbf{F}]]$.

Example 12.2. Let us show that the QBF

$$A = \forall x\big(\exists y(x \wedge y) \vee \forall z(\neg x \vee z)\big)$$

from Example 12.1 is valid. This is a closed formula so v is irrelevant. By Clause (8) of Definition 12.7, we need to evaluate $A[x := \mathbf{T}]$ and $A[x := \mathbf{F}]$.

To evaluate $A[x := \mathbf{T}]$, by Clause (5) of Definition 12.7, we need to evaluate $(\exists y(x \wedge y))[x := \mathbf{T}]$ and $(\forall z(\neg x \vee z))[x := \mathbf{T}]$.

To evaluate $(\exists y(x \wedge y))[x := \mathbf{T}]$, by Clause (9) of Definition 12.7, we need to evaluate $(x \wedge y)[x := \mathbf{T}, y := \mathbf{T}]$ and $(x \wedge y)[x := \mathbf{T}, y := \mathbf{F}]$.

We have (by Clause (6)) $(x \wedge y)[x := \mathbf{T}, y := \mathbf{T}] = \mathbf{T}\,\mathbf{and}\,\mathbf{T} = \mathbf{T}$ and $(x \wedge y)[x := \mathbf{T}, y := \mathbf{F}] = \mathbf{T}\ \mathbf{and}\ \mathbf{F} = \mathbf{F}$, so

$$(\exists y(x \wedge y))[x := \mathbf{T}] = (x \wedge y)[x := \mathbf{T}, y := \mathbf{T}]\ \mathbf{or}\ (x \wedge y)[x := \mathbf{T}, y := \mathbf{F}] = \mathbf{T}\ \mathbf{or}\ \mathbf{F} = \mathbf{T}. \quad (1)$$

To evaluate $(\forall z(\neg x \vee z))[x := \mathbf{T}]$, by Clause (8) of Definition 12.7, we need to evaluate $(\neg x \vee z)[x := \mathbf{T}, z := \mathbf{T}]$ and $(\neg x \vee z)[x := \mathbf{T}, z := \mathbf{F}]$.

Using Clauses (4) and (5) of Definition 12.7, we have $(\neg x \vee z)[x := \mathbf{T}, z := \mathbf{T}] = \overline{\mathbf{T}}\ \mathbf{land}\ \mathbf{T} = \mathbf{T}$ and $(\neg x \vee z)[x := \mathbf{T}, z := \mathbf{F}] = \overline{\mathbf{T}}\ \mathbf{land}\ \mathbf{F} = \mathbf{F}$, so

$$(\forall z(\neg x \vee z))[x := \mathbf{T}] = (\neg x \vee z)[x := \mathbf{T}, z := \mathbf{T}]\ \mathbf{and}\ (\neg x \vee z)[x := \mathbf{T}, z := \mathbf{F}] = \mathbf{F}. \quad (2)$$

By (1) and (2) we have

$$A[x := \mathbf{T}] = (\exists y(x \wedge y))[x := \mathbf{T}]\ \mathbf{or}\ (\forall z(\neg x \vee z))[x := \mathbf{T}] = \mathbf{T}\ \mathbf{or}\ \mathbf{F} = \mathbf{T}. \quad (3)$$

Now we need to evaluate $A[x := \mathbf{F}]$. By Clause (5) of Definition 12.7, we need to evaluate $(\exists y(x \wedge y))[x := \mathbf{F}]$ and $(\forall z(\neg x \vee z))[x := \mathbf{F}]$.

To evaluate $(\exists y(x \wedge y))[x := \mathbf{F}]$, by Clause (9) of Definition 12.7, we need to evaluate $(x \wedge y)[x := \mathbf{F}, y := \mathbf{T}]$ and $(x \wedge y)[x := \mathbf{F}, y := \mathbf{F}]$.

We have (by Clause (6)) $(x \wedge y)[x := \mathbf{F}, y := \mathbf{T}] = \mathbf{F}\,\mathbf{and}\,\mathbf{T} = \mathbf{F}$ and $(x \wedge y)[x := \mathbf{F}, y := \mathbf{F}] = \mathbf{F}\ \mathbf{and}\ \mathbf{F} = \mathbf{F}$, so

$$(\exists y(x \wedge y))[x := \mathbf{F}] = (x \wedge y)[x := \mathbf{F}, y := \mathbf{T}]\ \mathbf{or}\ (x \wedge y)[x := \mathbf{F}, y := \mathbf{F}] = \mathbf{F}\ \mathbf{or}\ \mathbf{F} = \mathbf{F}. \quad (4)$$

To evaluate $(\forall z(\neg x \vee z))[x := \mathbf{F}]$, by Clause (8) of Definition 12.7, we need to evaluate $(\neg x \vee z)[x := \mathbf{F}, z := \mathbf{T}]$ and $(\neg x \vee z)[x := \mathbf{F}, z := \mathbf{F}]$.

Using Clauses (4) and (5) of Definition 12.7, we have $(\neg x \vee z)[x := \mathbf{F}, z := \mathbf{T}] = \overline{\mathbf{F}}\,\mathbf{or}\,\mathbf{T} = \mathbf{T}$ and $(\neg x \vee z)[x := \mathbf{F}, z := \mathbf{F}] = \overline{\mathbf{F}}\ \mathbf{or}\ \mathbf{F} = \mathbf{T}$, so

$$(\forall z(\neg x \vee z))[x := \mathbf{F}] = (\neg x \vee z)[x := \mathbf{F}, z := \mathbf{T}]\ \mathbf{and}\ (\neg x \vee z)[x := \mathbf{F}, z := \mathbf{F}] = \mathbf{T} \quad (5)$$

By (4) and (5) we have

$$A[x := \mathbf{F}] = (\exists y(x \wedge y))[x := \mathbf{F}] \text{ or } (\forall z(\neg x \vee z))[x := \mathbf{F}] = \mathbf{F} \text{ or } \mathbf{T} = \mathbf{T}. \tag{6}$$

Finally, by (3) and (6) we get

$$A[x := \mathbf{T}] \text{ and } A[x := \mathbf{F}] = \mathbf{T} \text{ and } \mathbf{T} = \mathbf{T}, \tag{7}$$

so A is valid.

The reader should observe that in evaluating

$$(\exists xB)[v] = B[v[x := \mathbf{T}]] \text{ or } B[v[x := \mathbf{F}]],$$

if $(\exists xB)[v] = \mathbf{T}$, it is only necessary to guess which of $B[v[x := \mathbf{T}]]$ or $B[v[x := \mathbf{F}]]$ evaluates to $\mathbf{T}$, so we can view the computation of $A[v]$ as an AND/OR tree, where an AND node corresponds to the evaluation of a formula $(\forall xB)[v]$, and an OR node corresponds to the evaluation of a formula $(\exists xB)[v]$.

Evaluating the truth value $A[v]$ of a QBF A can take exponential time in the size n of A, but we will see that it only requires $O(n^2)$ space. Also, the validity of QBF's of the form

$$\exists x_1 \exists x_2 \cdots \exists x_m B$$

where B is quantifier-free and $FV(B) = \{x_1, \ldots, x_m\}$ is equivalent to SAT (the satisfiability problem), and the validity of QBF's of the form

$$\forall x_1 \forall x_2 \cdots \forall x_m B$$

where B is quantifier-free and $FV(B) = \{x_1, \ldots, x_m\}$ is equivalent to TAUT (the validity problem). This is why the validity problem for QBF's is as hard as both SAT and TAUT.

We mention the following technical results. Part (1) and Part (2) are used all the time.

Proposition 12.5. *Let A be any QBF.*

(1) For any two valuations v_1 and v_2, if $v_1(x) = v_2(x)$ for all $x \in FV(A)$, then $A[v_1] = A[v_2]$. In particular, if A is a sentence, then $A[v]$ is independent of v.

(2) If A' is any rectified QBF obtained from A, then $A[v] = A'[v]$ for all valuations v; that is, $A \equiv A'$.

(3) For any QBF A of the form $A = \forall xB$ and any QBF C such that $BV(B) \cap FV(C) = \emptyset$, if A is valid, then $B[x := C]$ is also valid.

(4) For any QBF B and any QBF C such that $BV(B) \cap FV(C) = \emptyset$, if $B[x := C]$ is valid, then $\exists xB$ is also valid.

We also repeat Proposition 3.16 which states that the connectives $\wedge, \vee, \neg$ and $\exists$ are definable in terms of $\Rightarrow$ and $\forall$. This shows the power of the second-order quantifier $\forall$.

Proposition 12.6. *The connectives $\wedge, \vee, \neg, \perp$ and $\exists$ are definable in terms of $\Rightarrow$ and $\forall$, which means that the following equivalences are valid, where x is not free in B or C:*

$$\begin{aligned}
B \wedge C &\equiv \forall x\big((B \Rightarrow (C \Rightarrow x)) \Rightarrow x\big)\\
B \vee C &\equiv \forall x\big((B \Rightarrow x) \Rightarrow ((C \Rightarrow x) \Rightarrow x)\big)\\
\perp &\equiv \forall x x\\
\neg B &\equiv B \Rightarrow \forall x x\\
\exists y B &\equiv \forall x\big((\forall y(B \Rightarrow x)) \Rightarrow x\big).
\end{aligned}$$

We now prove the first step in establishing that the validity problem for QBF's is $\mathcal{PS}$-complete.

Proposition 12.7. *Let A be any QBF of length n. Then for any valuation v, the truth value $A[v]$ can be evaluated in $O(n^2)$ space. Thus the validity problem for closed QBF's is in $\mathcal{PS}$.*

Proof. The clauses of Definition 12.7 show that $A[v]$ is evaluated recursively. In clauses (5)-(9), even though two recursive calls are performed, it is only necessary to save one of the two stack frames at a time. It follows that the stack will never contain more than n stack frames, and each stack frame has size at most n. Thus only $O(n^2)$ space is needed. For more details, see Hopcroft, Motwani and Ullman [23] (Section 11.3.4, Theorem 11.10). □

Finally we state the main theorem proven by Meyer and Stockmeyer (1973).

Theorem 12.8. *The validity problem for closed QBF's is $\mathcal{PS}$-complete.*

We will not prove Theorem 12.8, mostly because it requires simulating the computation of a polynomial-space bounded deterministic Turing machine, and this is very technical and tedious. Most details of such a proof can be found in Hopcroft, Motwani and Ullman [23] (Section 11.3.4, Theorem 11.11).

Let us simply make the following comment which gives a clue as to why QBF's are helpful in describing the simulation (for details, see Hopcroft, Motwani and Ullman [23] (Theorem 11.11)). It turns out that the idea behind the function *reach* presented in Section 12.2 plays a key role. It is necessary to express for any two ID's I and J and any $i \geq 1$, that $I \vdash^k J$ with $k \leq i$. This is achieved by defining $N_{2i}(I, J)$ as the following QBF:

$$N_{2i}(I, J) = \exists K \forall R \forall S\Big(\big((R = I \wedge S = K) \vee (R = K \wedge S = J)\big) \Rightarrow N_i(R, S)\Big).$$

Another interesting $\mathcal{PS}$-complete problem due to Karp (1972) is the following. Given any alphabet Σ, decide whether a regular expression R denotes Σ^*; that is, $\mathcal{L}[R] = \Sigma^*$.

We conclude with some comments regarding some remarkable results of Statman regarding the connection between validity of closed QBF's and provability in intuitionistic propositional logic.

12.4 Complexity of Provability in Intuitionistic Propositional Logic

Recall that intuitionistic logic is obtained from classical logic by taking away the proof-by-contradiction rule. The reader is strongly advised to review Chapter **??**, especially Sections **??**, **??**, **??**, **??** and **??**, before proceeding.

Statman [40] shows how to reduce the validity problem for QBFs to provability in intuitionistic propositional logic. To simplify the construction we may assume that we consider QBF's in *prenex form*, which means that they are of the form

$$A = Q_n x_n Q_{n-1} x_{n-1} \cdots Q_1 x_1 B_0$$

where B_0 is quantifier-free and $Q_i \in \{\forall, \exists\}$ for $i = 1, \ldots, n$. We also assume that A is rectified. It is easy to show that any QBF A is equivalent to some QBF A' in prenex form by adapting the method for converting a first-order formula to prenex form; see Gallier [18] or Shoenfield [39].

Statman's clever trick is to exploit some properties of intuitionistic provability that do not hold for classical logic. One of these properties is that if a proposition $B \vee C$ is provable intuitionistically, we write $\vdash_I B \vee C$, then either $\vdash_I B$ or $\vdash_I C$, that is, either B is provable or C is provable (of course, intuitionistically). This fact is used in the "easy direction" of the proof of Theorem 12.9.

To illustrate the power of the above fact, in his construction, Statman associates the proposition

$$(x \Rightarrow B) \vee (\neg x \Rightarrow B) \qquad (*)$$

to the QBF $\exists x B$. Classically this is useless, because $(*)$ is classically valid, but if $(*)$ is *intuitionistically provable*, then either $x \Rightarrow B$ is provable or $\neg x \Rightarrow B$ is intuitionistically provable, but this implies that either $x \Rightarrow B$ is classically provable or $\neg x \Rightarrow B$ is classically provable, and so either $B[x := \mathbf{T}]$ is valid or $B[x := \mathbf{F}]$ is valid, which means that $\exists x B$ is valid.

As a first step, Statman defines the proposition B_k^+ inductively as follows: for all k such that $0 \leq k \leq n-1$,

$$\begin{aligned}
B_0^+ &= \neg\neg B_0 \\
B_{k+1}^+ &= (x_{k+1} \vee \neg x_{k+1}) \Rightarrow B_k^+ && \text{if } Q_{k+1} = \forall \\
B_{k+1}^+ &= (x_{k+1} \Rightarrow B_k^+) \vee (\neg x_{k+1} \Rightarrow B_k^+), && \text{if } Q_{k+1} = \exists
\end{aligned}$$

and set $A^+ = B_n^+$. Obviously A^+ is quantifier-free. We also let $B_{k+1} = Q_{k+1} x_{k+1} B_k$ for $k = 0, \ldots n-1$, so that $A = B_n$.

The following example illustrates the above definition.

Example 12.3. Consider the QBF is prenex form

$$A = \exists x_3 \forall x_2 \exists x_1 ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_1)).$$

It is indeed valid, as we see by setting $x_3 = \mathbf{F}$, and if $x_2 = \mathbf{T}$ then $x_1 = \mathbf{F}$, else if $x_2 = \mathbf{F}$ then $x_1 = \mathbf{T}$. We have

$$\begin{aligned} B_0^+ &= \neg\neg((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_1)) \\ B_1^+ &= (x_1 \Rightarrow B_0^+) \vee (\neg x_1 \Rightarrow B_0^+) \\ B_2^+ &= (x_2 \vee \neg x_2) \Rightarrow B_1^+ \\ B_3^+ &= (x_3 \Rightarrow B_2^+) \vee (\neg x_3 \Rightarrow B_2^+), \end{aligned}$$

and $A^+ = B_3^+$.

Statman proves the following remarkable result (Statman [40], Proposition 1).

Theorem 12.9. *For any closed QBF A in prenex form, A is valid iff $\vdash_I A^+$; that is, A^+ is intuitionistically provable.*

Proof sketch. Here is a sketch of Statman's proof using the QBF of Example 12.3. First assume the QBF A is valid. The first step is to eliminate existential quantifiers using a variant of what is known as Skolem functions; see Gallier [18] or Shoenfield [39].

The process is to assign to the jth existential quantifier $\exists x_k$ from the left in the formula $Q_n x_n \cdots Q_1 x_1 B_0$ a boolean function C_j depending on the universal quantifiers $\forall x_{i_1}, \ldots, \forall x_{i_p}$ to the left of $\exists x_k$ and defined such that $Q_n x_n \cdots Q_{k+1} x_{k+1} \exists x_k B_{k-1}$ is valid iff $\forall x_{i_1} \cdots \forall x_{i_q} B_{k-1}^s$ is valid, where B_{k-1}^s is the result of substituting the functions $C_1, \ldots, C_j$ associated with the j existential quantifiers from the left for these existentially quantified variables.

We associate with $\exists x_3$ the constant C_1 such that $C_1 = \mathbf{F}$, and with $\exists x_1$ the boolean function $C_2(x_2)$ given by

$$C_2(\mathbf{T}) = \mathbf{F}, \quad C_2(\mathbf{F}) = \mathbf{T}.$$

The constant C_1 and the function C_2 are chosen so that

$$A = \exists x_3 \forall x_2 \exists x_1 ((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_1))$$

is valid iff

$$A^s = \forall x_2((C_2(x_2) \vee x_2) \wedge (\neg C_2(x_2) \vee \neg x_2) \wedge (\neg C_1 \vee C_2(x_2))) \tag{S}$$

is valid. Indeed, since $C_1 = \mathbf{F}$, the clause $(\neg C_1 \vee C_2(x_2))$ evaluates to $\mathbf{T}$ regardless of the value of x_2, and by definition of C_2, the expression

$$\forall x_2 (C_2(x_2) \vee x_2) \wedge (\neg C_2(x_2) \vee \neg x_2))$$

also evaluates to $\mathbf{T}$. We now build a tree of Gentzen sequents (from the root up) from the expression in (S) which guides us in deciding which disjunct to pick when dealing with a proposition B_k^+ associated with an existential quantifier. Here is the tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\neg x_3, x_2, \neg x_1 \to B_0^+}{\neg x_3, x_2 \to \neg x_1 \Rightarrow B_0^+}}{\neg x_3, x_2 \to (x_1 \Rightarrow B_0^+) \lor (\neg x_1 \Rightarrow B_0^+)} \qquad \dfrac{\dfrac{\neg x_3, \neg x_2, x_1 \to B_0^+}{\neg x_3, \neg x_2 \to x_1 \Rightarrow B_0^+}}{\neg x_3, \neg x_2 \to (x_1 \Rightarrow B_0^+) \lor (\neg x_1 \Rightarrow B_0^+)}}{\neg x_3, x_2 \lor \neg x_2 \to B_1^+}}{\neg x_3 \to (x_2 \lor \neg x_2) \Rightarrow B_1^+}}{\to \neg x_3 \Rightarrow B_2^+}}{\to (x_3 \Rightarrow B_2^+) \lor (\neg x_3 \Rightarrow B_2^+)}$$

We will see that by adding subtrees proving the sequents in the leaf nodes, this tree becomes an intuitionistic proof of A^+. Note that this a proof in a Gentzen sequent style formulation of intuitionistic logic (see Kleene [24], Gallier [13], Takeuti [42]), not a proof in a natural deduction style proof system as in Section **??**.

The tree is constructed from the bottom-up starting with $\to A^+$. For every leaf node in the tree where a sequent is of the form

$$\ell_n, \ldots, \ell_{k+1} \to (x_k \Rightarrow B_{k-1}^+) \lor (\neg x_k \Rightarrow B_{k-1}^+)$$

where $\ell_n, \ldots, \ell_{k+1}$ are literals, we know that $Q_k = \exists$ is the jth existential quantifier from the left, so we use the boolean function C_j to determine which of the two disjuncts $x_k \Rightarrow B_{k-1}^+$ or $\neg x_k \Rightarrow B_{k-1}^+$ to keep. The function C_j depends on the value of the literals $\ell_n, \ldots, \ell_{k+1}$ associated with universal quantifiers (where ℓ_i has the value $\mathbf{T}$ if $\ell_i = x_i$ and ℓ_i has the value $\mathbf{F}$ if $\ell_i = \neg x_i$). Even though C_j is independent of the value of the literals ℓ_i associated with existential quantifiers, to simplify notation we write $C_j(\ell_n, \ldots, \ell_{k+1})$ for the value of the function C_j. If $C_j(\ell_n, \ldots, \ell_{k+1}) = \mathbf{T}$, then we pick the disjunct $x_k \Rightarrow B_{k-1}^+$, else if $C_j(\ell_n, \ldots, \ell_{k+1}) = \mathbf{F}$, then we pick the disjunct $\neg x_k \Rightarrow B_{k-1}^+$. Denote the literal corresponding to the chosen disjunct by ℓ_k ($\ell_k = x_k$ in the first case, $\ell_k = \neg x_k$ in the second case). Then we grow two new nodes

$$\ell_n, \ldots, \ell_{k+1} \to \ell_k \Rightarrow B_{k-1}^+$$

and

$$\ell_n, \ldots, \ell_{k+1}, \ell_k \to B_{k-1}^+$$

above the (leaf) node

$$\ell_n, \ldots, \ell_{k+1} \to (x_k \Rightarrow B_{k-1}^+) \lor (\neg x_k \Rightarrow B_{k-1}^+).$$

For every leaf node of the form

$$\ell_n, \ldots, \ell_{k+1} \to (x_k \lor \neg x_k) \Rightarrow B_{k-1}^+,$$

we grow the new node

$$\ell_n, \ldots, \ell_{k+1}, x_k \lor \neg x_k \to B_{k-1}^+,$$

and then the two new nodes (both descendants of the above node, so there is branching in the tree),

$$\ell_n, \ldots, \ell_{k+1}, x_k \to B^+_{k-1} \quad \text{and} \quad \ell_n, \ldots, \ell_{k+1}, \neg x_k \to B^+_{k-1}.$$

By induction from the bottom-up, since A is valid and since the tree was constructed in terms of the constant C_1 and the function C_2 which ensure the validity of A, it is easy to see that for every node $\ell_n, \ldots, \ell_{k+1} \to B^+_k$, the sequent $\ell_n, \ldots, \ell_{k+1} \to B_k$ (note, the right-hand side is the original formula B_k) is classically valid, and thus classically provable (by the completeness theorem for propositional logic). Consequently *every leaf* $\ell_n, \ldots, \ell_1 \to B_0$ is classically provable, so by Glivenko's theorem (see Kleene [24] (Theorem 59), or Gallier [13] (Section 13)), the sequent $\ell_n, \ldots, \ell_1 \to \neg\neg B_0$ is *intuitionistically provable*. But this is the sequent $\ell_n, \ldots, \ell_1 \to B^+_0$ so all the leaves of the tree are intuitionistically provable, and since the tree is a deduction tree in a Gentzen sequent style formulation of intuitionistic logic (see Kleene [24], Gallier [13], Takeuti [42]), the root $A^+ = B^+_n$ is intuitionistically provable.

In the other direction, assume that A^+ is intuitionistically provable. We use the fact that if

$$\ell_n, \ldots, \ell_k \to A \vee B$$

is intuitionistically provable and the ℓ_i are literals, then either $\ell_n, \ldots, \ell_k \to A$ is intuitionistically provable or $\ell_n, \ldots, \ell_k \to B$ is intuitionistically provable, and other proof rules of intuitionistic logic (see Kleene [24], Gallier [13], Takeuti [42]), to build a proof tree just as we did before. Then every sequent $\ell_n, \ldots, \ell_{k+1} \to B^+_k$ is intuitionistically provable, thus classically provable, and consequently classically valid. But this immediately implies (by induction starting from the leaves) that $\ell_n, \ldots, \ell_{k+1} \to B_k$ is also classically valid for all k, and thus $A = B_n$ is valid. □

Statman does not specifically state which proof system of intuitionistic logic is used in Theorem 12.9. Careful inspection of the proof shows that we can construct proof trees in a Gentzen sequent calculus as described in Gallier [13] (system $\mathcal{G}_i$, Section 4) or Kleene [24] (system G3a, Section 80, pages 481-482). This brings up the following issue: could we use instead proofs in natural deduction style, as in Prawitz [35] or Gallier [13]? The answer is yes, because there is a polynomial-time translation of intuitionistic proofs in Gentzen sequent style to intuitionistic proofs in natural deduction style, as shown in Gallier [13], Section 5. So Theorem 12.9 applies to a Gentzen sequent style proof system or to a natural deduction style proof system.

The problem with the translation $A \mapsto A^+$ is that A^+ may not have size polynomial in the size (the length of A as a string) of A because in the case of an existential quantifier the length of the formula B^+_{k+1} is more than twice the length of the formula B^+_k, so Statman introduces a second translation.

The proposition $B^\dagger_k$ is defined inductively as follows. Let $y_0, y_1, \ldots, y_n$ be $n + 1$ new propositional variables. For all k such that $0 \leq k \leq n-1$,

$$\begin{aligned}
B_0^\dagger &= \neg\neg B_0 \equiv y_0 \\
B_{k+1}^\dagger &= ((x_{k+1} \vee \neg x_{k+1}) \Rightarrow y_k) \equiv y_{k+1} && \text{if } Q_{k+1} = \forall \\
B_{k+1}^\dagger &= ((x_{k+1} \Rightarrow y_k) \vee (\neg x_{k+1} \Rightarrow y_k)) \equiv y_{k+1}, && \text{if } Q_{k+1} = \exists
\end{aligned}$$

and set

$$A^* = B_0^\dagger \Rightarrow (B_1^\dagger \Rightarrow (\cdots (B_n^\dagger \Rightarrow y_n) \cdots)).$$

It is easy to see that the translation $A \mapsto A^*$ can be done in polynomial space. Statman proves the following result (Statman [40], Proposition 2).

Theorem 12.10. *For any closed QBF A in prenex form, $\vdash_I A^+$ iff $\vdash_I A^*$; that is, A^+ is intuitionistically provable iff A^* is intuitionistically provable.*

Proof. First suppose the sequent $\rightarrow A^+$ is provable (in Kleene G3a). We claim that the sequent

$$B_0^\dagger, \ldots, B_k^\dagger \rightarrow B_k^+ \equiv y_k$$

is provable for $k = 0, \ldots, n$. We proceed by induction on k. For the base case $k = 0$, we have $B_0^\dagger = (\neg\neg B_0 \equiv y_0)$ and $B_0^+ = \neg\neg B_0$, so $B_0^\dagger \rightarrow (B_0^+ \equiv y_0) = (B_0^+ \equiv y_0) \rightarrow (B_0^+ \equiv y_0)$, which is an axiom.

For the induction step, if $Q_{k+1} = \forall$, then

$$B_{k+1}^+ = (x_{k+1} \vee \neg x_{k+1}) \Rightarrow B_k^+, \qquad B_{k+1}^\dagger = ((x_{k+1} \vee \neg x_{k+1}) \Rightarrow y_k) \equiv y_{k+1},$$

by the induction hypothesis

$$B_0^\dagger, \ldots, B_k^\dagger \rightarrow B_k^+ \equiv y_k$$

is provable, and since the sequent

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \rightarrow B_{k+1}^\dagger$$

is an axiom, by substituting B_k^+ for y_k in $B_{k+1}^\dagger = ((x_{k+1} \vee \neg x_{k+1}) \Rightarrow y_k) \equiv y_{k+1}$ in the conclusion of the above sequent, we deduce that

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \rightarrow ((x_{k+1} \vee \neg x_{k+1}) \Rightarrow B_k^+) \equiv y_{k+1}$$

is provable. Since $B_{k+1}^+ = (x_{k+1} \vee \neg x_{k+1}) \Rightarrow B_k^+$, we conclude that

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \rightarrow B_{k+1}^+ \equiv y_{k+1}$$

is provable.

If $Q_{k+1} = \exists$, then

$$B_{k+1}^+ = (x_{k+1} \Rightarrow B_k^+) \vee (\neg x_{k+1} \Rightarrow B_k^+), \qquad B_{k+1}^\dagger = ((x_{k+1} \Rightarrow y_k) \vee (x_{k+1} \Rightarrow y_k)) \equiv y_{k+1},$$

by the induction hypothesis

$$B_0^\dagger, \ldots, B_k^\dagger \to B_k^+ \equiv y_k$$

is provable, and since the sequent

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \to B_{k+1}^\dagger$$

is an axiom, by substituting B_k^+ for y_k in $B_{k+1}^\dagger = ((x_{k+1} \Rightarrow y_k) \vee (x_{k+1} \Rightarrow y_k)) \equiv y_{k+1}$ in the conclusion of the above sequent, we deduce that

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \to ((x_{k+1} \Rightarrow B_k^+) \vee (\neg x_{k+1} \Rightarrow B_k^+)) \equiv y_{k+1}$$

is provable. Since $B_{k+1}^+ = (x_{k+1} \Rightarrow B_k^+) \vee (\neg x_{k+1} \Rightarrow B_k^+)$, we conclude that

$$B_0^\dagger, \ldots, B_k^\dagger, B_{k+1}^\dagger \to B_{k+1}^+ \equiv y_{k+1}$$

is provable. Therefore the induction step holds. For $k = n$, we see that the sequent

$$B_0^\dagger, \ldots, B_n^\dagger \to (B_n^+ \equiv y_n) = B_0^\dagger, \ldots, B_n^\dagger \to (A^+ \equiv y_n)$$

is provable, and since by hypothesis $\to A^+$ is provable, we deduce that

$$B_0^\dagger, \ldots, B_n^\dagger \to y_n$$

is provable. Finally we deduce that

$$A^* = B_0^\dagger \Rightarrow (B_1^\dagger \Rightarrow (\cdots (B_n^\dagger \Rightarrow y_n) \cdots))$$

is provable intuitionistically.

Conversely assume that $A^* = B_0^\dagger \Rightarrow (B_1^\dagger \Rightarrow (\cdots (B_n^\dagger \Rightarrow y_n) \cdots))$ is provable intuitionistically. Then using basic properties of intuitionistic provability, the sequent

$$B_0^\dagger, \ldots, B_n^\dagger \to y_n$$

is provable intuitionistically. Now if we substitute B_{k+1}^+ for y_{k+1} in $B_{k+1}^\dagger$ for $k = 0, \ldots, n-1$, we see immediately that

$$B_{k+1}^\dagger[y_{k+1} := B_{k+1}^+] = B_{k+1}^+ \equiv B_{k+1}^+,$$

so the proof of

$$B_0^\dagger, \ldots, B_n^\dagger \to y_n$$

yields a proof of

$$B_0^+ \equiv B_0^+, \ldots, B_n^+ \equiv B_n^+ \to B_n^+,$$

that is, a proof (intuitionistic) of $B_n^+ = A^+$.

Remark: Note that we made implicit use of the cut rule several times, but by Gentzen's cut-elimination theorem this does not matter (see Gallier [13]). □

Using Theorems 12.9 and 12.10 we deduce from the fact that validity of QBF's is $\mathcal{PS}$-complete that provability in propositional intuitionistic logic is $\mathcal{PS}$-hard (every problem in $\mathcal{PS}$ reduces in polynomial time to provability in propositional intuitionistic logic). Using results of Tarski and Ladner, it is can be shown that validity in Kripke models for propositional intuitionistic logic belongs to $\mathcal{PS}$, so Statman proves the following result (Statman [40], Section 2, Theorem).

Theorem 12.11. *The problem of deciding whether a proposition is valid in all Kripke models is $\mathcal{PS}$-complete.*

Theorem 12.11 also applies to any proof system for intuitionisic logic which is sound and complete for Kripke semantics.

Theorem 12.12. *The problem of deciding whether a proposition is intuitionistically provable in any sound and complete proof system (for Kripke semantics) is $\mathcal{PS}$-complete.*

Theorem 12.12 applies to Gallier's system $\mathcal{G}_i$, to Kleene's system G3a, and to natural deduction systems. To prove that $\mathcal{G}_i$ is complete for Kripke semantics it is better to convert proofs in $\mathcal{G}_i$ to proofs in a system due to Takeuti, the system denoted $\mathcal{GKT}_i$ in Gallier [13]; see Section 9, Definition 9.3. Since there is a polynomial-time translation of proofs in $\mathcal{G}_i$ to proofs in natural deduction, the latter system is also complete. This is also proven in van Dalen [44].

Statman proves an even stronger remarkable result, namely that $\mathcal{PS}$-completeness holds even for propositions using only the connective $\Rightarrow$ (Statman [40], Section 2, Proposition 3).

Theorem 12.13. *There is an algorithm which given any proposition A constructs another proposition $A^\sharp$ only involving $\perp, \Rightarrow$, such that that $\vdash_I A$ iff $\vdash_I A^\sharp$.*

Theorem 12.13 is somewhat surprising in view of the fact that $\vee, \wedge, \Rightarrow$ are independent connectives in propositional intuitionistic logic. Finally Statman obtains the following beautiful result (Statman [40], Section 2, Theorem).

Theorem 12.14. *The problem of deciding whether a proposition only involving $\perp, \Rightarrow$ is valid in all Kripke models, and intuitionistically provable in any sound and complete proof system, is $\mathcal{PS}$-complete.*

We highly recommend reading Statman [40], but we warn the reader that this requires perseverance.

Bibliography

[1] Tom M. Apostol. *Introduction to Analytic Number Theory.* Undergraduate Texts in Mathematics. Springer, first edition, 1976.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity. A Modern Approach.* Cambridge University Press, first edition, 2009.

[3] H.P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* Studies in Logic No. 103. North-Holland, second revised edition, 1984.

[4] H.P. Barendregt. Lambda Calculi With Types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[5] Richard Crandall and Carl Pomerance. *Prime Numbers. A Computational Perspective.* Springer, second edition, 2005.

[6] H. Davenport. *The Higher Arithmetic.* Cambridge University Press, eighth edition, 2008.

[7] Martin Davis. *Computability and Unsolvability.* McGraw-Hill, first edition, 1958.

[8] Martin Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80(3):233–269, 1973.

[9] Martin Davis, Yuri Matijasevich, and Julia Robinson. Hilbert's Tenth Problem. Diophantine Equations: Positive Aspects of a Negative Solution. In *Mathematical Developments Arising from Hilbert Problems*, volume XXVIII, Part 2, pages 323–378. AMS, 1976.

[10] Burton Dreben and Warren D. Goldfarb. *The Decision Problem. Solvable Classes of Quantificational Formulas.* Addison-Wesley, first edition, 1979.

[11] Herbert B. Enderton. *A Mathematical Introduction to Logic.* Elsevier, second edition, 2001.

[12] Jean Gallier. What's so special about Kruskal's theorem and the ordinal Γ_0? A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53:199–260, 1991.

[13] Jean Gallier. Constructive Logics. Part I: A Tutorial on Proof Systems and Typed λ-Calculi. *Theoretical Computer Science*, 110(2):249–339, 1993.

[14] Jean Gallier. On the Correspondence Between Proofs and λ-Terms. In Philippe de Groote, editor, *Cahiers Du Centre de Logique, Vol. 8*, pages 55–138. Academia, Louvain-La-Neuve, 1995.

[15] Jean Gallier and Jocelyn Quaintance. Notes on Primality Testing and Public Key Cryptography. Part I: Randomized Algorithms, Miller–Rabin and Solovay–Strassen Tests. Technical report, University of Pennsylvania, Levine Hall, Philadelphia, PA 19104, 2017. pdf file available from http://www.cis.upenn.edu/∼jean/RSA-primality-testing.pdf.

[16] Jean H. Gallier. On Girard's "candidats de reductibilité". In P. Odifreddi, editor, *Logic And Computer Science*, pages 123–203. Academic Press, London, New York, May 1990.

[17] Jean H. Gallier. *Discrete Mathematics.* Universitext. Springer-Verlag, first edition, 2011.

[18] Jean H. Gallier. *Logic For Computer Science; Foundations of Automatic Theorem Proving.* Dover, second edition, 2015.

[19] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[20] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *Proc. 2nd Scand. Log. Symp.*, pages 63–92. North-Holland, 1971.

[21] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université de Paris VII, June 1972. Thèse de Doctorat d'Etat.

[22] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student texts.* Cambridge University Press, 1986.

[23] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Automata Theory, Languages,and Computation.* Addison-Wesley, third edition, 2007.

[24] Stephen Kleene. *Introduction to Metamathematics.* North-Holland, seventh edition, 1952.

[25] Stephen Kleene. *Mathematical Logic.* Wiley, first edition, 1967.

[26] Jean-Louis Krivine. *Lambda-calcul types et modèles.* Masson, first edition, 1990.

[27] Harry R. Lewis. *Unsolvable Classes of Quantificational Formulas.* Addison-Wesley, first edition, 1979.

[28] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, second edition, 1997.

[29] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms.* Elsevier North-Holland, first edition, 1978.

[30] Zohar Manna. *Mathematical Theory of Computation.* Dover, first edition, 2003.

[31] Ivan Niven, Herbert S. Zuckerman, and Hugh L. Montgomery. *An Introduction to the Theory of Numbers.* Wiley, fifth edition, 1991.

[32] Christos H. Papadimitriou. *Computational Complexity.* Pearson, first edition, 1993.

[33] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, first edition, 2002.

[34] Vaughan R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[35] D. Prawitz. *Natural deduction, a proof-theoretical study.* Almquist & Wiksell, Stockholm, 1965.

[36] Paulo Ribenboim. *The Little Book of Bigger Primes.* Springer-Verlag, second edition, 2004.

[37] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, first edition, 1987.

[38] Kurt Schütte. *Proof Theory*, volume 225 of *Grundlehren der Mathematischen Wissenschaften.* Springer-Verlag, 1977.

[39] Joseph Shoenfield. *Mathematical Logic.* Addison-Wesley, first edition, 2001.

[40] Richard Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67–72, 1979.

[41] Richard Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, 1979.

[42] G. Takeuti. *Proof Theory*, volume 81 of *Studies in Logic.* North-Holland, 1975.

[43] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1996.

[44] D. van Dalen. *Logic and Structure.* Universitext. Springer-Verlag, second edition, 1980.

Symbol Index

Index