Computability Theory

– Monograph –

Karl-Heinz Zimmermann

Computability Theory

– Monograph –

Hamburg University of Technology

Prof. Dr. Karl-Heinz Zimmermann
Hamburg University of Technology
21071 Hamburg
Germany

This monograph is listed in the GBV database and the TUHH library.

urn:nbn:de:gbv:830-88214582

For Gela and Eileen

Preface

Why do we need a formalization of the notion of algorithm or effective computation? In order to show that a specific problem is algorithmically solvable, it is sufficient to provide an algorithm that solves it in a sufficiently precise manner. However, in order to prove that a problem is in principle not computable by an algorithm, a rigorous formalism is necessary that allows mathematical proofs. The need for such a formalism became apparent in the studies of David Hilbert (1900) on the foundations of mathematics and Kurt Gödel (1931) on the incompleteness of elementary arithmetic.

The first investigations in this field were conducted by the logicians Alonzo Church, Stephen Kleene, Emil Post, and Alan Turing in the early 1930s. They have provided the foundation of computability theory as a branch of theoretical computer science. The fundamental results established Turing computability as the correct formalization of the informal idea of effective calculation. The results have led to Church's thesis stating that "everything computable is computable by a Turing machine". The theory of computability has grown rapidly from its beginning. Its questions and methods are penetrating many other mathematical disciplines. Today, computability theory provides an important theoretical background for formal logicians, pure mathematicians, and theoretical computer scientists. Many mathematical problems are known to be undecidable such as the word problems for semigroup and groups, the word problems for string rewriting systems (semi-Thue systems), the halting problem, and the solvability of diophantine equations (Hilbert's tenth problem).

This book is a development of class notes for a two-hour lecture including a two-hour lab held for second-year Bachelor students of Computer Science at the Hamburg University of Technology during the last few years. The course aims to present the basic results of computability theory, including well-known mathematical models of computability such as the Turing machine, the unlimited register machine (URM), and GOTO and LOOP programs, the principal classes of computational functions like the classes of primitive recursive, recursive, and partial recursive functions, the famous Ackermann function, the main theoretical concepts of computability such as Gödel numbering, universal functions, parametrization, Kleene's normal form, Kleene's recursion theorem, the theorems of Rice, and undecidable and semidecidable (or recursively enumerable) sets, and last but not least several important undecidable word problems.

The manuscript has partly grown out of notes taken by the author during his studies at the University of Erlangen-Nuremberg. An added chapter provides a brief presentation of the central open question in complexity theory which is one of the millenium price problems in mathematics asking roughly whether each problem whose solution can be verified in polynomial time can also be solved in polynomial time. This chapter includes the well-known result of Stephen Cook and Leonid Lewin that the satisfiabilty problem is NP-complete and also its proof from scratch. Finally, an appendix provides the necessary mathematical background on semigroups and monoids.

First, I would like to thank my teachers Martin Becker† and Volker Strehl for their inspiring lectures in this field. Moreover, I would like to express my thanks to Ralf Möller for valuable comments. I am also grateful to Natalia Schmidt, Mahwish Saleemi, and Robert Leppert for conducting the labs and Wolfgang Brandt for valuable technical support. Finally, I would like to thank my students for their attention, their stimulating questions, and their dedicated work.

Hamburg, July 2016 *Karl-Heinz Zimmermann*

Mathematical Notation

General notation

$\mathbb{N}_0$	monoid of natural numbers
$\mathbb{N}$	semigroup of natural number without 0
$\mathbb{Z}$	ring of integers
$\mathbb{Q}$	field of rational number
$\mathbb{R}$	field of real number
$\mathbb{C}$	field of complex numbers
2^A	power set of A
Σ^*	set of words over Σ
ϵ	empty word
Σ^+	set of non-empty words over Σ
$\mathcal{F}$	class of partial functions
$\mathcal{P}$	class of primitive recursive functions
$\mathcal{R}$	class of partial recursive functions
$\mathcal{T}$	class of recursive functions
$\mathcal{F}_{\text{URM}}$	class of URM computable functions
$\mathcal{T}_{\text{URM}}$	class of total URM computable functions
$\mathcal{F}_{\text{LOOP}}$	class of LOOP computable functions
$\mathcal{P}_{\text{LOOP}}$	class of LOOP programs
$\mathcal{F}_{\text{LOOP}-n}$	class of LOOP-n computable functions
$\mathcal{P}_{\text{LOOP}-n}$	class of LOOP-n programs
$\mathcal{F}_{\text{GOTO}}$	class of GOTO computable functions
$\mathcal{P}_{\text{GOTO}}$	class of GOTO programs
$\mathcal{P}_{\text{SGOTO}}$	class of SGOTO programs
$\mathcal{T}_{\text{Turing}}$	class of Turing computable functions

Chapter 1

Ω	state set of URM
$E(\Omega)$	set of state transitions
$\text{dom} f$	domain of a function
$\text{ran} f$	range of a function
$\dot{-}$	conditional decrement
sgn	sign function
csg	cosign function
f^{*n}	iteration of f w.r.t. nth register
$A\sigma$	increment
$S\sigma$	conditional decrement
$(P)\sigma$	iteration of a program

$P;Q$	composition of programs
$\lvert P\rvert$	state transition function of a program
$\lVert P\rVert_{k,m}$	function of a program
α_k	load function
β_m	result function
$R(i;j_1,\ldots,j_k)$	reload program
$C(i;j_1,\ldots,j_k)$	copy program

Chapter 2

ν	successor function
$c_0^{(n)}$	n-ary zero function
$\pi_k^{(n)}$	n-ary projection function
$\mathrm{pr}(g,h)$	primitive recursion
$g(h_1,\ldots,h_n)$	composition
$\varSigma f$	bounded sum
$\varPi f$	bounded product
$\overline{\mu}f$	bounded minimalization
J_2	Cantor pairing function
K_2, L_2	inverse component functions of J_2
χ_S	characteristic function of a set S
p_i	ith prime
$(x)_i$	ith exponent in prime-power representation of x
$Z\sigma$	zero-setting of register σ
$\overline{C}(\sigma,\tau)$	copy program
$[P]\sigma$	iteration of a program

Chapter 3

μf	unbounded minimalization
$(l, x_i \leftarrow x_i + 1, m)$	GOTO increment
$(l, x_i \leftarrow x_i \dot{-} 1, m)$	GOTO conditional decrement
$(l, \texttt{if}\ x_i = 0, k, m)$	GOTO conditional jump
$V(P)$	set of GOTO variables
$L(P)$	set of GOTO labels
$\vdash$	one-step computation
G	encoding of URM state
G_i	URM state of ith register
$M(k)$	GOTO-2 multiplication
$D(k)$	GOTO-2 division
$T(k)$	GOTO-2 divisibility test

Chapter 4

B_n	small Ackermann function

A	Ackermann function
$\gamma(P)$	runtime of LOOP program
$\lambda(P)$	complexity of LOOP program
$f \leq g$	function bound

Chapter 5

J	encoding of $\mathbb{N}_0^*$
K, L	inverse component functions of J
lg	length function
$I(s_l)$	encoding of SGOTO statement
$\varGamma(P)$	Gödel number of SGOTO program
P_e	SGOTO program with Gödel number e
$\phi_e^{(n)}$	n-ary computable function with index e
$s_{m,n}$	smn function
$\psi_{\text{univ}}^{(n)}$	n-ary universal function
E_A	unbounded existential quantification
U_A	unbounded universal quantification
μA	unbounded minimalization
S_n	extended Kleene set
T_n	Kleene set

Chapter 6

M	Turing machine
b	blank symbol
$\varSigma$	tape alphabet
$\varSigma_I$	input alphabet
Q	state set
T	state transition function
q_0	initial state
q_F	final state
L	left move
R	right move
$\varDelta$	no move
$\vdash$	one-step computation

Chapter 7

K	prototype of undecidable set
H	halting problem
$\mathcal{A}$	class of monadic partial recursive functions
$f_\uparrow$	nowhere-defined function
prog($\mathcal{A}$)	set of indices of $\mathcal{A}$
r.e.	recursive enumerable

$f \subseteq g$	ordering relation

Chapter 8

Σ	alphabet
R	rule set of STS
$\rightarrow$	rule
$\rightarrow_R$	one-step rewriting rule
$\rightarrow_R^*$	reflexive transitive closure
$R^{(s)}$	symmetric rule set of STS
$\rightarrow_{R^{(s)}}^*$	reflexive transitive symmetric hull
$[s]$	equivalence class
$\circ$	concatenation of words
Π	rule set of PCS
$\overline{\Sigma}$	extended alphabet of PCS
$\underline{\boldsymbol{i}}$	solution of PCS
$\alpha(\underline{\boldsymbol{i}})$	left string of solution
$\beta(\underline{\boldsymbol{i}})$	right string of solution
$p(X_1, \ldots, X_n)$	diophantine polynomial
$V(p)$	natural variety

Chapter 9

T	time constructible function
$\lvert x \rvert$	length of string x
$f = O(g)$	Landau notation (big-Oh)
$f = o(g)$	Landau notation (little-Oh)
$f = \Theta(g)$	Landau notation
P	complexity class
DTIME	complexity class
NP	complexity class
EXP	complexity class
NTIME	complexity class
$\leq_p$	reducibility
$\neg$	Boolean negation
$\wedge$	Boolean and
$\vee$	Boolean or

Appendix

S	semigroup
M	monoid
e	identity element
$P(X)$	power set of X
$\mathbb{Z}^{2\times 2}$	matrix monoid

O	zero matrix
I	unit matrix
Σ^*	word monoid over Σ
T_X	full transformation monoid of X
ℓ_x	left multiplication with x
L_M	monoid of left multiplications of M
$\langle X \rangle$	submonoid generated by X
$\mathrm{im}(\phi)$	image of homomorphism ϕ
$\ker(\phi)$	kernel of homomorphism ϕ
$\mathrm{End}(M)$	monoid of endomorphisms of M

Contents

1

Unlimited Register Machine

The unlimited register machine (URM) introduced by Sheperdson and Sturgis (1963) is an abstract computing machine that allows to make precise the notion of computability. It consists of an infinite (unlimited) sequence of registers each capable of storing a natural number which can be arbitrarily large. The registers can be manipulated by using simple instructions. This chapter introduces the syntax and semantics of URMs and the class of URM computable functions.

1.1 States and State Transformations

An unlimited register machine (URM) contains an infinite number of registers named

$$R_0, R_1, R_2, R_3, \ldots. \tag{1.1}$$

The *state set* of an URM is given as

$$\Omega = \{\omega : \mathbb{N}_0 \to \mathbb{N}_0 \mid \omega \text{ is 0 almost everywhere}\}. \tag{1.2}$$

The elements of Ω are denoted as sequences

$$\omega = (\omega_0, \omega_1, \omega_2, \omega_3, \ldots), \tag{1.3}$$

where for each $n \in \mathbb{N}_0$, the component $\omega_n = \omega(n)$ denotes the content of the register R_n.

Proposition 1.1. *The set Ω is countable.*

Proof. Let $(p_0, p_1, p_2, \ldots)$ denote the sequence of prime numbers. Due to the unique factorization of each natural number into a product of prime powers and the states being 0 almost everywhere, the mapping

$$\Omega \to \mathbb{N} : \omega \mapsto \prod_i p_i^{\omega_i}$$

is a bijection. □

Let $E(\Omega)$ denote the set of all partial functions from Ω to Ω. Note that *partial* means that for each function $f \in E(\Omega)$ and each state $\omega \in \Omega$ there does not necessarily exist a value $f(\omega)$. Each partial function $f \in E(\Omega)$ has a *domain* given as

$$\mathrm{dom}(f) = \{\omega \in \Omega \mid f(\omega) \text{ is defined}\} \tag{1.4}$$

and a *range* defined by

$$\mathrm{ran}(f) = \{\omega' \in \Omega \mid \exists \omega \in \Omega : \ f(\omega) = \omega'\}. \tag{1.5}$$

Proposition 1.2. *The set of partial functions $E(\Omega)$ is uncountable.*

Proof. For each real number $x = 0.x_0x_1x_2\ldots$ from the real-valued interval $[0,1]$, define the function $f_x \in E(\Omega)$ such that $f_x(\omega) = \omega'$, where for all $n \geq 0$,

$$\omega'_n = \begin{cases} x_n \text{ if } \omega_n \neq 0 \\ 0 \quad \text{otherwise.} \end{cases} \tag{1.6}$$

For different real numbers $x = 0.x_0x_1x_2\ldots$ and $y = 0.y_0y_1y_2\ldots$, the function f_x and f_y are distinct. To see this, note that if $x_n \neq y_n$ for some $n \geq 0$, then for the state ω which is 1 at register R_n and 0 elsewhere, we have $f_x(\omega) \neq f_y(\omega)$. Since the interval $[0,1]$ is uncountable, the result follows. □

Two partial functions $f, g \in E(\Omega)$ are *equal*, written $f = g$, if they have the same domain, i.e., $\mathrm{dom}(f) = \mathrm{dom}(g)$, and for all arguments in the (common) domain, they coincide, i.e., for all $\omega \in \mathrm{dom}(f)$, $f(\omega) = g(\omega)$. A partial function $f \in E(\Omega)$ is called *total* if $\mathrm{dom}(f) = \Omega$. So a total function is a function in the usual sense.

Example 1.3.

- The *increment function* $a_k \in E(\Omega)$ with respect to the k-th register is given by the assignment $a_k : \omega \mapsto \omega'$, where

$$\omega'_n = \begin{cases} \omega_n & \text{if } n \neq k, \\ \omega_k + 1 & \text{otherwise.} \end{cases} \tag{1.7}$$

- The *decrement function* $s_k \in E(\Omega)$ w.r.t. the k-th register is defined as $s_k : \omega \mapsto \omega'$, where

$$\omega'_n = \begin{cases} \omega_n & \text{if } n \neq k, \\ \omega_k \dot{-} 1 & \text{otherwise.} \end{cases} \tag{1.8}$$

The dyadic operator $\dot{-}$ on $\mathbb{N}_0$ denotes the *asymmetric difference* given as

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y, \\ 0 & \text{otherwise.} \end{cases} \tag{1.9}$$

Both functions a_k and s_k are total. ♢

The *graph* of a partial function $f \in E(\Omega)$ is given by the relation

$$R_f = \{(\omega, f(\omega)) \mid \omega \in \operatorname{dom}(f)\}. \tag{1.10}$$

For instance, the increment function a_k, $k \in \mathbb{N}_0$, gives

$$R_{a_k} = \{(\omega, \omega') \mid \omega \in \Omega, \omega' = (w_0, \ldots, w_{k-1}, w_k + 1, w_{k+1}, \ldots)\}.$$

It is clear that two partial functions $f, g \in E(\Omega)$ are equal if and only if the corresponding graphs R_f and R_g are equal as sets.

The *composition* of two partial functions $f, g \in E(\Omega)$ is a partial function, denoted by $g \circ f$, defined as

$$(g \circ f)(\omega) = g(f(\omega)), \tag{1.11}$$

where ω belongs to the domain of $g \circ f$ given by

$$\operatorname{dom}(g \circ f) = \{\omega \in \Omega \mid \omega \in \operatorname{dom}(f) \wedge f(\omega) \in \operatorname{dom}(g)\}. \tag{1.12}$$

For instance, the increment functions a_k and a_l, $k, l \in \mathbb{N}_0$ with $k < l$, give

$$(a_k \circ a_l)(\omega) = (w_0, \ldots, w_k + 1, \ldots, w_l + 1, \ldots)\}.$$

It is clear that if f and g are total functions in $E(\Omega)$, the composition $g \circ f$ is also a total function.

Proposition 1.4. *The set $E(\Omega)$ together with the dyadic operation of composition is a semigroup.*

Proof. It is clear that the composition of partial functions is an associative operation. □

In this way, the set $E(\Omega)$ is called the *semigroup of transformations* of Ω.

The *powers* of a partial function $f \in E(\Omega)$ are inductively defined as follows:

$$f^0 = \mathrm{id}_\Omega, \quad \text{and} \quad f^{n+1} = f \circ f^n, \ n \in \mathbb{N}_0. \tag{1.13}$$

In particular, $f^1 = f \circ \mathrm{id}_\Omega = f$. For instance, the increment function a_k, $k \in \mathbb{N}_0$, gives

$$a_k^n(\omega) = (w_0, \ldots, w_{k-1}, w_k + n, w_{k+1}, \ldots), \quad n \in \mathbb{N}_0.$$

Consider for each function $f \in E(\Omega)$ and each (initial) state $\omega \in \Omega$ the following sequence of consecutive states:

$$\omega = f^0(\omega), f^1(\omega), f^2(\omega), \ldots. \tag{1.14}$$

This sequence is finite if $\omega \not\in \operatorname{dom}(f^j)$ for some $j \in \mathbb{N}_0$. For this, put

$$\lambda(f, \omega) = \begin{cases} \min\{j \in \mathbb{N}_0 \mid \omega \not\in \operatorname{dom}(f^j)\} & \text{if the minimum exists,} \\ \infty & \text{otherwise.} \end{cases} \tag{1.15}$$

Note that if f is a total function, then $\lambda(f, \omega) = \infty$ for any state ω. For instance, this holds for both the increment and decrement functions a_k and s_k, $k \in \mathbb{N}_0$.

Algorithm 1.1 Computation of iteration f^{*n}.

```
Require: $\omega \in \Omega$
  while $\omega_n > 0$ do
    $w \leftarrow f(\omega)$
  end while
```

The *iteration* of a function $f \in E(\Omega)$ with respect to the n-th register is the partial function $f^{*n} \in E(\Omega)$ defined as

$$f^{*n}(\omega) = f^k(\omega), \tag{1.16}$$

if there is an integer $k \geq 0$ with $k < \lambda(f, \omega)$ such that the content of the n-th register is 0 in $f^k(\omega)$, but non-zero in $f^j(\omega)$ for each $0 \leq j < k$. If no such integer k exists, the value of $f^{*n}(\omega)$ is taken to be undefined, written $\uparrow$. The computation of f^{*n} can be carried out by a `while` loop (Alg. 1.1).

Examples 1.5.

- Let $f = a_k$. Then

$$f^{*k}(\omega) = \begin{cases} f^0(w) = w \text{ if } \omega_k = 0, \\ \uparrow \qquad\qquad\quad \text{otherwise.} \end{cases}$$

- Let $f = s_k$. Then

$$f^{*k}(\omega) = \begin{cases} f^l(w) \text{ if } w_k = l, \\ \uparrow \qquad\quad \text{otherwise.} \end{cases}$$

♢

1.2 Syntax of URM Programs

The set of all (decimal) numbers over the alphabet of digits $\Sigma_{10} = \{0, 1, \ldots, 9\}$ is defined as

$$Z = (\Sigma_{10} \setminus \{0\})\Sigma_{10}^+ \cup \Sigma_{10}. \tag{1.17}$$

That is, a number is either a digit or a non-empty word of digits that does not begin with 0.

The URM programs are words over the alphabet

$$\Sigma_{\mathrm{URM}} = \{A, S, (,), ;\} \cup Z. \tag{1.18}$$

Define the set of *URM programs* $\mathcal{P}_{\mathrm{URM}}$ inductively as follows:

1. $A\sigma \in \mathcal{P}_{\mathrm{URM}}$ for each $\sigma \in Z$.
2. $S\sigma \in \mathcal{P}_{\mathrm{URM}}$ for each $\sigma \in Z$.
3. If $P \in \mathcal{P}_{\mathrm{URM}}$ and $\sigma \in Z$, then $(P)\sigma \in \mathcal{P}_{\mathrm{URM}}$.
4. If $P, Q \in \mathcal{P}_{\mathrm{URM}}$, then $P; Q \in \mathcal{P}_{\mathrm{URM}}$.

The programs $A\sigma$ and $S\sigma$ are *atomic*, the program $(P)\sigma$ is the *iteration* of the program P with respect to the register R_σ, and the program $P;Q$ is the *composition* of the programs P and Q. For each program $P \in \mathcal{P}_{\mathrm{URM}}$ and each integer $n \geq 0$, define the n-fold composition of P as

$$P^n = P; P; \ldots; P \quad (n \text{ times}). \tag{1.19}$$

The atomic programs and the iterations are called *blocks*. The set of blocks in $\mathcal{P}_{\mathrm{URM}}$ is denoted by $\mathcal{B}$.

Lemma 1.6. *For each program $P \in \mathcal{P}_{\mathrm{URM}}$, there are uniquely determined blocks $P_1, \ldots, P_k \in \mathcal{B}$ such that*

$$P = P_1; P_2; \ldots; P_k.$$

The separation symbol ";" can be removed although it increases eventually readability. In this way, we obtain the following result.

Proposition 1.7. *The set $\mathcal{P}_{\mathrm{URM}}$ together with the operation of concatenation is a subsemigroup of $\Sigma_{\mathrm{URM}}^{+}$ which is freely generated by the set of blocks $\mathcal{B}$.*

Example 1.8. The URM program $P = (A3; A4; S1)1; ((A1; S3)3; S2; (A0; A3; S4)4; (A4; S0)0)2$ consists of the blocks $P_1 = (A3; A4; S1)1$ and $P_2 = ((A1; S3)3; S2; (A0; A3; S4)4; (A4; S0)0)2$. ♢

1.3 Semantics of URM Programs

The URM programs can be interpreted by the semigroup of transformations $E(\Omega)$. The *semantics* of URM programs is a mapping $|\cdot| : \mathcal{P}_{\mathrm{URM}} \to E(\Omega)$ defined inductively as follows:

1. $|A\sigma| = a_\sigma$ for each $\sigma \in Z$.
2. $|S\sigma| = s_\sigma$ for each $\sigma \in Z$.
3. If $P \in \mathcal{P}_{\mathrm{URM}}$ and $\sigma \in Z$, then $|(P)\sigma| = |P|^{*\sigma}$.
4. If $P, Q \in \mathcal{P}_{\mathrm{URM}}$, then $|P;Q| = |Q| \circ |P|$.

The semantics of blocks is defined by the first three items, and the last item indicates that the mapping $|\cdot|$ is a morphism of semigroups.

Proposition 1.9. *For each mapping $\psi : \mathcal{B} \to E(\Omega)$, there is a unique semigroup homomorphism $\phi : \mathcal{P}_{\mathrm{URM}} \to E(\Omega)$ making the following diagram commutative:*

$$\begin{tikzcd}
\mathcal{P}_{\mathrm{URM}} \arrow[r, "\phi"'] & E(\Omega) \\
\mathcal{B} \arrow[u, "\mathrm{id}"'] \arrow[ur, "\psi"'] &
\end{tikzcd}$$

Proof. Given a mapping $\psi : \mathcal{B} \to E(\Omega)$. Since each URM program P is a composition of blocks, there are elements $B_0, \ldots, B_n$ of $\mathcal{B}$ such that $P = B_0; \ldots; B_n$. Define $\phi(P) = \psi(B_0); \ldots; \psi(B_n)$. This gives a semigroup homomorphism $\phi : \mathcal{P}_{\mathrm{URM}} \to E(\Omega)$ with the required property.

On the other hand, if $\phi' : \mathcal{P}_{\mathrm{URM}} \to E(\Omega)$ is a semigroup homomorphism with the property $\psi(B) = \phi'(B)$ for each block B. Then $\phi = \phi'$, since all URM programs are sequences of blocks. □

This algebraic statement asserts that the semantics on blocks can be uniquely extended to the full set of URM programs.

1.4 URM Computable Functions

A partial function $f \in E(\Omega)$ is *URM computable* if there is an URM program P such that $|P| = f$. Note that the class $\mathcal{P}_{\mathrm{URM}}$ is countable, while the set $E(\Omega)$ is not. It follows that there are partial functions in $E(\Omega)$ that are not URM computable. In the following, let $\mathcal{F}_{\mathrm{URM}}$ denote the class of all partial functions that are URM computable, and let $\mathcal{T}_{\mathrm{URM}}$ depict the class of all total functions which are URM computable. Then we have $\mathcal{T}_{\mathrm{URM}} \subset \mathcal{F}_{\mathrm{URM}}$.

Functions like addition or multiplication of two natural numbers are URM computable. In general, the calculation of an URM-computable function $f : \mathbb{N}_0^k \to \mathbb{N}_0^m$ requires to load the registers with initial values and to read out the result. For this, define the total functions

$$\alpha_k : \mathbb{N}_0^k \to \Omega : (x_1, \ldots, x_k) \mapsto (0, x_1, \ldots, x_k, 0, 0, \ldots) \tag{1.20}$$

and

$$\beta_m : \Omega \to \mathbb{N}_0^m : (\omega_0, \omega_1, \omega_2, \ldots,) \mapsto (\omega_1, \omega_2, \ldots, \omega_m). \tag{1.21}$$

Given an URM program P and integers $k, m \in \mathbb{N}_0$, define the partial function $\|P\|_{k,m} : \mathbb{N}_0^k \to \mathbb{N}_0^m$ by the composition

$$\|P\|_{k,m} = \beta_m \circ |P| \circ \alpha_k. \tag{1.22}$$

Note that the k-ary function $\|P\|_{k,m}$ is computed by loading the registers with an argument $x \in \mathbb{N}_0^k$, calculating the program P on the initial state $\alpha_k(x)$ and then reading out the result using β_m.

A (partial) function $f : \mathbb{N}_0^k \to \mathbb{N}_0^m$ is called *URM computable* if there is an URM program P such that

$$f = \|P\|_{k,m}. \tag{1.23}$$

Examples 1.10.

- The program $P = (A1)1$ computes the monadic function $f : \mathbb{N}_0 \to \mathbb{N}_0$ given by

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

- The program $P = A1; (A1)1$ provides the monadic function $f_\uparrow : \mathbb{N}_0 \to \mathbb{N}_0$ which is nowhere defined; i.e., $f_\uparrow(x) = \uparrow$ for all $x \in \mathbb{N}_0$.
- The program $P = (S1)1$ calculates the monadic zero-function $f : \mathbb{N}_0 \to \mathbb{N}_0$, i.e., $f(x) = 0$ for all $x \in \mathbb{N}_0$.

Examples 1.11.

- The addition of two natural numbers is URM computable. To see this, consider the URM program

$$P_+ = (A1; S2)2. \tag{1.24}$$

This program transforms the initial state (ω_n) into the state $(\omega_0, \omega_1 + \omega_2, 0, \omega_3, \omega_4, \ldots)$ and thus realizes the function

$$\|P_+\|_{2,1}(x, y) = x + y, \quad x, y \in \mathbb{N}_0. \tag{1.25}$$

- The multiplication of two natural number is URM computable. For this, take the URM program
$$P_\cdot = (A3;A4;S1)1;((A1;S3)3;S2;(A0;A3;S4)4;(A4;S0)0)2. \tag{1.26}$$
The first block $(A3;A4;S1)1$ transforms the initial state $(0,x,y,0,0,\ldots)$ into $(0,0,y,x,x,0,0,\ldots)$. Then the subprogram $(A1;S3)3;S_2;(A0;A3;S4)4;(A3;S0)0$ is carried out y times adding the content of R_3 to that of R_1 and copying the content of R_4 to R_3. This iteration provides the state $(0,xy,0,x,x,0,0,\ldots)$. It follows that
$$\|P_\cdot\|_{2,1}(x,y) = xy, \quad x,y \in \mathbb{N}_0. \tag{1.27}$$
- The asymmetric difference is URM computable. For this, pick the URM program
$$P_{\dot{-}} = (S1;S2)2. \tag{1.28}$$
This program transforms the initial state (ω_n) into the state $(\omega_0, \omega_1 \dot{-} \omega_2, 0, \omega_3, \ldots)$ and thus yields the URM computable function
$$\|P_{\dot{-}}\|_{2,1}(x,y) = x \dot{-} y, \quad x,y \in \mathbb{N}_0. \tag{1.29}$$
- Consider the *sign function* sgn : $\mathbb{N}_0 \to \mathbb{N}_0$ given by $\text{sgn}(x) = 1$ if $x > 0$ and $\text{sgn}(x) = 0$ if $x = 0$. This function is URM computable since it is calculated by the URM program
$$P_{\text{sgn}} = (A2;S1)1;(A1;(S2)2)2. \tag{1.30}$$
From the sign function, it is easy to devise an URM program for the *cosign function* csg : $\mathbb{N}_0 \to \mathbb{N}_0$ given by $\text{csg}(x) = 0$ if $x > 0$ and $\text{sgn}(x) = 1$ if $x = 0$. We have $\text{csg}(x) = 1 - \text{sgn}(x)$ for all $x \in \mathbb{N}_0$. ♢

Note that URM programs are *invariant of translation* in the sense that if an URM program P manipulates the registers $R_{i_1}, \ldots, R_{i_k}$, then for each integer $n \geq 0$ there is an URM program that manipulates the registers $R_{i_1+n}, \ldots, R_{i_k+n}$. This program will be denoted by $(P)[+n]$. For instance, if $P = (A1;S2)2$, then $(P)[+5] = (A6;S7)7$.

Let $f : \mathbb{N}_0^k \to \mathbb{N}_0$ be an URM computable function. An URM program P with $\|P\|_{k,1} = f$ is called *normal* if for all inputs $(x_1, \ldots, x_k) \in \mathbb{N}_0^k$,
$$(P \circ \alpha_k)(x_1,\ldots,x_k) = \begin{cases} (0, f(x_1,\ldots,x_k), 0, 0, \ldots) & \text{if } (x_1,\ldots,x_k) \in \text{dom}(f), \\ \uparrow & \text{otherwise.} \end{cases} \tag{1.31}$$
A normal URM-program computes a monadic function in such a way that whenever the computation ends the register R_1 contains the result while all other registers are set to zero.

Proposition 1.12. *For each URM-computable function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ there is a normal URM-program P such that $\|P\|_{k,1} = f$.*

Proof. Let Q be an URM-program such that $\|Q\|_{k,1} = f$. Suppose R_σ is the largest register that contains a non-zero value in the final state of computation. Then the corresponding normal URM-program is given by
$$P = Q;(S0)0;(S2)2;\ldots;(S\sigma)\sigma. \tag{1.32}$$
Here the block $(Si)i$ sets the value of the i-th register to zero. □

Finally, we introduce two programs that are useful for the transport or distribution of register values. The first function *reloads* the content of register R_i, $i > 0$, into $k \geq 1$ registers $R_{j_1}, \ldots, R_{j_k}$ different from R_i and thereby deleting the content of R_i. This is achieved by the URM program

$$R(i; j_1, j_2, \ldots, j_k) = (Aj_1; Aj_2; \ldots; Aj_k; Si)i. \tag{1.33}$$

Indeed, the program transforms the initial state ω into the state ω' where

$$\omega'_n = \begin{cases} \omega_n + \omega_i & \text{if } n \in \{j_1, j_2, \ldots, j_k\}, \\ 0 & \text{if } n = i, \\ \omega_n & \text{otherwise.} \end{cases} \tag{1.34}$$

The second function *copies* the content of register R_i, $i > 0$, into $k \geq 1$ registers $R_{j_1}, \ldots, R_{j_k}$ different from R_i where the content of register R_i is retained. Here the register R_0 is used for distributing the value of R_i. This is achieved by the URM program

$$C(i; j_1, j_2, \ldots, j_k) = R(i; 0, j_1, j_2, \ldots, j_k); R(0; i). \tag{1.35}$$

In fact, the program transforms the initial state ω into the state ω' where

$$\omega'_n = \begin{cases} \omega_n + \omega_i & \text{if } n \in \{j_1, j_2, \ldots, j_k\}, \\ \omega_n + \omega_0 & \text{if } n = i, \\ 0 & \text{if } n = 0, \\ \omega_n & \text{otherwise.} \end{cases} \tag{1.36}$$

2

Primitive Recursive Functions

The primitive recursive functions form an important building block on the way to a full formalization of computability. They are formally defined using composition and primitive recursion as central operations. Most of the functions studied in arithmetics are primitive recursive such as the basic operations of addition and multiplication. Indeed, it is difficult to devise a function that is total and computable but not primitive recursive. From the programming point of view, the primitive recursive functions can be implemented using `do`-loops only.

2.1 Peano Structures

We will use Peano structures to define the concept of primitive recursion, which is central for the class of primitive recursive functions.

A *semi-Peano structure* is a triple $\mathcal{A} = (A, \alpha, a)$ consisting of a non-empty set A, a monadic operation $\alpha : A \to A$, and an element $a \in A$.

Example 2.1.

- For each natural number x_0, the triple $(\mathbb{N}_0, \nu, x_0)$ is a semi-Peano structure, where $\nu : \mathbb{N}_0 \to \mathbb{N}_0 : n \mapsto n + 1$ denotes the successor function.
- For each alphabet Σ and each word $w_0 \in \Sigma^*$, the triple $(\Sigma^*, \circ_a, w_0)$ is a semi-Peano structure, where $\circ_a$ denotes the concatenation of a word w with a symbol $a \in \Sigma$, i.e., $\circ_a(w) = wa$ for each $w \in \Sigma^*$. ♢

Let $\mathcal{A} = (A, \alpha, a)$ and $\mathcal{B} = (B, \beta, b)$ be two semi-Peano structures. A mapping $\phi : A \to B$ is called a *morphism*, written $\phi : \mathcal{A} \to \mathcal{B}$, if ϕ correspondingly assigns the distinguished elements, i.e., $\phi(a) = b$, and commutes with the monadic operations, i.e., $\beta \circ \phi = \phi \circ \alpha$. That is, the following diagram is commutative:

$$\begin{tikzcd}
A \arrow[r, "\alpha"] \arrow[d, "\phi"] & A \arrow[d, "\phi"] \\
B \arrow[r, "\beta"] & B
\end{tikzcd}$$

Example 2.2. Consider the semi-Peano structures $(\mathbb{N}_0, \nu, 0)$ and $(\Sigma^*, \circ_a, \epsilon)$, where ϵ denotes the empty word. The mapping $\phi : \Sigma^* \to \mathbb{N}_0$ which assigns to each word $w \in \Sigma^*$ its length $|w|$ is a morphism, since $\phi(\epsilon) = 0$ and $\phi(\circ_a(w)) = \phi(wa) = |wa| = |w| + |a| = |w| + 1 = \nu(\phi(w))$ for each $w \in \Sigma^*$. $\diamondsuit$

A *Peano structure* is a semi-Peano structure $\mathcal{A} = (A, \alpha, a)$ with the following properties:

- α is injective,
- $a \not\in \mathrm{ran}(\alpha)$, and
- A fulfills the *induction axiom*, i.e., if $T \subseteq A$ such that $a \in T$ and $\alpha(x) \in T$ whenever $x \in T$, then $T = A$.

Let $\nu : \mathbb{N}_0 \to \mathbb{N}_0 : n \mapsto n + 1$ be the *successor function.* The Peano structure given by the triple $(\mathbb{N}_0, \nu, 0)$ corresponds to the axioms postulated by the Italian mathematician Guiseppe Peano (1858-1932).

Lemma 2.3. *If $\mathcal{A} = (A, \alpha, a)$ is a Peano structure, then $A = \{\alpha^n(a) \mid n \in \mathbb{N}_0\}$.*

Proof. Let $T = \{\alpha^n(a) \mid n \in \mathbb{N}_0\}$. Then $a = \alpha^0(a) \in T$, and for each $\alpha^n(a) \in T$, $\alpha(\alpha^n(a)) = \alpha^{n+1}(a) \in T$. Hence by the induction axiom, $T = A$. □

Lemma 2.4. *If $\mathcal{A} = (A, \alpha, a)$ is a Peano structure, then for all $m, n \in \mathbb{N}_0$, $m \neq n$, we have $\alpha^m(a) \neq \alpha^n(a)$.*

Proof. Define T as the set of all elements $\alpha^m(a)$ such that $\alpha^n(a) \neq \alpha^m(a)$ for all $n \in \mathbb{N}_0$ with $n > m$.

First, suppose that $\alpha^n(a) = \alpha^0(a) = a$ for some $n > 0$. Then $a \in \mathrm{ran}(\alpha)$ contradicting the definition. It follows that $a \in T$.

Second, let $x \in T$; that is, $x = \alpha^m(a)$ for some $m \geq 0$. Suppose that $\alpha(x) = \alpha^{m+1}(x) \not\in T$. Then there is a number $n > m$ such that $\alpha^{m+1}(a) = \alpha^{n+1}(a)$. But α is injective and so $\alpha^m(a) = \alpha^n(a)$ contradicting the hypothesis. It follows that $\alpha(x) \in T$.

Thus the induction axiom implies that $T = A$ as required. □

These assertions lead to an important assertion for Peano structures.

Proposition 2.5 (Fundamental Lemma). *If $\mathcal{A} = (A, \alpha, a)$ is a Peano structure and $\mathcal{B} = (B, \beta, b)$ is a semi-Peano structure, then there is a unique morphism $\phi : \mathcal{A} \to \mathcal{B}$.*

Proof. We have $A = \{\alpha^n(a) \mid n \in \mathbb{N}_0\}$. To prove existence, define $\phi(\alpha^n(a)) = \beta^n(b)$ for all $n \in \mathbb{N}_0$. The above assertions imply that the mapping ϕ is well-defined. Moreover, $\phi(a) = \phi(\alpha^0(a)) = \beta^0(b) = b$. Finally, let $x \in A$. Then $x = \alpha^m(a)$ for some $m \in \mathbb{N}_0$ and so $(\phi \circ \alpha)(x) = \phi(\alpha^{m+1}(a)) = \beta^{m+1}(b) = \beta(\beta^m(b)) = \beta(\phi(\alpha^m(a))) = (\beta \circ \phi)(x)$. Hence ϕ is a morphism.

To show uniqueness, suppose there is another morphism $\psi : \mathcal{A} \to \mathcal{B}$. Define $T = \{x \in A \mid \phi(x) = \psi(x)\}$. First, $\phi(a) = b = \psi(a)$ and so $a \in T$. Second, let $x \in T$. Then $\phi(\alpha(x)) = (\phi \circ \alpha)(x) = (\beta \circ \phi)(x) = (\beta \circ \psi)(x) = (\psi \circ \alpha)(x) = \psi(\alpha(x))$ and so $\alpha(x) \in T$. Hence by the induction axiom, $T = A$ and so $\phi = \psi$. □

The fundamental lemma provides immediately a result of Richard Dedekind (1831-1916).

Corollary 2.6. *Each Peano structure is isomorphic to* $(\mathbb{N}_0, \nu, 0)$.

Proof. We have already seen that $(\mathbb{N}_0, \nu, 0)$ is a Peano structure. Suppose there are Peano structures $\mathcal{A} = (A, \alpha, a)$ and $\mathcal{B} = (B, \beta, b)$. It is sufficient to show that there are morphisms $\phi : \mathcal{A} \to \mathcal{B}$ and $\psi : \mathcal{B} \to \mathcal{A}$ such that $\psi \circ \phi = \mathrm{id}_A$ and $\phi \circ \psi = \mathrm{id}_B$. For this, note that the composition of morphisms is also a morphism. Thus $\psi \circ \phi : \mathcal{A} \to \mathcal{A}$ is a morphism. On the other hand, the identity mapping $\mathrm{id}_A : \mathcal{A} \to \mathcal{A} : x \mapsto x$ is a morphism. Hence, by the fundamental lemma, $\psi \circ \phi = \mathrm{id}_A$. Similarly, it follows that $\phi \circ \psi = \mathrm{id}_B$. □

The fundamental lemma can be applied to the basic Peano structure $(\mathbb{N}_0, \nu, 0)$ in order to define recursively new functions.

Proposition 2.7. *If (A, α, a) is a semi-Peano structure, there is a unique total function $g : \mathbb{N}_0 \to A$ such that*

1. *$g(0) = a$,*
2. *$g(y + 1) = \alpha(g(y))$ for all $y \in \mathbb{N}_0$.*

Proof. By the fundamental lemma, there is a unique morphism $g : \mathbb{N}_0 \to A$ such that $g(0) = a$ and $g(y + 1) = g \circ \nu(y) = \alpha \circ g(y) = \alpha(g(y))$ for each $y \in \mathbb{N}_0$. □

Example 2.8. There is a unique total function $f_+ : \mathbb{N}_0^2 \to \mathbb{N}_0$ such that

1. $f_+(x, 0) = x$ for all $x \in \mathbb{N}_0$,
2. $f_+(x, y + 1) = f_+(x, y) + 1$ for all $x, y \in \mathbb{N}_0$.

To see this, consider the semi-Peano structure $(\mathbb{N}_0, \nu, x)$ for a fixed number $x \in \mathbb{N}_0$. By the fundamental lemma, there is a unique total function $f_x : \mathbb{N}_0 \to \mathbb{N}_0$ such that

1. $f_x(0) = x$,
2. $f_x(y + 1) = f_x \circ \nu(y) = \nu \circ f_x(y) = f_x(y) + 1$ for all $y \in \mathbb{N}_0$.

The function f_+ is obtained by putting $f_+(x, y) = f_x(y)$ for all $x, y \in \mathbb{N}_0$. By induction, it follows that $f_+(x, y) = x + y$ for all $x, y \in \mathbb{N}_0$. This provides a recursive definition of the addition of two numbers. ♢

Proposition 2.9. *If $g : \mathbb{N}_0^k \to \mathbb{N}_0$ and $h : \mathbb{N}_0^{k+2} \to \mathbb{N}_0$ are total functions, there is a unique total function $f : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ such that*

$$f(\boldsymbol{x}, 0) = g(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{N}_0^k, \tag{2.1}$$

and

$$f(\boldsymbol{x}, y + 1) = h(\boldsymbol{x}, y, f(\boldsymbol{x}, y)), \quad \boldsymbol{x} \in \mathbb{N}_0^k, y \in \mathbb{N}_0. \tag{2.2}$$

Proof. For each $\boldsymbol{x} \in \mathbb{N}_0^k$, consider the semi-Peano structure $(\mathbb{N}_0^2, \alpha_{\boldsymbol{x}}, a_{\boldsymbol{x}})$, where $a_{\boldsymbol{x}} = (0, g(\boldsymbol{x}))$ and $\alpha_{\boldsymbol{x}} : \mathbb{N}_0^2 \to \mathbb{N}_0^2 : (y, z) \mapsto (y + 1, h(\boldsymbol{x}, y, z))$. By Proposition 2.7, there is a unique total function $f_{\boldsymbol{x}} : \mathbb{N}_0 \to \mathbb{N}_0^2$ such that

1. $f_{\boldsymbol{x}}(0) = (0, g(x))$,
2. $f_{\boldsymbol{x}}(y + 1) = (f_{\boldsymbol{x}} \circ \nu)(y) = (\alpha_{\boldsymbol{x}} \circ f_{\boldsymbol{x}})(y) = \alpha_{\boldsymbol{x}}(y, f_{\boldsymbol{x}}(y)) = (y + 1, h(\boldsymbol{x}, y, f_{\boldsymbol{x}}(y)))$ for all $y \in \mathbb{N}_0$.

The projection mapping $\pi_2^{(2)} : \mathbb{N}_0^2 \to \mathbb{N}_0 : (y, z) \mapsto z$ leads to the desired function $f(\boldsymbol{x}, y) = \pi_2^{(2)} \circ f_{\boldsymbol{x}}(y)$, where $\boldsymbol{x} \in \mathbb{N}_0^k$ and $y \in \mathbb{N}_0$. □

The function f given in (2.1) and (2.2) is said to be defined by *primitive recursion* of the functions g and h. The above example shows that the addition of two numbers can be defined by primitive recursion.

2.2 Primitive Recursive Functions

The class of primitive recursive functions is inductively defined. For this, the *basic functions* are the following:

1. The *0-ary constant function* $c_0^{(0)} : \to \mathbb{N}_0 : \mapsto 0$.
2. The *monadic constant function* $c_0^{(1)} : \mathbb{N}_0 \to \mathbb{N}_0 : x \mapsto 0$.
3. The *successor function* $\nu : \mathbb{N}_0 \to \mathbb{N}_0 : x \mapsto x + 1$.
4. The *projection functions* $\pi_k^{(n)} : \mathbb{N}_0^n \to \mathbb{N}_0 : (x_1, \ldots, x_n) \mapsto x_k$, where $n \geq 1$ and $1 \leq k \leq n$.

Using these functions, more complex primitive recursive functions can be introduced.

1. If g is a k-ary total function and $h_1, \ldots, h_k$ are n-ary total functions, the *composition* of g along $(h_1, \ldots, h_k)$ is an n-ary function $f = g(h_1, \ldots, h_k)$ defined as
$$f(\boldsymbol{x}) = g(h_1(\boldsymbol{x}), \ldots, h_k(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathbb{N}_0^n. \tag{2.3}$$
2. If g is an n-ary total function and h is an $n+2$-ary total function, the *primitive recursion* of g along h is an $n+1$-ary function f given as
$$f(\boldsymbol{x}, 0) = g(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathbb{N}_0^n, \tag{2.4}$$
and
$$f(\boldsymbol{x}, y+1) = h(\boldsymbol{x}, y, f(\boldsymbol{x}, y)), \quad \boldsymbol{x} \in \mathbb{N}_0^n, y \in \mathbb{N}_0. \tag{2.5}$$

This function is also denoted by $f = \mathrm{pr}(g, h)$.

The class of *primitive recursive functions* is given by the basic functions and those obtained from the basic functions by applying composition and primitive recursion a finite number of times. These functions were first studied by Richard Dedekind.

Proposition 2.10. *Each primitive recursive function is total.*

Proof. The basic functions are total. Let $f = g(h_1, \ldots, h_k)$ be the composition of g along $(h_1, \ldots, h_k)$. By induction, it can be assumed that the functions $g, h_1, \ldots, h_k$ are total. Then the function f is also total.

Let $f = \mathrm{pr}(g, h)$ be the primitive recursion of g along h. By induction, suppose that the functions g and h are total. Then the function f is total, too. □

Examples 2.11. The dyadic functions of addition and multiplication are primitive recursive.

1. The function $f_+ : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x, y) \mapsto x + y$ obeys the following scheme of primitive recursion:
   a) $f_+(x, 0) = x = \mathrm{id}_{\mathbb{N}_0}(x)$ for all $x \in \mathbb{N}_0$,
   b) $f_+(x, y+1) = f_+(x, y) + 1 = (\nu \circ \pi_3^{(3)})(x, y, f_+(x, y))$ for all $x, y \in \mathbb{N}_0$.
2. Define the function $f_. : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x, y) \mapsto xy$ inductively as follows:
   a) $f_.(x, 0) = 0$ for all $x \in \mathbb{N}_0$,
   b) $f_.(x, y+1) = f_.(x, y) + x = f_+(x, f_.(x, y))$ for all $x, y \in \mathbb{N}_0$.

   This leads to the following scheme of primitive recursion:
   a) $f_.(x, 0) = c_0^{(1)}(x)$ for all $x \in \mathbb{N}_0$,

b) $f_{\cdot}(x, y+1) = f_{+}(\pi_1^{(3)}, \pi_3^{(3)})(x, y, f_{\cdot}(x, y))$ for all $x, y \in \mathbb{N}_0$. ♢

Theorem 2.12. *Each primitive recursive function is URM computable.*

Proof. First, claim that the basic functions are URM computable.

1. 0-ary constant function: The URM program
$$P_0^{(0)} = A0; S0$$
gives $\|P_0^{(0)}\|_{0,1} = c_0^{(0)}$.
2. Unary constant function: The URM program
$$P_0^{(1)} = (S1)1$$
provides $\|P_0^{(1)}\|_{1,1} = c_0^{(1)}$.
3. Successor function: The URM program
$$P_{+1} = A1$$
yields $\|P_{+1}\|_{1,1} = \nu$.
4. Projection function $\pi_k^{(n)}$ with $n \geq 1$ and $1 \leq k \leq n$: The URM program
$$P_{p(n,k)} = R(k;0); (S1)1; R(0;1)$$
shows that $\|P_{n,k}\|_{n,1} = \pi_k^{(n)}$.

Second, consider the composition $f = g(h_1, \ldots, h_k)$. By induction, assume that there are normal URM programs P_g and $P_{h_1}, \ldots, P_{h_k}$ such that $\|P_g\|_{k,1} = g$ and $\|P_{h_i}\|_{n,1} = h_i$, $1 \leq i \leq k$. A normal URM program for the composite function f can be obtained as follows: For each $1 \leq i \leq k$,

- copy the values $x_1, \ldots, x_n$ into the registers $R_{n+k+2}, \ldots, R_{2n+k+1}$,
- compute the value $h_i(x_1, \ldots, x_n)$ by using the registers $R_{n+k+2}, \ldots, R_{2n+k+j}$, where $j \geq 1$,
- store the result $h_i(x_1, \ldots, x_n)$ in R_{n+i}.

Formally, this computation is carried out as follows:
$$Q_i = C(1; n+k+2); \ldots; C(n, 2n+k+1); (P_{h_i})[+n+k+1]; R(n+k+2; n+i) \tag{2.6}$$

Afterwards, the values in R_{n+i} are copied into R_i, $1 \leq i \leq k$, and the function $g(h_1(\boldsymbol{x}), \ldots, h_k(\boldsymbol{x}))$ is computed. Formally, the overall computation is achieved by the URM program
$$P_f = Q_1; \ldots; Q_k; (S1)1; \ldots; (Sn)n; R(n+1;1); \ldots; R(n+k;k); P_g \tag{2.7}$$
giving $\|P_f\|_{n,1} = f$.

Third, consider the primitive recursion $f = \mathrm{pr}(g, h)$. By induction, assume that there are normal URM programs P_g and P_h such that $\|P_g\|_{n,1} = g$ and $\|P_h\|_{n+2,1} = h$. As usual, the registers $R_1, \ldots, R_{n+1}$ contain the input values for the computation of $f(\boldsymbol{x}, y)$.

Note that the computation of the function value $f(\boldsymbol{x}, y)$ can be accomplished in $y+1$ steps:

- $f(\boldsymbol{x}, 0) = g(\boldsymbol{x})$, and

- for each $1 \leq i \leq y$, $f(\boldsymbol{x}, i) = h(\boldsymbol{x}, i-1, f(\boldsymbol{x}, i-1))$.

For this, the register R_{n+2} is used as a counter and the URM programs $(P_g)[+n+3]$ and $(P_h)[+n+3]$ make only use of the registers R_{n+3+j}, where $j \geq 0$. Formally, the overall computation is given as follows:

$$\begin{aligned} P_f = {} & R(n+1; n+2); \\ & C(1; n+4); \ldots; C(n, 2n+3); \\ & (P_g)[+n+3]; \\ & (R(n+4; 2n+5); C(1; n+4); \ldots; C(n+1; 2n+4); (P_h)[+n+3]; An+1; Sn+2)n+2; \\ & (S1)1; \ldots; (Sn+1)n+1; \\ & R(n+4; 1). \end{aligned} \tag{2.8}$$

First, the input value y is stored in R_{n+2} to serve as a counter, and the input values $x_1, \ldots, x_n$ are copied into $R_{n+4}, \ldots, R_{2n+3}$, respectively. Then $f(\boldsymbol{x}, 0) = g(\boldsymbol{x})$ is calculated. Afterwards, the following iteration is performed while the value of R_{n+2} is non-zero: Copy $x_1, \ldots, x_n$ into $R_{n+4}, \ldots, R_{2n+3}$, respectively, copy the value of R_{n+1} into R_{2n+4}, which gives the ith iteration, and copy the result of the previous computation into R_{2n+5}. Then invoke the program P_h to obtain $f(\boldsymbol{x}, i) = h(\boldsymbol{x}, i-1, f(\boldsymbol{x}, i-1))$. At the end, the input arguments are set to zero and the result of the last iteration is copied into the first register. This provides the desired result: $\|P_f\|_{n+1,1} = f$. □

The URM programs for composition and primitive recursion make also sense if the URM subprograms used in the respective induction step are not primitive recursive. These ideas will be formalized in the remaining part of the section.

Let $g : \mathbb{N}_0^k \to \mathbb{N}_0$ and $h_i : \mathbb{N}_0^n \to \mathbb{N}_0$, $1 \leq i \leq k$, be partial functions. The *composition* of g along $(h_1, \ldots, h_k)$ is a partial function f, denoted by $f = g(h_1, \ldots, h_k)$, such that

$$\mathrm{dom}(f) = \{\boldsymbol{x} \in \mathbb{N}_0^n \mid \boldsymbol{x} \in \bigcap_{i=1}^{k} \mathrm{dom}(h_i) \wedge (h_1(\boldsymbol{x}), \ldots, h_k(\boldsymbol{x})) \in \mathrm{dom}(g)\} \tag{2.9}$$

and

$$f(\boldsymbol{x}) = g(h_1(\boldsymbol{x}), \ldots, h_k(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathrm{dom}(f). \tag{2.10}$$

The proof of the previous theorem provides the following result.

Proposition 2.13. *The class of URM computable functions is closed under composition; that is, if $g : \mathbb{N}_0^k \to \mathbb{N}_0$ and $h_i : \mathbb{N}_0^n \to \mathbb{N}_0$, $1 \leq i \leq k$, are URM computable, $f = g(h_1, \ldots, h_k)$ is URM computable.*

The situation is analogous for primitive recursion.

Proposition 2.14. *Let $g : \mathbb{N}_0^n \to \mathbb{N}_0$ and $h : \mathbb{N}_0^{n+2} \to \mathbb{N}_0$ be partial functions. There is a unique function $f : \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ such that*

1. *$(\boldsymbol{x}, 0) \in \mathrm{dom}(f)$ if and only if $\boldsymbol{x} \in \mathrm{dom}(g)$ for all $\boldsymbol{x} \in \mathbb{N}_0^n$,*
2. *$(\boldsymbol{x}, y+1) \in \mathrm{dom}(f)$ if and only if $(\boldsymbol{x}, y) \in \mathrm{dom}(f)$ and $(\boldsymbol{x}, y, f(\boldsymbol{x}, y)) \in \mathrm{dom}(h)$ for all $\boldsymbol{x} \in \mathbb{N}_0^n$, $y \in \mathbb{N}_0$,*

3. $f(\boldsymbol{x},0) = g(\boldsymbol{x})$ *for all* $\boldsymbol{x} \in \operatorname{dom}(f)$*, and*

4. $f(\boldsymbol{x},y+1) = h(\boldsymbol{x},y,f(\boldsymbol{x},y))$ *for all* $(\boldsymbol{x},y+1) \in \operatorname{dom}(f)$*.*

The proof makes use of the fundamental lemma. The partial function f defined by g and h in this proposition is also denoted by $f = \operatorname{pr}(g,h)$ and said to be defined by *primitive recursion* of g along h.

Proposition 2.15. *The class of URM computable functions is closed under primitive recursion; that is, if* $g : \mathbb{N}_0^n \to \mathbb{N}_0$ *and* $h : \mathbb{N}_0^{n+2} \to \mathbb{N}_0$ *are URM computable,* $f = \operatorname{pr}(g,h)$ *is URM computable.*

Primitively Closed Function Classes

Let $\mathcal{F}$ be a class of functions, i.e.,

$$\mathcal{F} \subseteq \bigcup_{k\geq 0} \mathbb{N}_0^{(\mathbb{N}_0^k)}.$$

The class $\mathcal{F}$ is called *primitively closed* if it contains the basic functions $c_0^{(0)}$, $c_0^{(1)}$, ν, $\pi_k^{(n)}$, $1 \leq k \leq n$, $n \geq 1$, and is closed under composition and primitive recursion.

Let $\mathcal{P}$ denote the class of all primitive recursive functions, $\mathcal{T}_{\text{URM}}$ the class of all URM computable total functions, and $\mathcal{T}$ the class of all total functions.

Proposition 2.16. *The classes* $\mathcal{P}$*,* $\mathcal{T}_{\text{URM}}$*, and* $\mathcal{T}$ *are primitively closed.*

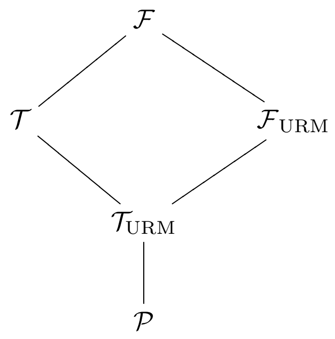
In particular, the class $\mathcal{P}$ of primitive recursive functions is the smallest class of functions which is primitively closed. Indeed, we have

$$\mathcal{P} = \bigcap \{\mathcal{F} \mid \mathcal{F} \subseteq \bigcup_{k\geq 0} \mathbb{N}_0^{(\mathbb{N}_0^k)}, \mathcal{F} \text{ primitively closed}\}. \tag{2.11}$$

The concept of primitive closure carries over to partial functions, since composition and primitive recursion have been defined for partial functions as well. Let $\mathcal{F}_{\text{URM}}$ denote the class of URM computable functions and $\mathcal{F}$ the class of all functions.

Proposition 2.17. *The classes* $\mathcal{F}_{\text{URM}}$ *and* $\mathcal{F}$ *are primitively closed.*

The lattice of the introduced classes (under inclusion) is the following:



All inclusions are strict. Indeed, the strict inclusions $\mathcal{T}_{\text{URM}} \subset \mathcal{F}_{\text{URM}}$ and $\mathcal{T} \subset \mathcal{F}$ are obvious, while the strict inclusions $\mathcal{T}_{\text{URM}} \subset \mathcal{T}$ and $\mathcal{F}_{\text{URM}} \subset \mathcal{F}$ follow by counting arguments. However, the strict inclusion $\mathcal{P} \subset \mathcal{T}_{\text{URM}}$ is not clear at all. An example of a total URM computable function that is not primitive recursive will be given in Chapter 4.

2.3 Closure Properties

This section provides a small repository of algorithmic properties and constructions for later use.

Transformation of Variables and Parametrization

Given a function $f : \mathbb{N}_0^n \to \mathbb{N}_0$ and a mapping $\phi : [n] \to [m]$. The function f^ϕ obtained from f by *transformation of variables* with respect to ϕ is defined as

$$f^\phi : \mathbb{N}_0^m \to \mathbb{N}_0 : (x_1, \ldots, x_m) \mapsto f(x_{\phi(1)}, \ldots, x_{\phi(n)}). \tag{2.12}$$

Proposition 2.18. *If the function f is primitive recursive, the function f^ϕ is also primitive recursive.*

Proof. Transformation of variables can be described by the composition

$$f^\phi = f(\pi_{\phi(1)}^{(m)}, \ldots, \pi_{\phi(n)}^{(m)}). \tag{2.13}$$

□

Examples 2.19. Three important special cases for dyadic functions:

- Permutation of variables: $f^\phi : (x, y) \mapsto f(y, x)$, where $\phi : [2] \to [2] : (a, b) \mapsto (b, a)$.
- Adjunct of variables: $f^\phi : (x, y) \mapsto f(x)$, where $\phi : [2] \to [1] : (a, b) \mapsto a$.
- Identification of variables: $f^\phi : x \mapsto f(x, x)$, where $\phi : [1] \to [2] : a \mapsto (a, a)$.

♢

Let $c_i^{(k)}$ denote the k-ary constant function with value $i \in \mathbb{N}_0$, i.e.,

$$c_i^{(k)} : \mathbb{N}_0^k \to \mathbb{N}_0 : (x_1, \ldots, x_k) \mapsto i. \tag{2.14}$$

Proposition 2.20. *The constant function $c_i^{(k)}$ is primitive recursive.*

Proof. If $k = 0$, $c_i^{(0)} = \nu^i \circ c_0^{(0)}$. Otherwise, $c_i^{(k)} = \nu^i \circ c_0^{(1)} \circ \pi_1^{(k)}$. □

Let $f : \mathbb{N}_0^n \to \mathbb{N}_0$ be a function. Take a positive integer m with $m < n$ and $\boldsymbol{a} = (a_1, \ldots, a_m) \in \mathbb{N}_0^m$. The function $f_{\boldsymbol{a}}$ obtained from f by *parametrization* with respect to a is defined as

$$f_{\boldsymbol{a}} : \mathbb{N}_0^{n-m} \to \mathbb{N}_0 : (x_1, \ldots, x_{n-m}) \mapsto f(x_1, \ldots, x_{n-m}, a_1, \ldots, a_m). \tag{2.15}$$

Proposition 2.21. *If the function f is primitive recursive, the function $f_{\boldsymbol{a}}$ is also primitive recursive.*

Proof. Parametrization can be described by the composition

$$f_{\boldsymbol{a}} = f(\pi_1^{(n-m)}, \ldots, \pi_{n-m}^{(n-m)}, c_{a_1}^{(n-m)}, \ldots, c_{a_m}^{(n-m)}). \tag{2.16}$$

□

Definition by Cases

Let $h_i : \mathbb{N}_0^k \to \mathbb{N}_0$, $1 \leq i \leq r$, be total functions with the property that for each $\boldsymbol{x} \in \mathbb{N}_0^k$ there is a unique index $i \in [r]$ such that $h_i(\boldsymbol{x}) = 0$. That is, the sets $H_i = \{\boldsymbol{x} \in \mathbb{N}_0^k \mid h_i(\boldsymbol{x}) = 0\}$ form a partition of the whole set $\mathbb{N}_0^k$. Moreover, let $g_i : \mathbb{N}_0^k \to \mathbb{N}_0$, $1 \leq i \leq r$, be arbitrary total functions. Define the function

$$f : \mathbb{N}_0^k \to \mathbb{N}_0 : \boldsymbol{x} \mapsto \begin{cases} g_1(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in H_1, \\ \vdots & \vdots \\ g_r(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in H_r. \end{cases} \tag{2.17}$$

The function f is clearly total and said to be *defined by cases.*

Proposition 2.22. *If the above functions g_i and h_i, $1 \leq i \leq r$, are primitive recursive, the function f is also primitive recursive.*

Proof. The function f is given as follows,

$$f = \sum_{i=1}^{r} g_i \cdot (\mathrm{csg} \circ h_i). \tag{2.18}$$

Indeed, let $\boldsymbol{x} \in \mathbb{N}_0^k$. Then there is exactly one index j, $1 \leq j \leq r$, such that $\boldsymbol{x} \in H_j$. By definition, we have $h_j(\boldsymbol{x}) = 0$ and $h_i(\boldsymbol{x}) \neq 0$ for all $i \neq j$, $1 \leq i \leq r$. It follows that $f(\boldsymbol{x}) = \sum_{i=1}^r g_i(\boldsymbol{x}) \cdot (\mathrm{csg} \circ h_i(\boldsymbol{x})) = g_j(\boldsymbol{x}) \cdot (\mathrm{csg} \circ h_j(\boldsymbol{x})) = g_j(\boldsymbol{x})$, as claimed. □

Example 2.23. Consider the monadic function $f : \mathbb{N}_0 \to \mathbb{N}_0$ defined by

$$f(x) = \begin{cases} 2x & \text{if } x > 0, \\ 1 & \text{otherwise.} \end{cases} \tag{2.19}$$

The cases are defined by the partition given by the sets $H_1 = \{x \in \mathbb{N}_0 \mid x > 0\}$ and $H_2 = \{0\}$, and the corresponding mappings $h_1 = \mathrm{csg}$ and $h_2 = 1 - \mathrm{csg} = \mathrm{sgn}$, respectively. Moreover, $g_1(x) = 2x$ and $g_2(x) = 1$ for each $x \in \mathbb{N}_0$. In this way, we obtain for each $x \in \mathbb{N}_0$, $f(x) = g_1(x) \cdot (\mathrm{csg} \circ h_1(x)) + g_2(x) \cdot (\mathrm{csg} \circ h_2(x)) = g_1(x) \cdot \mathrm{sgn}(x) + g_2(x) \cdot \mathrm{csg}(x)$. ♢

Bounded Sum and Product

Let $f : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ be a total function. The *bounded sum* of f is the function

$$\Sigma f : \mathbb{N}_0^{k+1} \to \mathbb{N}_0 : (x_1, \ldots, x_k, y) \mapsto \sum_{i=0}^{y} f(x_1, \ldots, x_k, i) \tag{2.20}$$

and the *bounded product* of f is the function

$$\Pi f : \mathbb{N}_0^{k+1} \to \mathbb{N}_0 : (x_1, \ldots, x_k, y) \mapsto \prod_{i=0}^{y} f(x_1, \ldots, x_k, i). \tag{2.21}$$

Proposition 2.24. *If the function f is primitive recursive, the functions Σf and Πf are also primitive recursive.*

Proof. The function Σf is given as

$$\Sigma f(\boldsymbol{x},0)=f(\boldsymbol{x},0) \quad \text{and} \quad \Sigma f(\boldsymbol{x},y+1)=\Sigma f(\boldsymbol{x},y)+f(\boldsymbol{x},y+1). \tag{2.22}$$

This corresponds to the primitive recursive scheme $\Sigma f=\mathrm{pr}(g,h)$, where $g(\boldsymbol{x})=f(\boldsymbol{x},0)$ and $h(\boldsymbol{x},y,z)=+(f(\boldsymbol{x},\nu(y)),z)$. The function Πf can be similarly defined. □

Example 2.25. Take the function $f:\mathbb{N}_0\to\mathbb{N}_0$ defined by $f(x)=1$ if $x=0$ and $f(x)=x$ if $x>0$. This function defined by cases is primitive recursive and thus the bounded product (factorial function), given by $\Pi f(x)=x!$ for all $x\in\mathbb{N}_0$, is primitive recursive as well. ♢

Bounded Minimalization

Let $f:\mathbb{N}_0^{k+1}\to\mathbb{N}_0$ be a total function. The *bounded minimalization* of f is the function

$$\overline{\mu}f:\mathbb{N}_0^{k+1}\to\mathbb{N}_0:(\boldsymbol{x},y)\mapsto\mu(i\leq y)[f(\boldsymbol{x},i)=0], \tag{2.23}$$

where for each $(\boldsymbol{x},y)\in\mathbb{N}_0^{k+1}$,

$$\mu(i\leq y)[f(\boldsymbol{x},i)=0]=\begin{cases} j & \text{if } j=\min\{i\mid 0\leq i\leq y\wedge f(\boldsymbol{x},i)=0\} \text{ exists,}\\ y+1 & \text{otherwise.}\end{cases} \tag{2.24}$$

That is, the value $\overline{\mu}f(\boldsymbol{x},y)$ provides the smallest index j with $0\leq j\leq y$ such that $f(\boldsymbol{x},j)=0$. If there is no such index, the value is $y+1$. In this way, bounded minimalization can be viewed as a bounded search process.

Proposition 2.26. *If the function f is primitive recursive, the function $\overline{\mu}f$ is also primitive recursive.*

Proof. By definition,

$$\overline{\mu}f(\boldsymbol{x},0)=\mathrm{sgn}(f(\boldsymbol{x},0)) \tag{2.25}$$

and

$$\overline{\mu}f(\boldsymbol{x},y+1)=\begin{cases} \overline{\mu}f(\boldsymbol{x},y) & \text{if } \overline{\mu}f(\boldsymbol{x},y)\leq y \text{ or } f(\boldsymbol{x},y+1)=0,\\ y+2 & \text{otherwise.}\end{cases} \tag{2.26}$$

Define the $k+2$-ary functions

$$\begin{aligned} g_1&:(\boldsymbol{x},y,z)\mapsto z,\\ g_2&:(\boldsymbol{x},y,z)\mapsto y+2,\\ h_1&:(\boldsymbol{x},y,z)\mapsto(z\dot{-}y)\cdot\mathrm{sgn}(f(\boldsymbol{x},y+1)),\\ h_2&:(\boldsymbol{x},y,z)\mapsto\mathrm{csg}(h_1(\boldsymbol{x},y,z)).\end{aligned} \tag{2.27}$$

These functions are primitive recursive. Moreover, the functions h_1 and h_2 provide a partition of $\mathbb{N}_0$. Thus the following function defined by cases is also primitive recursive:

$$g(\boldsymbol{x}, y, z) = \begin{cases} g_1(\boldsymbol{x}, y, z) \text{ if } h_1(\boldsymbol{x}, y, z) = 0, \\ g_2(\boldsymbol{x}, y, z) \text{ if } h_2(\boldsymbol{x}, y, z) = 0. \end{cases} \tag{2.28}$$

We have $h_1(\boldsymbol{x}, y, \overline{\mu}f(\boldsymbol{x}, y)) = 0$ if and only if $(\overline{\mu}f(\boldsymbol{x}, y)) \dot{-} y) \cdot \text{sgn}(f(\boldsymbol{x}, y+1)) = 0$, which is equivalent to $\overline{\mu}f(\boldsymbol{x}, y)) \leq y$ or $f(\boldsymbol{x}, y+1) = 0$. In this case, $g_1(\boldsymbol{x}, y, \overline{\mu}f(\boldsymbol{x}, y)) = \overline{\mu}f(\boldsymbol{x}, y+1)$. The other case can be similarly evaluated. It follows that the bounded minimalization $\overline{\mu}f$ corresponds to the primitive recursive scheme $\overline{\mu}f = \text{pr}(s, g)$, where $s : \mathbb{N}_0^k \to \mathbb{N}_0$ is defined as $s(\boldsymbol{x}) = \text{sgn}(f(\boldsymbol{x}, 0))$ and g is given as above. □

Example 2.27. Consider the integral division function

$$\div : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x, y) \mapsto \begin{cases} \lfloor x/y \rfloor & \text{if } y > 0, \\ x & \text{if } y = 0, \end{cases} \tag{2.29}$$

where the expression $\lfloor x/y \rfloor$ means that $\lfloor x/y \rfloor = z$ if $y \cdot z \geq x$ and z is minimal with this property.

The function $\div$ is defined by cases. In view of the case $y > 0$, the value z can be calculated by bounded minimalization. In this way, the integral division function is given by

$$\div(x, y) = \mu(z \leq x)[y \cdot z \dot{-} x = 0] \cdot \text{sgn}(y) + x \cdot \text{csg}(y), \quad x, y \in \mathbb{N}_0.$$

The associated remainder function given by $\text{rem}(x, y) = y \cdot \div(x, y) \dot{-} x$ if $y > 0$ and $\text{rem}(x, y) = 0$ if $y = 0$ and is therefore also primitive recursive. ♢

Iteration

The *powers* of a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ are inductively defined as

$$f^0 = \text{id}_{\mathbb{N}_0} \quad \text{and} \quad f^{n+1} = f \circ f^n, \; n \geq 0. \tag{2.30}$$

In particular, $f^1 = f \circ f^0 = f \circ \text{id}_{\mathbb{N}_0} = f$. The *iteration* of a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ is given by the function

$$g : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x, y) \mapsto f^y(x). \tag{2.31}$$

Example 2.28. Consider the function $f : \mathbb{N}_0 \to \mathbb{N}_0 : x \mapsto 2x$. The iteration of f is the function $g : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by $g(x, y) = 2^y \cdot x$. ♢

Proposition 2.29. *If f is a monadic primitive recursive function, the iteration of f is also primitive recursive.*

Proof. The iteration g of f follows the primitive recursive scheme

$$g(x, 0) = x \tag{2.32}$$

and

$$g(x, y+1) = f(g(x, y)) = f \circ \pi_3^{(3)}(x, y, g(x, y)), \; x, y \in \mathbb{N}_0. \tag{2.33}$$

□

Iteration can also be defined for multivariate functions. For this, let $f : \mathbb{N}_0^k \to \mathbb{N}_0^k$ be a function defined by coordinate functions $f_i : \mathbb{N}_0^k \to \mathbb{N}_0$, $1 \leq i \leq k$, as follows:

$$f(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathbb{N}_0^k. \tag{2.34}$$

Write $f = (f_1, \ldots, f_k)$ and define the *powers* of f inductively as follows:

$$f^0(\boldsymbol{x}) = \boldsymbol{x} \quad \text{and} \quad f^{n+1}(\boldsymbol{x}) = (f_1(f^n(\boldsymbol{x})), \ldots, f_k(f^n(\boldsymbol{x}))), \quad \boldsymbol{x} \in \mathbb{N}_0^k. \tag{2.35}$$

These definitions give immediately rise to the following result.

Proposition 2.30. *If the functions $f = (f_1, \ldots, f_k)$ are primitive recursive, the powers of f are also primitive recursive.*

The *iteration* of $f = (f_1, \ldots, f_k)$ is defined by the functions

$$g_i : \mathbb{N}_0^{k+1} \to \mathbb{N}_0 : (\boldsymbol{x}, y) \mapsto (\pi_i^{(k)} \circ f^y)(\boldsymbol{x}), \quad 1 \leq i \leq k. \tag{2.36}$$

Proposition 2.31. *If the functions $f = (f_1, \ldots, f_k)$ are primitive recursive, the iteration of f is also primitive recursive.*

Proof. The iteration of $f = (f_1, \ldots, f_k)$ follows the primitive recursive scheme

$$g_i(\boldsymbol{x}, 0) = x_i \tag{2.37}$$

and

$$g_i(\boldsymbol{x}, y + 1) = f_i(f^y(\boldsymbol{x})) = f_i \circ \pi_{k+2}^{(k+2)}(\boldsymbol{x}, y, g_i(\boldsymbol{x}, y)), \quad \boldsymbol{x} \in \mathbb{N}_0^k, y \in \mathbb{N}_0,\ 1 \leq i \leq k. \tag{2.38}$$

□

Pairing Functions

A pairing function encodes the pairs of natural numbers by single natural numbers. A primitive recursive bijection from $\mathbb{N}_0^2$ onto $\mathbb{N}_0$ is called a *pairing function.* In set theory, any pairing function can be used to prove that the rational numbers have the same cardinality as the natural numbers. For instance, the *Cantor function* $J_2 : \mathbb{N}_0^2 \to \mathbb{N}_0$ is defined as

$$J_2(m, n) = \binom{m + n + 1}{2} + m. \tag{2.39}$$

This function will provide a proof that the cartesian product $\mathbb{N}_0^2$ is denumerable. To this end, write down the elements of $\mathbb{N}_0^2$ in a table as follows:

$$\begin{array}{llllll}
(0,0) & (0,1) & (0,2) & (0,3) & (0,4) & \ldots \\
(1,0) & (1,1) & (1,2) & (1,3) & \ldots & \\
(2,0) & (2,1) & (2,2) & \ldots & & \\
(3,0) & (3,1) & \ldots & & & \\
(4,0) & \ldots & & & & \\
\ldots & & & & &
\end{array} \tag{2.40}$$

For each number $k \geq 0$, the *k-th anti-diagonal* in this table is given by the sequence

$$(0,k),(1,k-1),\ldots,(k,0). \tag{2.41}$$

Now generate a list of all elements of $\mathbb{N}_0^2$ by writing down the anti-diagonals in consecutive order starting with the 0-th anti-diagonal:

$$(0,0),(0,1),(1,0),(0,2),(1,1),(2,0),(0,3),(1,2),(2,1),(3,0),(0,4),\ldots. \tag{2.42}$$

Claim that the pair $(0,m+n)$ is at position $\binom{m+n+1}{2}$ in the list. Indeed, the pair $(0,0)$ lies at position 0 if we put $\binom{1}{2}=0$. Suppose that the pair $(0,m+n)$ is at position $\binom{m+n+1}{2}$. Since the $m+n$-th antidiagonal has $m+n+1$ elements, the pair $(0,m+n+1)$ lies at position $\binom{m+n+1}{2}+m+n+1$ which equals $\binom{(m+n+1)+1}{2}$, as claimed.

The pair (m,n) lies at position m in the $m+n$-th anti-diagonal and therefore occurs in the list at position $J_2(m,n)=\binom{m+n+1}{2}+m$. This shows that the function J_2 is bijective.

Proposition 2.32. *The Cantor function J_2 is primitive recursive.*

Proof. By using the integral division function $\div$, one obtains

$$J_2(m,n) = \div((m,n)\cdot(m+n+1),2)+m. \tag{2.43}$$

Thus J_2 is primitive recursive. □

The Cantor function J_2 can be inverted by taking coordinate functions $K_2, L_2 : \mathbb{N}_0 \to \mathbb{N}_0$ such that

$$J_2^{-1}(n) = (K_2(n), L_2(n)), \quad n \in \mathbb{N}_0. \tag{2.44}$$

In order to define them, take $n \in \mathbb{N}_0$. Find a number $s \geq 0$ such that

$$\frac{1}{2}s(s+1) \leq n < \frac{1}{2}(s+1)(s+2) \tag{2.45}$$

and put

$$m = n - \frac{1}{2}s(s+1). \tag{2.46}$$

Then

$$m = n - \frac{1}{2}s(s+1) \leq [\frac{1}{2}(s+1)(s+2)-1] - [\frac{1}{2}s(s+1)] = s. \tag{2.47}$$

Finally, set

$$K_2(n) = m \quad \text{and} \quad L_2(n) = s - m. \tag{2.48}$$

Example 2.33. Let $n = 17$. Then $\frac{1}{2}s(s+1) \leq 17 < \frac{1}{2}(s+1)(s+2)$ is satisfied by $s = 5$. Thus $K_2(17) = 2$ and $L_2(17) = 3$. Note that $J_2(2,3) = 17$. ♢

Proposition 2.34. *The coordinate functions K_2 and L_2 are primitive recursive and the pair (K_2, L_2) is the inverse of J_2.*

Proof. Let $n \in \mathbb{N}_0$. The corresponding number s in (2.45) can be determined by bounded minimalization as follows:

$$\mu(i \leq n)[n \mathbin{\dot{-}} \frac{1}{2}s(s+1) = 0] = \begin{cases} j & \text{if } j = \min\{i \mid i \leq y \wedge n \mathbin{\dot{-}} \frac{1}{2}i(i+1) = 0\} \text{ exists,} \\ n+1 & \text{otherwise.} \end{cases} \tag{2.49}$$

The value j always exists and we have

$$s = \begin{cases} j & \text{if } n = \frac{1}{2}j(j+1), \\ j \mathbin{\dot{-}} 1 & \text{otherwise.} \end{cases} \tag{2.50}$$

Then $K_2(n) = n - \div(s(s+1), 2)$ and $L_2(n) = s - K_2(n)$. Thus both, K_2 and L_2 are primitive recursive. Finally, by (2.46) and (2.48), $J_2(K_2(n), L_2(n)) = J_2(m, s-m-1) = \frac{1}{2}s(s+1) + m = n$ and so (2.44) follows. □

This assertion implies that the Cantor function J_2 is a pairing function.

2.4 Primitive Recursive Sets

The studies can be extended to relations given as subsets of $\mathbb{N}_0^k$ by taking their characteristic function. Let S be a subset of $\mathbb{N}_0^k$. The *characteristic function* of S is the function $\chi_S : \mathbb{N}_0^k \to \mathbb{N}_0$ defined by

$$\chi_S(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in S, \\ 0 & \text{otherwise.} \end{cases} \tag{2.51}$$

A subset S of $\mathbb{N}_0^k$ is called *primitive* if its characteristic function χ_S is primitive recursive.

Examples 2.35. Here are some primitive basic relations:

1. The equality relation $R_= = \{(x,y) \in \mathbb{N}_0^2 \mid x = y\}$ is primitive, since the corresponding characteristic function $\chi_{R_=}(x,y) = \mathrm{csg}(|x-y|)$ is primitive recursive.
2. The inequality relation $R_{\neq} = \{(x,y) \in \mathbb{N}_0^2 \mid x \neq y\}$ is primitive, since its characteristic function $\chi_{R_{\neq}}(x,y) = 1 - \mathrm{csg}(|x-y|)$ is primitive recursive.
3. The smaller relation $R_< = \{(x,y) \in \mathbb{N}_0^2 \mid x < y\}$ is primitive, since the associated characteristic function $\chi_{R_<}(x,y) = \mathrm{sgn}(y \mathbin{\dot{-}} x)$ is primitive recursive.

♢

Proposition 2.36. *If S and T are primitive subsets of $\mathbb{N}_0^k$, then $S \cup T$, $S \cap T$, and $\mathbb{N}_0^k \setminus S$ are also primitive.*

Proof. Clearly, for each $\boldsymbol{x} \in \mathbb{N}_0^k$, we have $\chi_{S \cup T}(\boldsymbol{x}) = \mathrm{sgn}(\chi_S(\boldsymbol{x}) + \chi_T(\boldsymbol{x}))$, $\chi_{S \cap T}(\boldsymbol{x}) = \chi_S(\boldsymbol{x}) \cdot \chi_T(\boldsymbol{x})$, and $\chi_{\mathbb{N}_0^k \setminus S}(\boldsymbol{x}) = \mathrm{csg} \circ \chi_S(\boldsymbol{x})$. □

Let S be a subset of $\mathbb{N}_0^{k+1}$. The *bounded existential quantification* of S is a subset of $\mathbb{N}_0^{k+1}$ given as

$$\exists S = \{(\boldsymbol{x}, y) \mid (\boldsymbol{x}, i) \in S \text{ for some } 0 \leq i \leq y\}. \tag{2.52}$$

The *bounded universal quantification* of S is a subset of $\mathbb{N}_0^{k+1}$ defined by

$$\forall S = \{(\boldsymbol{x}, y) \mid (\boldsymbol{x}, i) \in S \text{ for all } 0 \leq i \leq y\}. \tag{2.53}$$

Proposition 2.37. *If S is a primitive subset of $\mathbb{N}_0^{k+1}$, the sets $\exists S$ and $\forall S$ are also primitive.*

Proof. Clearly, for each $(\boldsymbol{x}, y) \in \mathbb{N}_0^{k+1}$, we have $\chi_{\exists S}(\boldsymbol{x}, y) = \text{sgn}((\varSigma\chi_S)(\boldsymbol{x}, y))$ and $\chi_{\forall S}(\boldsymbol{x}, y) = (\varPi\chi_S)(\boldsymbol{x}, y)$. □

Consider the sequence of increasing primes $(p_0, p_1, p_2, p_3, p_4, \ldots) = (2, 3, 5, 7, 11, \ldots)$. By the fundamental theorem of arithmetic, each natural number $x \geq 1$ can be uniquely written as a product of prime powers, i.e.,

$$x = \prod_{i=0}^{r-1} p_i^{e_i}, \quad e_0, \ldots, e_{r-1} \in \mathbb{N}_0. \tag{2.54}$$

Write $(x)_i = e_i$ for each $i \in \mathbb{N}_0$, and put $(0)_i = 0$ for all $i \in \mathbb{N}_0$. For instance, $24 = 2^3 \cdot 3$ and so $(24)_0 = 3$, $(24)_1 = 1$, and $(24)_i = 0$ for all $i \geq 2$.

Proposition 2.38.

1. *The divisibility relation* $D = \{(x,y) \in \mathbb{N}_0^2 \mid x$ *divides* $y\}$ *is primitive.*
2. *The set of primes is primitive.*
3. *The function* $p : \mathbb{N}_0 \to \mathbb{N}_0 : i \mapsto p_i$ *is primitive recursive.*
4. *The function* $\tilde{p} : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x,i) \mapsto (x)_i$ *is primitive recursive.*

Proof. First, x divides y, written $x \mid y$, if and only if $x \cdot i = y$ for some $0 \leq i \leq y$. Thus the characteristic function of D can be written as

$$\chi_D(x,y) = \mathrm{sgn}[\chi_=(x \cdot 1, y) + \chi_=(x \cdot 2, y) + \ldots + \chi_=(x \cdot y, y)]. \tag{2.55}$$

Hence, the relation D is primitive.

Second, a natural number x is prime if and only if $x \geq 2$ and i divides x implies $i = 1$ or $i = x$ for all $i \leq x$. Thus the characteristic function of the set P of primes is given as follows: $\chi_P(0) = \chi_P(1) = 0$, $\chi_P(2) = 1$, and

$$\chi_P(x) = \mathrm{csg}[\chi_D(2,x) + \chi_D(3,x) + \ldots + \chi_D(x-1,x)], \quad x \geq 3. \tag{2.56}$$

Third, define the functions

$$g(z,x) = |\chi_{R_<}(z,x) \cdot \chi_P(x) - 1| = \begin{cases} 0 \text{ if } z < x \text{ and } x \text{ prime,} \\ 1 \text{ otherwise,} \end{cases} \tag{2.57}$$

and

$$h(z) = \overline{\mu} g(z, z!+1) = \mu(y \leq z!+1)[g(z,y) = 0]. \tag{2.58}$$

The function g is primitive recursive and thus the function g is also primitive recursive. By a theorem of Euclid, the $i+1$th prime is bounded by the ith prime in a way that $p_{i+1} \leq p_i! + 1$ for all $i \geq 0$. Thus the value $h(p_i)$ provides the next prime p_{i+1}. That is, the sequence of prime numbers is given by the primitive recursive scheme

$$p_0 = 2 \quad \text{and} \quad p_{i+1} = h(p_i),\ i \geq 0. \tag{2.59}$$

Fourth, we have

$$(x)_i = \mu(y \leq x)[p_i^{y+1} \not| \, x] = \mu(y \leq x)[\chi_D(p_i^{y+1}, x) = 0]. \tag{2.60}$$

□

2.5 LOOP Programs

This section provides a mechanistic description of the class of primitive recursive functions. For this, a class of URM computable functions is introduced in which the use of loop variables is restricted. More specifically, the only loops or iterations allowed will be of the form $(M; S\sigma)\sigma$, where the variable σ does not appear in the program M. In this way, the program M cannot manipulate the register R_σ and thus it can be guaranteed that the program M will be carried out n times, where n is the content of the register R_σ at the start of the computation. Hence, loops of this type allow an explicit control over the loop variables.

Two abbreviations will be used in the following: If P is an URM program and $\sigma \in Z$, write $[P]\sigma$ for the program $(P; S\sigma)\sigma$, and denote by $Z\sigma$ the URM program $(S\sigma)\sigma$.

The class $\mathcal{P}_{\text{LOOP}}$ of LOOP programs is inductively defined as follows:

1. Define the class of LOOP-0 programs $\mathcal{P}_{\text{LOOP}(0)}$:
   a) For each $\sigma \in Z$, $A\sigma \in \mathcal{P}_{\text{LOOP}(0)}$ and $Z\sigma \in \mathcal{P}_{\text{LOOP}(0)}$.
   b) For each $\sigma, \tau \in Z$ with $\sigma \neq \tau$ and $\sigma \neq 0 \neq \tau$, $\bar{C}(\sigma,\tau) = Z\tau; Z0; C(\sigma;\tau) \in \mathcal{P}_{\text{LOOP}(0)}$.
   c) If $P, Q \in \mathcal{P}_{\text{LOOP}(0)}$, then $P; Q \in \mathcal{P}_{\text{LOOP}(0)}$.
2. Suppose the class of LOOP-n programs $\mathcal{P}_{\text{LOOP}(n)}$ has already been defined. Define the class of LOOP-$n+1$ programs $\mathcal{P}_{\text{LOOP}(n+1)}$:
   a) Each $P \in \mathcal{P}_{\text{LOOP}(n)}$ belongs to $\mathcal{P}_{\text{LOOP}(n+1)}$.
   b) If $P, Q \in \mathcal{P}_{\text{LOOP}(n+1)}$, then $P; Q \in \mathcal{P}_{\text{LOOP}(n+1)}$.
   c) if $P \in \mathcal{P}_{\text{LOOP}(n)}$ and $\sigma \in Z$ does not appear in P, then $[P]\sigma \in \mathcal{P}_{\text{LOOP}(n+1)}$.

Note that for each $\omega \in \Omega$, $\bar{w} = \bar{C}(\sigma,\tau)(\omega)$ is given by

$$\bar{\omega}_n = \begin{cases} 0 & \text{if } n = 0, \\ \omega_\sigma & \text{if } n = \sigma \text{ or } n = \tau, \\ \omega_n & \text{otherwise.} \end{cases} \tag{2.61}$$

That is, the content of register R_σ is copied into register R_τ and the register R_0 is set to zero.

Note that the LOOP-n programs form as sets a proper hierarchy:

$$\mathcal{P}_{\text{LOOP}(0)} \subset \mathcal{P}_{\text{LOOP}(1)} \subset \mathcal{P}_{\text{LOOP}(2)} \subset \ldots. \tag{2.62}$$

The class of *LOOP programs* is defined as as the union of LOOP-n programs for all $n \in \mathbb{N}_0$:

$$\mathcal{P}_{\text{LOOP}} = \bigcup_{n \geq 0} \mathcal{P}_{\text{LOOP}(n)}. \tag{2.63}$$

In particular, $\mathcal{P}_{\text{LOOP}(n)}$ is called the class of LOOP programs *of depth n*, $n \in \mathbb{N}_0$.

Proposition 2.39. *For each LOOP program P, the function $\|P\|$ is total.*

Proof. For each LOOP-0 program P, it is clear that the function $\|P\|$ is total. Let P be a LOOP-n program and let $\sigma \in Z$ such that σ does not appear in P. By induction hypothesis, the function $\|P\|$ is total. Moreover, $\|[P]\sigma\| = \|P^k\|$, where k is the content of register R_σ at the beginning of the computation. Thus the function $\|[P]\sigma\|$ is also total. The remaining cases are clear. □

A function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is called *LOOP-n computable* if there is a LOOP-n program P such that $\|P\|_{k,1} = f$. Let $\mathcal{F}_{\text{LOOP}(n)}$ denote the class of all LOOP-n computable functions and define the class of all LOOP computable functions $\mathcal{F}_{\text{LOOP}}$ as the union of LOOP-n computable functions for all $n \geq 0$:

$$\mathcal{F}_{\text{LOOP}} = \bigcup_{n \geq 0} \mathcal{F}_{\text{LOOP}(n)}. \tag{2.64}$$

Note that if P is a LOOP-n program, $n \geq 1$, and P' is the normal program corresponding to P, then P' is also a LOOP-n program.

Example 2.40. The program $S\sigma$ does not belong to the basic LOOP programs. But it can be described by a LOOP-1 program. Indeed, put

$$P_{\dot{-}\,1} = \bar{C}(1;3); [\bar{C}(2;1); A2]3. \tag{2.65}$$

Then we have for input $x = 0$,

0	1	2	3	4	… registers
0	0	0	0	0	… init
0	0	0	0	0	… $\bar{C}(1;3)$
0	0	0	0	0	… end

and for input $x \geq 1$,

0	1	2	3	4	… registers
0	x	0	0	0	… init
0	x	0	x	0	… $\bar{C}(1;3)$
0	0	0	x	0	… $\bar{C}(2;1)$
0	0	1	x	0	… $A2$
0	0	1	$x-1$	0	… $S3$
…					
0	1	2	$x-2$	0	…
…					
0	$x-1$	x	0	0	… end

It follows that $\|P_{\dot{-}\,1}\|_{1,1} = \|S1\|_{1,1}$. ♢

The LOOP-n computable functions form a hierarchy but at this stage it is not clear whether it is proper or not:

$$\mathcal{T}_{\text{LOOP}(0)} \subseteq \mathcal{T}_{\text{LOOP}(1)} \subseteq \mathcal{T}_{\text{LOOP}(2)} \subseteq \ldots. \tag{2.66}$$

Theorem 2.41. *The class of LOOP computable functions equals the class of primitive recursive functions.*

Proof. First, claim that each primitive recursive function is LOOP computable. Indeed, the basic primitive recursive functions are LOOP computable:

1. 0-ary constant function : $\|Z0\|_{0,1} = c_0^{(0)}$,
2. monadic constant function : $\|Z1\|_{1,1} = c_0^{(1)}$,
3. successor function: $\|A1\|_{1,1} = \nu$,
4. projection function : $\|Z0\|_{k,1} = \pi_1^{(k)}$ and $\|\bar{C}(\sigma;1)\|_{k,1} = \pi_\sigma^{(k)}$, $\sigma \neq 1$.

Moreover, the class of LOOP computable functions is closed under composition and primitive recursion. This can be shown as in the proof of Theorem 2.12, where subtraction is replaced by the program in 2.40. But the class of primitive recursive functions is the smallest class of functions that is primitively closed. Hence, all primitive recursive functions are LOOP computable. This proves the claim.

Second, claim that each LOOP computable function is primitive recursive. Indeed, for each LOOP program P, let $n(P)$ denote the largest address (or register number) used in P. For integers $m \geq n(P)$ and $0 \leq j \leq m$, consider the functions

$$k_j^{(m+1)}(P) : \mathbb{N}_0^{m+1} \to \mathbb{N}_0 : (x_0, x_1, \ldots, x_m) \mapsto (\pi_j \circ |P|)(x_0, x_1, \ldots, x_m, , 0, 0, \ldots), \tag{2.67}$$

where for each $j \in \mathbb{N}_0$,

$$\pi_j : \Omega \to \mathbb{N}_0 : (\omega_0, \omega_1, \omega_2, \ldots) \mapsto \omega_j. \tag{2.68}$$

The assertion to be shown is a special case of the following assertion: For all LOOP programs P, for all integers $m \geq n(P)$ and $0 \leq j \leq m$, the function $k_j^{(m+1)}(P)$ is primitive recursive. The proof makes use of the inductive definition of LOOP programs.

First, let $P = A\sigma$, $m \geq \sigma$ and $0 \leq j \leq m$. Then

$$k_j^{(m+1)}(P) : \boldsymbol{x} \mapsto \begin{cases} (\nu \circ \pi_j^{(m+1)})(\boldsymbol{x}) & \text{if } j = \sigma, \\ \pi_j^{(m+1)}(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{2.69}$$

Clearly, this function is primitive recursive.

Second, let $P = Z\sigma$, $m \geq \sigma$ and $0 \leq j \leq m$. We have

$$k_j^{(m+1)}(P) : \boldsymbol{x} \mapsto \begin{cases} (c_0^{(1)} \circ \pi_j^{(m+1)})(\boldsymbol{x}) & \text{if } j = \sigma, \\ \pi_j^{(m+1)}(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{2.70}$$

This function is also primitive recursive.

Third, let $P = \bar{C}(\sigma, \tau)$, where $\sigma \neq \tau$ and $\sigma \neq 0 \neq \tau$, $m \geq n(P) = \max\{\sigma, \tau\}$ and $0 \leq j \leq m$. Then

$$k_j^{(m+1)}(P) : \boldsymbol{x} \mapsto \begin{cases} (c_0^{(1)} \circ \pi_j^{(m+1)})(\boldsymbol{x}) & \text{if } j = 0, \\ \pi_\sigma^{(m+1)}(\boldsymbol{x}) & \text{if } j = \tau, \\ \pi_j^{(m+1)}(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{2.71}$$

This function is clearly primitive recursive.

Fourth, let $P = Q; R \in \mathcal{P}_{\text{LOOP}}$. By induction, assume that the assertion holds for Q and R. Let $m \geq n(Q; R) = \max\{n(Q), n(R)\}$ and $0 \leq j \leq m$. Then

$$\begin{aligned} k_j^{(m+1)}(P)(\boldsymbol{x}) &= (\pi_j \circ P)(\boldsymbol{x}, 0, 0, \ldots) \\ &= (\pi_j \circ |R| \circ |Q|)(\boldsymbol{x}, 0, 0, \ldots) \\ &= k_j^{(m+1)}(R)(k_0^{(m+1)}(Q)(\boldsymbol{x}), \ldots, k_m^{(m+1)}(Q)(\boldsymbol{x})), \\ &= k_j^{(m+1)}(R)(k_0^{(m+1)}(Q), \ldots, k_m^{(m+1)}(Q))(\boldsymbol{x}). \end{aligned} \tag{2.72}$$

Thus $k_j^{(m+1)}(P)$ is a composition of primitive recursive functions and hence also primitive recursive.

Finally, let $P = [Q]\sigma \in \mathcal{P}_{\text{LOOP}}$, where Q is a LOOP program in which the address σ is not involved. By induction, assume that the assertion holds for Q. Let $m \geq n([Q]\sigma) = \max\{n(Q), \sigma\}$ and $0 \leq j \leq m$.

First the program $Q; S\sigma$ yields

$$k_j^{(m+1)}(Q; S\sigma) : \boldsymbol{x} \mapsto \begin{cases} k_j^{(m+1)}(Q)(\boldsymbol{x}) & \text{if } j \neq \sigma, \\ (f_{\dot{-}\,1} \circ \pi_j^{(m+1)})(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{2.73}$$

Let $k^{(m+1)}(Q; S\sigma) : \mathbb{N}_0^{m+1} \to \mathbb{N}_0^{m+1}$ denote the product of the $m+1$ functions $k_j^{(m+1)}(Q; S\sigma)$ for $0 \leq j \leq m$. That is,

$$k^{(m+1)}(Q; S\sigma)(\boldsymbol{x}) = (k_0^{(m+1)}(Q; S\sigma)(\boldsymbol{x}), \ldots, k_m^{(m+1)}(Q; S\sigma)(\boldsymbol{x})). \tag{2.74}$$

Let $g : \mathbb{N}_0^{m+2} \to \mathbb{N}_0^{m+1}$ denote the iteration of $k^{(m+1)}(Q; S\sigma)$; that is,

$$g(\boldsymbol{x}, 0) = \boldsymbol{x} \quad \text{and} \quad g(\boldsymbol{x}, y+1) = k^{(m+1)}(Q; S\sigma)(g(\boldsymbol{x}, y)). \tag{2.75}$$

For each index j, $0 \leq j \leq m$, the composition $\pi_j^{(m+1)} \circ g$ is also primitive recursive giving

$$(\pi_j^{(m+1)} \circ g)(\boldsymbol{x}, y) = k_j^{(m+1)}((Q; S\sigma)^y)(\boldsymbol{x}). \tag{2.76}$$

But the register R_σ is never used by the program Q and thus

$$|P|(\omega) = |(Q; S\sigma)\sigma|(\omega) = |(Q; S\sigma)^{\omega_\sigma}|(\omega),\ \omega \in \Omega. \tag{2.77}$$

It follows that the function $k_j^{(m+1)}(P)$ can be obtained from the primitive recursive function $\pi_j^{(m+1)} \circ g$ by transformation of variables, i.e.,

$$k_j^{(m+1)}(P)(\boldsymbol{x}) = (\pi_j^{(m+1)} \circ g)(\boldsymbol{x}, \pi_\sigma^{(m+1)}(\boldsymbol{x})),\ \boldsymbol{x} \in \mathbb{N}_0^{m+1}. \tag{2.78}$$

Thus $k_j^{(m+1)}(P)$ is also primitive recursive. □

3

Partial Recursive Functions

The partial recursive functions form a class of partial functions that are computable in an intuitive sense. They are closely related to the primitive recursive functions and their inductive definition builds upon them. The partial recursive functions are precisely the functions that can be computed by unlimited register machines or GOTO programs. The latter will be useful later on for Gödelization and reduction processes. Computability theory centers around Church's thesis, which states that the partial recursive functions provide a formalization of the notion of computability. This chapter introduces partial recursive functions, GOTO programs, and GOTO computable functions.

3.1 Partial Recursive Functions

The class of partial recursive functions is the basic object of study in computability theory. This class was first investigated by Stephen Cole Kleene (1909-1994) in the 1930s and provides a formalization of the intuitive notion of computability. To this end, a formal analogue of the `while` loop is required. For this, each partial function $f : \mathbb{N}^{k+1} \to \mathbb{N}_0$ is associated with a partial function

$$\mu f : \mathbb{N}_0^k \to \mathbb{N}_0 : \boldsymbol{x} \mapsto \begin{cases} y \text{ if } f(\boldsymbol{x}, y) = 0 \text{ and } f(\boldsymbol{x}, i) \text{ is defined with } f(\boldsymbol{x}, i) \neq 0 \text{ for } 0 \leq i < y, \\ \uparrow \text{ otherwise.} \end{cases} \tag{3.1}$$

The function μf is said to be defined by *(unbounded) minimalization* of f. The domain of the function μf is given by all elements $\boldsymbol{x} \in \mathbb{N}_0^k$ with the property that $f(\boldsymbol{x}, y) = 0$ and $(\boldsymbol{x}, i) \in \mathrm{dom}(f)$ for all $0 \leq i \leq y$. It is clear that in the context of programming, unbounded minimalization corresponds to a `while` loop (Algorithm 3.1).

Examples 3.1.

- The minimalization function μf may be partial even if f is total: The function $f(x, y) = (x + y) \dot{-} 3$ is total, while its minimalization μf is partial with $\mathrm{dom}(\mu f) = \{0, 1, 2, 3\}$ and $\mu f(0) = \mu f(1) = \mu f(2) = \mu f(3) = 0$.
- The minimalization function μf may be total even if f is partial: Take the partial function $f(x, y) = x - y$ if $y \leq x$ and $f(x, y)$ undefined if $y > x$. The corresponding minimalization $\mu f(x) = x$ is total with $\mathrm{dom}(\mu f) = \mathbb{N}_0$. ♢

Algorithm 3.1 Minimalization of f.

```
Require: $\boldsymbol{x} \in \mathbb{N}_0^n$
  $y \leftarrow -1$
  repeat
    $y \leftarrow y + 1$
    $z \leftarrow f(\boldsymbol{x}, y)$
  until $z = 0$
  return $y$
```

The class $\mathcal{R}$ of *partial recursive functions* over $\mathbb{N}_0$ is inductively defined:

- $\mathcal{R}$ contains all the base functions.
- If partial functions $g : \mathbb{N}_0^k \to \mathbb{N}_0$ and $h_i : \mathbb{N}_0^n \to \mathbb{N}_0$, $1 \leq i \leq k$, belong to $\mathcal{R}$, the composite function $f = g(h_1, \ldots, h_k) : \mathbb{N}_0^n \to \mathbb{N}_0$ is in $\mathcal{R}$.
- If partial functions $g : \mathbb{N}_0^n \to \mathbb{N}_0$ and $h : \mathbb{N}_0^{n+2} \to \mathbb{N}_0$ lie in $\mathcal{R}$, the primitive recursion $f = \mathrm{pr}(g, h) : \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ is contained in $\mathcal{R}$.
- If a partial function $f : \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ is in $\mathcal{R}$, the partial function $\mu f : \mathbb{N}_0^n \to \mathbb{N}_0$ obtained by minimalization belongs to $\mathcal{R}$.

Thus the class $\mathcal{R}$ consists of all partial recursive functions obtained from the base functions by finitely many applications of composition, primitive recursion, and minimalization. In particular, each total partial recursive function is called a *recursive function*, and the subclass of all total partial recursive functions is denoted by $\mathcal{T}$. It is clear that the class $\mathcal{P}$ of primitive recursive functions is a subclass of the class $\mathcal{T}$ of recursive functions.

Theorem 3.2. *Each partial recursive function is URM computable.*

Proof. It is sufficient to show that for each URM computable function $f : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ the corresponding minimalization μf is URM computable. For this, let P_f be an URM program for the function f. This program can be modified to provide an URM program P_f' with the following property:

$$|P_f'|(0, \boldsymbol{x}, y, 0, 0, \ldots) = \begin{cases} (0, \boldsymbol{x}, y, f(\boldsymbol{x}, y), 0, 0, \ldots) & \text{if } (\boldsymbol{x}, y) \in \mathrm{dom}(f), \\ \uparrow & \text{otherwise.} \end{cases} \tag{3.2}$$

Consider the URM program

$$P_{\mu f} = P_f';\ (Ak+1; (Sk+2)k+2; P_f')k+2;\ (S1)1; \ldots; (Sk)k; R_{k+1,1}. \tag{3.3}$$

The first block P_f' provides the computation in (3.2) for $y = 0$. The second block $(Ak+1; (Sk+2)k+2; P_f')k+2$ calculates iteratively (3.2) for increasing values of y. This iteration stops when the function value becomes 0; in this case, the subsequent blocks reset the registers $R_1, \ldots, R_k$ to 0 and store the argument y in the first register. Otherwise, the program runs forever. It follows that the program $P_{\mu f}$ computes the minimalization of f, i.e., $\|P_{\mu f}\|_{k,1} = \mu f$. □

3.2 GOTO Programs

GOTO programs offer another way to formalize the notion of computability. They are closely related to programs in BASIC or FORTRAN.

First, define the *syntax* of GOTO programs. For this, let $V = \{x_\sigma \mid \sigma \in \mathbb{N}\}$ be a set of variables. The *instructions* of a GOTO program are the following:

- Incrementation:

$$(l, x_\sigma \leftarrow x_\sigma + 1, m), \quad l, m \in \mathbb{N}_0,\ x_\sigma \in V, \tag{3.4}$$

- Decrementation:

$$(l, x_\sigma \leftarrow x_\sigma - 1, m), \quad l, m \in \mathbb{N}_0,\ x_\sigma \in V, \tag{3.5}$$

- Branching:

$$(l, \texttt{if}\ x_\sigma = 0, k, m), \quad k, l, m \in \mathbb{N}_0,\ x_\sigma \in V. \tag{3.6}$$

The first component of an instruction, denoted by l, is called a *label*, the third component of an instruction $(l, x_\sigma+, m)$ or $(l, x_\sigma-, m)$, denoted by m, is termed *next label*, and the last two components of an instruction $(l, \texttt{if}\ x_\sigma = 0, k, m)$, denoted by k and m, are called *bifurcation labels.*

A *GOTO program* is given by a finite sequence of GOTO instructions

$$P = s_0; s_1; \ldots; s_q, \tag{3.7}$$

such that there is a unique instruction s_i which has the label $\lambda(s_i) = 0$, $0 \leq i \leq q$, and different instructions have distinct labels, i.e., for $0 \leq i < j \leq q$, $\lambda(s_i) \neq \lambda(s_j)$.

In the following, let $\mathcal{P}_{\text{GOTO}}$ denote the class of all GOTO programs. Moreover, for each GOTO program P, let $V(P)$ depict the set of variables occurring in P and $L(P) = \{\lambda(s_i) \mid 0 \leq i \leq q\}$ provide the set of labels in P.

A GOTO program $P = s_0; s_1; \ldots; s_q$ is called *standard* if the i-th instruction s_i carries the label $\lambda(s_i) = i$, $0 \leq i \leq q$.

Example 3.3. A standard GOTO program P_+ for the addition of two natural numbers is the following:

0	$\texttt{if}\ x_2 = 0$	3	1
1	$x_1 \leftarrow x_1 + 1$	2	
2	$x_2 \leftarrow x_2 - 1$	0	

The set of occurring variables is $V(P_+) = \{x_1, x_2\}$ and the set of labels is $L(P_+) = \{0, 1, 2\}$. ♢

Second, define the *semantics* of GOTO programs. For this, the idea is to run a GOTO program on an URM. To this end, the variable x_σ in V is assigned the register R_σ of the URM for all $\sigma > 0$. Moreover, the register R_0 serves as an instruction counter containing the label of the next instruction to be carried out. At the beginning, the instruction counter is set to 0 such that the execution starts with the instruction having label 0. The instruction $(l, x_\sigma \leftarrow x_\sigma + 1, m)$ increments the content of the register R_σ, the instruction $(l, x_\sigma \leftarrow x_\sigma - 1, m)$ decrements the content of the register R_σ provided that it contains a number greater than zero, and the conditional statement $(l, \texttt{if}\ x_\sigma = 0, k, m)$ provides a jump to the statement with label k if the content of register R_σ is 0; otherwise, a jump is performed to the statement with label m. The execution of a GOTO program *terminates* if the instruction counter does not correspond to an instruction label.

More specifically, let P be a GOTO program and let k be a number. Define the partial function

$$\|P\|_{k,1} = \beta_1 \circ R_P \circ \alpha_k, \tag{3.8}$$

where α_k and β_1 are total functions given by

$$\alpha_k : \mathbb{N}_0^k \to \mathbb{N}_0^{n+1} : (x_1, x_2, \ldots, x_k) \mapsto (0, x_1, x_2, \ldots, x_k, 0, 0, \ldots, 0) \tag{3.9}$$

and

$$\beta_1 : \mathbb{N}_0^{n+1} \to \mathbb{N}_0 : (x_0, x_1, \ldots, x_n) \mapsto x_1. \tag{3.10}$$

The function α_k loads the registers with the arguments and the function β_1 reads out the result. Note that the number n needs to be chosen large enough to provide sufficient workspace for the computation; that is,

$$n \geq \max\{k, \max\{\sigma \mid x_\sigma \in V(P)\}\}. \tag{3.11}$$

Finally, the function $R_P : \mathbb{N}_0^{n+1} \to \mathbb{N}_0^{n+1}$ providing the semantics of the program P will be formally described. For this, let $P = s_0; s_1; \ldots; s_q$ be a GOTO program. Each element $\boldsymbol{z} = (z_0, z_1, \ldots, z_n)$ in $\mathbb{N}_0^{n+1}$ is called a *configuration*. We say that the configuration $\boldsymbol{z}' = (z_0', z_1', \ldots, z_n')$ is *reached in one step* from the configuration $\boldsymbol{z} = (z_0, z_1, \ldots, z_n)$, written $\boldsymbol{z} \vdash_P \boldsymbol{z}'$, if z_0 is a label in P, say $z_0 = \lambda(s_i)$ for some $0 \leq i \leq q$, and

- if $s_i = (z_0, x_\sigma \leftarrow x_\sigma + 1, m)$,

$$\boldsymbol{z}' = (m, z_1, \ldots, z_{\sigma-1}, z_\sigma + 1, z_{\sigma+1}, \ldots, z_n), \tag{3.12}$$

- if $s_i = (z_0, x_\sigma \leftarrow x_\sigma - 1, m)$,

$$\boldsymbol{z}' = (m, z_1, \ldots, z_{\sigma-1}, z_\sigma \dot{-} 1, z_{\sigma+1}, \ldots, z_n), \tag{3.13}$$

- if $s_i = (z_0, \texttt{if}\ x_\sigma = 0, k, m)$,

$$\boldsymbol{z}' = \begin{cases} (k, z_1, \ldots, z_n) & \text{if } z_\sigma = 0, \\ (m, z_1, \ldots, z_n) & \text{otherwise.} \end{cases} \tag{3.14}$$

The configuration $\boldsymbol{z}'$ is called the *successor configuration* of $\boldsymbol{z}$. Moreover, if z_0 is not a label in P, there is no successor configuration of $\boldsymbol{z}$ and the successor configuration is *undefined*. The process to move from one configuration to the next one can be described by the *one-step function* $E_P : \mathbb{N}_0^{n+1} \to \mathbb{N}_0^{n+1}$ defined as

$$E_P : \boldsymbol{z} \mapsto \begin{cases} \boldsymbol{z}' & \text{if } \boldsymbol{z} \vdash_P \boldsymbol{z}', \\ \boldsymbol{z} & \text{otherwise.} \end{cases} \tag{3.15}$$

The function E_P is given by cases and has the following property.

Proposition 3.4. *For each GOTO program P and each number $n \geq \max\{\sigma \mid x_\sigma \in V(P)\}$, the one-step function $E_P : \mathbb{N}_0^{n+1} \to \mathbb{N}_0^{n+1}$ is primitive recursive in the sense that for each $0 \leq j \leq n$, the coordinate function $\pi_j^{(n+1)} \circ E_P$ is primitive recursive.*

The execution of the GOTO program P can be represented by a sequence $\boldsymbol{z}_0, \boldsymbol{z}_1, \boldsymbol{z}_2, \ldots$ of configurations such that $\boldsymbol{z}_i \vdash_P \boldsymbol{z}_{i+1}$ for all $i \geq 0$. During this process, a configuration $\boldsymbol{z}_t$ may eventually be reached whose label z_0 does not belong to $L(P)$. In this case, the program P terminates. However, such an event may eventually not happen. In this way, the *runtime function* of P is the partial function $Z_P : \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ given by

$$Z_P : \boldsymbol{z} \mapsto \begin{cases} \min\{t \in \mathbb{N}_0 \mid (\pi_0^{(n+1)} \circ E_P^t)(\boldsymbol{z}) \notin L(P)\} & \text{if } \{\ldots\} \neq \emptyset, \\ \uparrow & \text{otherwise.} \end{cases} \tag{3.16}$$

The runtime function may not be primitive recursive as it corresponds to an unbounded search process.

Proposition 3.5. *For each GOTO program P, the runtime function Z_P is partial recursive.*

Proof. By Proposition 3.4, the one-step function E_P is primitive recursive. It follows that the iteration of E_P is also primitive recursive:

$$E_P' : \mathbb{N}_0^{n+2} \to \mathbb{N}_0^{n+1} : (\boldsymbol{z}, t) \mapsto E_P^t(\boldsymbol{z}). \tag{3.17}$$

Moreover, the set $L(P)$ is finite and so the characteristic function $\chi_{L(P)}$ is primitive recursive. Therefore, the subsequent function is also primitive recursive:

$$E_P'' : \mathbb{N}_0^{n+2} \to \mathbb{N}_0^n : (\boldsymbol{z}, t) \mapsto (\chi_{L(P)} \circ \pi_0^{(n+1)} \circ E_P')(\boldsymbol{z}, t). \tag{3.18}$$

This function has the following property:

$$E_P''(\boldsymbol{z}, t) = \begin{cases} 1 \text{ if } E_P^t(\boldsymbol{z}) = (z_0', z_1', \ldots, z_n') \text{ and } z_0' \in L(P), \\ 0 \text{ otherwise.} \end{cases} \tag{3.19}$$

By definition, $Z_P = \mu E_P''$ and so the result follows. □

The *residual step function* of the GOTO program P is given by the partial function $R_P : \mathbb{N}_0^{n+1} \to \mathbb{N}_0^{n+1}$ that maps an initial configuration to the final configuration of the computation if any:

$$R_P(\boldsymbol{z}) = \begin{cases} E_P'(\boldsymbol{z}, Z_P(\boldsymbol{z})) & \text{if } \boldsymbol{z} \in \operatorname{dom}(Z_P), \\ \uparrow & \text{otherwise.} \end{cases} \tag{3.20}$$

Proposition 3.6. *For each GOTO program P, the function R_P is partial recursive.*

Proof. For each configuration $\boldsymbol{z} \in \mathbb{N}_0^{n+1}$, $R_P(\boldsymbol{z}) = E_P'(\boldsymbol{z}, (\mu E_P'')(\boldsymbol{z}))$. Thus R_P is a composition of partial recursive functions and hence is itself partial recursive. □

3.3 GOTO Computable Functions

A partial function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is called *GOTO computable* if there is a GOTO program P that computes f in the sense that

$$f = \|P\|_{k,1}. \tag{3.21}$$

Let $\mathcal{F}_{\mathrm{GOTO}}$ denote the class of all (partial) GOTO computable functions and let $\mathcal{T}_{\mathrm{GOTO}}$ depict the class of all total GOTO computable functions.

Theorem 3.7. *Each GOTO computable function is partial recursive, and each total GOTO computable function is recursive.*

Proof. If f is GOTO computable, there is a GOTO program such that $f = \|P\|_{k,1} = \beta_1 \circ R_P \circ \alpha_k$. But the functions β_1 and α_k are primitive recursive and the function R_P is partial recursive. Thus the composition is partial recursive. In particular, if f is total, then R_P is also total and so f is recursive. □

Example 3.8. The GOTO program P_+ for the addition of two natural numbers in Example 3.3 gives rise to the following configurations ($n = 2$) if $x_2 > 0$:

$(0, x_1, x_2)$	0	`if` $x_2 = 0$	3	1
$(1, x_1, x_2)$	1	$x_1 \leftarrow x_1 + 1$	2	
$(2, x_1 + 1, x_2)$	2	$x_2 \leftarrow x_2 - 1$	0	
$(0, x_1 + 1, x_2 - 1)$	3	`if` $x_2 = 0$	3	1
...		...		

Clearly, this program provides the addition function $\|P_+\|_{2,1}(x_1, x_2) = x_1 + x_2$. ♢

Finally, it will be shown that URM programs can be translated into GOTO programs.

Theorem 3.9. *Each URM computable function is GOTO computable.*

Proof. Claim that each URM program P can be compiled into a GOTO program $\phi(P)$ such that both compute the same function, i.e., $\|P\|_{k,1} = \|\phi(P)\|_{k,1}$ for all $k \in \mathbb{N}$. Indeed, let P be an URM program. We may assume that it does not make use of the register R_0. Write P as a string $P = \tau_0 \tau_1 \ldots \tau_q$, where each substring (token) τ_i is of the form ”$A\sigma$”, ”$S\sigma$”, ”(” or ”)σ” with $\sigma \in Z \setminus \{0\}$. Note that each opening parenthesis ”(” corresponds to a unique closing parenthesis ”)σ”.

Replace each string τ_i by a GOTO instruction s_i as follows:

- If $\tau_i =$ ”$A\sigma$”, put
$$s_i = (i, x_\sigma \leftarrow x_\sigma + 1, i + 1),$$
- if $\tau_i =$ ”$S\sigma$”, set
$$s_i = (i, x_\sigma \leftarrow x_\sigma - 1, i + 1),$$
- if $\tau_i =$ ”(” and $\tau_j =$ ”)σ” is the corresponding closing parenthesis, define
$$s_i = (i, \texttt{if } x_\sigma = 0, j + 1, i + 1),$$
- if $\tau_i =$ ”)σ” and $\tau_j =$ ”(” is the associated opening parenthesis, put
$$s_i = (i, \texttt{if } x_\sigma = 0, i + 1, j + 1).$$

In this way, a GOTO program $\phi(P) = s_0; s_1; \ldots; s_q$ is established that has the required property $\|P\|_{k,1} = \|\phi(P)\|_{k,1}$, as claimed. □

Example 3.10. The URM program $P = (A1; S2)2$ provides the addition of two natural numbers. Its translation into a GOTO program requires first to identify the substrings:

$$\tau_0 = \text{”(”},\ \tau_1 = \text{”}A1\text{”},\ \tau_2 = \text{”}S2\text{”},\ \tau_3 = \text{”)2”}.$$

These substrings give rise to the following GOTO program:

```
0  if $x_2 = 0$            4  1
1  $x_1 \leftarrow x_1 + 1$  2
2  $x_2 \leftarrow x_2 - 1$  3
3  if $x_2 = 0$            4  1
```

♢

3.4 GOTO-2 Programs

GOTO-2 programs are GOTO programs with two variables x_1 and x_2. Claim that each URM program can be simulated by an appropriate GOTO-2 program. For this, the set of states Ω of an URM is encoded by using the sequence of primes $(p_0, p_1, p_2, \ldots)$. To this end, define the function $G : \Omega \to \mathbb{N}_0$ that assigns to each state $\omega = (\omega_0, \omega_1, \omega_2, \ldots) \in \Omega$ the natural number

$$G(\omega) = p_0^{\omega_0} p_1^{\omega_1} p_2^{\omega_2} \ldots. \tag{3.22}$$

Clearly, this function is primitive recursive. The inverse functions $G_i : \mathbb{N}_0 \to \mathbb{N}_0$, $i \in \mathbb{N}_0$, are given by

$$G_i(x) = (x)_i, \tag{3.23}$$

where $(x)_i$ is the exponent of p_i in the prime factorization of x if $x > 0$. Define $G_i(0) = 0$ for all $i \in \mathbb{N}_0$. The functions G_i are primitive recursive by Proposition 2.38.

Claim that for each URM program P, there is a GOTO-2 program $\overline{P}$ with the same semantics; that is, for all states $\omega, \omega' \in \Omega$,

$$|P|(\omega) = \omega' \quad \Longleftrightarrow \quad |\overline{P}|(0, G(\omega)) = (0, G(\omega')). \tag{3.24}$$

To see this, first consider the GOTO-2 programs $M(k)$, $D(k)$, and $T(k)$, $k \in \mathbb{N}$, which have the following properties:

$$|M(k)|(0, x) = (0, k \cdot x), \tag{3.25}$$

$$|D(k)|(0, k \cdot x) = (0, x), \tag{3.26}$$

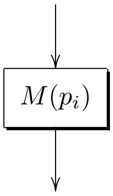
$$|T(k)|(0, x) = \begin{cases} (1, x) & \text{if } k \text{ divides } x, \\ (0, x) & \text{otherwise.} \end{cases} \tag{3.27}$$

For instance, the GOTO-2 program $M(k)$ can be implemented by two consecutive loops: Initially, put $x_1 = 0$ and $x_2 = x$. In the first loop, x_2 is decremented, while in each decremention step, x_1 is incremented k times. After this loop, $x_1 = k \cdot x$ and $x_2 = 0$. In the second loop, x_2 is incremented and x_1 is decremented such that upon termination, $x_1 = 0$ and $x_2 = k \cdot x$. This can be implemented by the following GOTO-2 program in standard form:

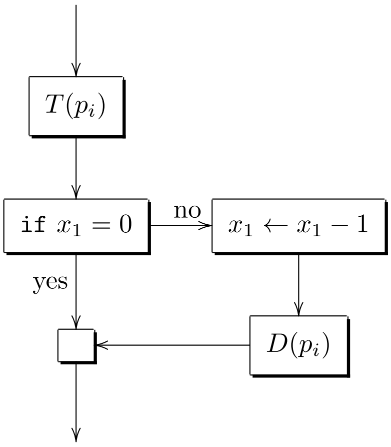
```
$0$      if $x_2 = 0$              $k+2$  $1$
$1$      $x_2 \leftarrow x_2 - 1$  $2$
$2$      $x_1 \leftarrow x_1 + 1$  $3$
         $\ldots$
$k+1$    $x_1 \leftarrow x_1 + 1$  $0$
$k+2$    if $x_1 = 0$              $k+5$  $k+3$
$k+3$    $x_2 \leftarrow x_2 + 1$  $k+4$
$k+4$    $x_1 \leftarrow x_1 - 1$  $k+2$
```

The other two kinds of programs can be analogously realized as GOTO-2 programs in standard form.

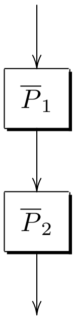
The assignment $P \mapsto \overline{P}$ will be established by making use of the inductive definition of URM programs. For this, flow diagrams will be employed to simplify the notation. First, the program $\overline{Ai}$ can be realized by the flow chart



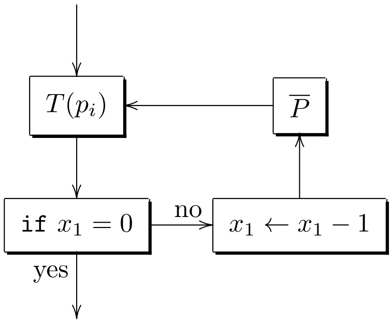
Second, the program $\overline{Si}$ is given by the flow diagram



Third, the program $\overline{P_1; P_2}$ can be decipted by the flow chart



Finally, the program $\overline{(P)i}$ can be represented by the flow diagram



All these GOTO-2 programs can be realized as standard GOTO programs.

Example 3.11. The URM program $P_+ = (A1; S2)2$ gets compiled into the following (standard) GOTO-2 program in pseudo code:

```
0 : $T(p_2)$
1 : if $x_1 = 0$ goto 9
2 : $x_1 \leftarrow x_1 - 1$
3 : $M(p_1)$
4 : $T(p_2)$
5 : if $x_1 = 0$ goto 8
6 : $x_1 \leftarrow x_1 - 1$
7 : $D(p_2)$
8 : goto 0
9 :
```

$\diamondsuit$

By using the inductive definition of URM programs, the assignment $P \mapsto \overline{P}$ is well-defined. The computation of a function $f : \mathbb{N}_0^k \to \mathbb{N}_0^m$ by a GOTO-2 program requires to load the registers with the initial values and to identify the result. For this, define the primitive recursive functions

$$\hat{\alpha}_k : \mathbb{N}_0^k \to \mathbb{N}_0^2 : (x_1, \ldots, x_k) \mapsto (0, G(0, x_1, \ldots, x_k, 0, \ldots)) \tag{3.28}$$

and

$$\hat{\beta}_m : \mathbb{N}_0^2 \to \mathbb{N}_0^m : (a, b) \mapsto (G_1(b), \ldots, G_m(b)). \tag{3.29}$$

Proposition 3.12. *Each URM program P is (semantically) equivalent to the associated GOTO-2 program $\overline{P}$ in the sense that for all $k, m \in \mathbb{N}_0$:*

$$\|P\|_{k,m} = \hat{\beta}_m \circ R_{\overline{P}} \circ \hat{\alpha}_k. \tag{3.30}$$

3.5 Church's Thesis

Our attempts made so far to formalize the notion of computability are equivalent in the following sense.

Theorem 3.13. *The class of partial recursive functions equals the class of URM computable functions and the class of GOTO computable functions. In particular, the class of recursive functions is equal to the class of total URM computable functions and to the class of total GOTO computable functions.*

Proof. By Theorem 3.9, each URM computable function is GOTO computable and by Theorem 3.7, each GOTO computable function is partial recursive. On the other hand, by Theorem 3.2, each partial recursive function is URM computable. A similar statement holds for total functions. □

This result has led scientists to believe that the concept of computability is accurately characterized by the class of partial recursive functions. The *Church thesis* proposed by Alonso Church (1903-1995) in the 1930s states that the class of computable partial functions (in the intuitive sense) coincides with the class of partial recursive functions, equivalently, with the class of URM computable functions and the class of GOTO computable functions. Church's thesis characterizes the nature of computation and cannot be formally proved. Nevertheless, it has reached universal acceptance. Church's thesis is often practically used in the sense that if a (partial) function is intuitively computable, it is assumed to be partial recursive. In this way, the thesis may lead to more intuitive and less rigorous proofs (see Theorem 5.13).

4

A Recursive Function

The primitive recursive functions are total and computable. The Ackermann function, named after the German mathematician Wilhelm Ackermann (1986-1962), was the first-discovered example of a total computable function that is not primitive recursive.

4.1 Small Ackermann Functions

The small Ackermann functions form an interesting class of primitive recursive functions. They can be used to define the "big" Ackermann function and provide upper bounds on the runtime of LOOP programs. The latter property allows to show that the hierarchy of LOOP-computable functions is strict.

Define a sequence (B_n) of monadic total functions inductively as follows:

$$B_0 : x \mapsto \begin{cases} 1 & \text{if } x = 0, \\ 2 & \text{if } x = 1, \\ x + 2 & \text{otherwise}, \end{cases} \tag{4.1}$$

and

$$B_{n+1} : x \mapsto B_n^x(1), \quad x, n \in \mathbb{N}_0, \tag{4.2}$$

where B_n^x denotes the x-fold iteration of the function B_n. For each $n \geq 0$, the function B_n is called the *n-th small Ackermann function.*

Proposition 4.1. *The small Ackermann functions have the following properties for all $x, n, p \in \mathbb{N}_0$:*

$$B_1(x) = \begin{cases} 1 & \textit{if } x = 0, \\ 2x & \textit{otherwise}, \end{cases} \tag{4.3}$$

$$B_2(x) = 2^x, \tag{4.4}$$

$$B_3(x) = \begin{cases} 1 & \textit{if } x = 0, \\ 2^{B_3(x-1)} & \textit{otherwise}, \end{cases} \tag{4.5}$$

$$x < B_n(x), \tag{4.6}$$

$$\begin{aligned}
B_n(x) &< B_n(x+1), && (4.7)\\
B_0^p(x) &\leq B_1^p(x), && (4.8)\\
B_n(x) &\leq B_{n+1}(x), && (4.9)\\
B_n^p(x) &< B_n^p(x+1), && (4.10)\\
B_n^p(x) &< B_n^{p+1}(x), && (4.11)\\
B_n^p(x) &\leq B_{n+1}^p(x), && (4.12)\\
2^{p+1}x &\leq B_1^{p+1}(x), && (4.13)\\
2B_n^p(x) &\leq B_n^{p+1}(x), \quad n \geq 1. && (4.14)
\end{aligned}$$

Proposition 4.2. *The small Ackermann functions B_n, $n \in \mathbb{N}_0$, are primitive recursive.*

Proof. The function B_0 is defined by cases:

$$B_0(x) = \begin{cases} (\nu \circ c_0^{(1)})(x) & \text{if } x = 0, \\ (\nu \circ \nu \circ c_0^{(1)})(x) & \text{if } |x \dot{-} 1| = 0, \\ (\nu \circ \nu \circ \pi_1^{(1)})(x) & \text{otherwise.} \end{cases} \tag{4.15}$$

By Eq. (4.3), the function B_1 follows the primitive recursive scheme

$$B_1(0) = 1, \tag{4.16}$$

$$B_1(x+1) = B_1(x) \cdot \operatorname{sgn}(x) + 2, \quad x \in \mathbb{N}_0. \tag{4.17}$$

Finally, if $n \geq 2$, B_n is given by the primitive recursive scheme

$$B_n(0) = 1, \tag{4.18}$$

$$B_n(x+1) = B_{n-1}(B_n(x)), \quad x \in \mathbb{N}_0. \tag{4.19}$$

By induction, the small Ackermann functions are primitive recursive. □

The $n+1$-th small Ackermann function grows faster than any power of the n-th small Ackermann function in the following sense.

Proposition 4.3. *For all number n and p, there is a number x_0 such that for all numbers $x \geq x_0$,*

$$B_n^p(x) < B_{n+1}(x). \tag{4.20}$$

Proof. Let $n = 0$. For each $x \geq 2$, $B_0^p(x) = x + 2p$. On the other hand, for each $x \geq 1$, $B_1(x) = 2x$. Put $x_0 = 2p + 1$. Then for each $x \geq x_0$,

$$B_0^p(x) = x + 2p \leq 2x = B_1(x).$$

Let $n > 0$. First, let $p = 0$. Then for each $x \geq 0$,

$$B_n^0(x) = x < B_n(x) \leq B_{n+1}(x)$$

by (4.6) and (4.9). Second, let $p > 0$ and assume that $B_n^p(x) < B_{n+1}(x)$ holds for all $x \geq x_0'$. Put $x_0 = x_0' + 5$. Then

$$\begin{aligned}
B_n^{p+1}(x) &< B_n^{p+1}(2 \cdot (x \mathbin{\dot{-}} 2)), \quad x \geq 5, \text{ by (4.10)},\\
&= B_n^{p+1}(B_1(x \mathbin{\dot{-}} 2))\\
&\leq B_n^{p+1}(B_n(x \mathbin{\dot{-}} 2)), \quad \text{by (4.12)},\\
&= B_n^{p+2}(x \mathbin{\dot{-}} 2)\\
&= B_n^2(B_n^p(x \mathbin{\dot{-}} 2))\\
&< B_n^2(B_{n+1}(x \mathbin{\dot{-}} 2)), \quad \text{by induction, } x \geq x_0' + 2,\\
&= B_n^2(B_n^{x \mathbin{\dot{-}} 2}(1))\\
&= B_n^x(1)\\
&= B_{n+1}(x), \quad x \geq 2.
\end{aligned}$$

□

Proposition 4.4. *For each $n \geq 1$, the n-th small Ackermann function B_n is LOOP-n computable.*

Proof. First, define the LOOP-1 program

$$P_1 = \bar{C}(1;2); \bar{C}(1;3); Z1; A1; [Z1]3; [A1; A1]2. \tag{4.21}$$

For each input x, the program evaluates as follows:

0	1	2	3	4	…	registers
0	x	0	0	0	…	init
0	x	x	0	0	…	
0	x	x	x	0	…	
0	0	x	x	0	…	
0	1	x	x	0	…	
0	1	0	0	0	…	$x = 0$ end
0	$2x$	0	0	0	…	$x \neq 0$ end

By (4.3), the program satisfies $\|P_1\|_{1,1} = B_1$.

Suppose there is a normal LOOP-n program P_n that computes B_n; that is, $\|P_n\|_{1,1} = B_n$, for $n \geq 1$. Put $m = n(P_n) + 1$ and consider the LOOP-$n+1$ program

$$P_{n+1} = [Am]1; A1; [P_n]m. \tag{4.22}$$

Note that the register R_m is unused in the program P_n. The program P_{n+1} computes $B_n^x(1)$ as follows:

0	1	2	3	4	…	m	…	registers
0	x	0	0	0	…	0	…	init
0	0	0	0	0	…	x	…	
0	1	0	0	0	…	x	…	
0	$B_n^x(1)$	0	0	0	…	0	…	end

But $B_{n+1}(x) = B_n^x(1)$ and so $\|P_{n+1}\|_{1,1} = B_{n+1}$. □

4.2 Runtime of LOOP Programs

The LOOP programs will be extended in a way that they do not only perform their task but simultaneously compute their runtime. This will finally allow us to show that Ackermann's function is not primitive recursive.

For this, assume that the LOOP programs do not make use the register R_0. This is not an essential restriction since the register R_0 can always be replaced by an unused register via a transformation of variables. The register R_0 will then be employed to calculate the runtime of a LOOP program. To this end, the objective is to assign to each LOOP program P another LOOP program $\gamma(P)$ that performs the computation of P and simultaneously calculates the runtime of P. The runtime of a program is essentially determined by the number of elementary operations:

$$\gamma(A\sigma) = A\sigma; A0, \quad \sigma \neq 0, \tag{4.23}$$

$$\gamma(Z\sigma) = Z\sigma; A0, \quad \sigma \neq 0, \tag{4.24}$$

$$\gamma(\bar{C}(\sigma;\tau)) = \bar{C}(\sigma;\tau); A0, \quad \sigma \neq \tau,\ \sigma \neq 0 \neq \tau, \tag{4.25}$$

$$\gamma(P;Q) = \gamma(P); \gamma(Q), \tag{4.26}$$

$$\gamma([P]\sigma) = [\gamma(P)]\sigma, \quad \sigma \neq 0. \tag{4.27}$$

This definition immediately leads to the following

Proposition 4.5. *Let $n \geq 0$.*

- *If P is a LOOP-n program, $\gamma(P)$ is also LOOP-n program.*
- *Let P be a LOOP-n program and let $|\gamma(P)| \circ \alpha_k(\boldsymbol{x}) = (\omega_n)$, where $\boldsymbol{x} \in \mathbb{N}_0^k$. Then $\omega_1 = \|P\|_{k,1}(\boldsymbol{x})$ is the result of the computation of P and ω_0 is the number of elementary operations made during the computation of P.*

The *runtime program* of a LOOP program P is the LOOP program P' given by

$$P' = \gamma(P); \bar{C}(0;1). \tag{4.28}$$

The corresponding function $\|P'\|_{k,1}$ with $k \geq n(P)$ is called the *runtime function* of P.

Proposition 4.6. *If P is a LOOP-n program, P' is a LOOP-n program with the property*

$$\|P'\|_{k,1} = \gamma(P), \quad k \geq n(P). \tag{4.29}$$

Example 4.7. The program $S1$ is given by the LOOP-1 program

$$P = \bar{C}(1;3); [\bar{C}(2;1); A2]3. \tag{4.30}$$

The corresponding runtime program is

$$P' = \bar{C}(1;3); A0; [\bar{C}(2;1); A0; A2; A0]3; \bar{C}(0;1). \tag{4.31}$$

♢

Next, consider a function $\lambda : \mathcal{P}_{\text{LOOP}} \to \mathbb{N}_0$ that assigns to each LOOP program a measure of complexity. Here elementary operations have unit complexity, while loops contribute with a higher complexity:

$$\begin{aligned}
\lambda(A\sigma) &= 1, \quad \sigma \in \mathbb{N}_0, && (4.32)\\
\lambda(Z\sigma) &= 1, \quad \sigma \in \mathbb{N}_0, && (4.33)\\
\lambda(\bar{C}(\sigma;\tau)) &= 1, \quad \sigma \neq \tau,\, \sigma,\tau \in \mathbb{N}_0, && (4.34)\\
\lambda(P;Q) &= \lambda(P) + \lambda(Q), \quad P,Q \in \mathcal{P}_{\text{LOOP}}, && (4.35)\\
\lambda([P]\sigma) &= \lambda(P) + 2, \quad P \in \mathcal{P}_{\text{LOOP}},\, \sigma \in \mathbb{N}_0. && (4.36)
\end{aligned}$$

Note that for each LOOP program P, the *complexity measure* $\lambda(P)$ is the number of LOOP-0 subprograms of P plus twice the number of iterations of P.

Example 4.8. The LOOP program $P = \bar{C}(1;3);[\bar{C}(2;1);A2]3$ has the complexity

$$\begin{aligned}
\lambda(P) &= \lambda(\bar{C}(1;3)) + \lambda([\bar{C}(2;1);A2]3)\\
&= 1 + \lambda(\bar{C}(2;1);A2) + 2\\
&= 3 + \lambda(\bar{C}(2;1)) + \lambda(A2)\\
&= 5.
\end{aligned}$$

♢

A k-ary total function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is said to be *bounded* by a monadic total function $g : \mathbb{N}_0 \to \mathbb{N}_0$ if for each $\boldsymbol{x} \in \mathbb{N}_0^k$,

$$f(\boldsymbol{x}) \leq g(\max(\boldsymbol{x})), \tag{4.37}$$

where $\max(\boldsymbol{x}) = \max\{x_1,\ldots,x_k\}$ if $\boldsymbol{x} = (x_1,\ldots,x_k) \in \mathbb{N}_0^k$.

Proposition 4.9. *For each LOOP-n program P and each input $\boldsymbol{x} \in \mathbb{N}_0^k$ with $k \geq n(P)$,*

$$\max(\|P\|_{k,k}(\boldsymbol{x})) \leq B_n^{\lambda(P)}(\max(\boldsymbol{x})). \tag{4.38}$$

Proof. First, let P be a LOOP-0 program; that is, $P = P_1;P_2;\ldots;P_m$, where each block P_i is an elementary operation, $1 \leq i \leq m$. Then

$$\max(\|P\|_{k,k}(\boldsymbol{x})) \leq m + \max(\boldsymbol{x}) = \lambda(P) + \max(\boldsymbol{x}) \leq B_0^{\lambda(P)}(\max(\boldsymbol{x})). \tag{4.39}$$

Suppose the assertion holds for LOOP-n programs. Let P be a LOOP program of the form $P = Q;R$, where Q and R are LOOP-$n+1$ programs. Then for $k \geq \max\{n(Q),n(R)\}$,

$$\begin{aligned}
\max(\|Q;R\|_{k,k}(\boldsymbol{x})) &= \max(\|R\|_{k,k}(\|Q\|_{k,k}(\boldsymbol{x})))\\
&\leq B_{n+1}^{\lambda(R)}(\max(\|Q\|_{k,k}(\boldsymbol{x}))),\ \text{by induction},\\
&\leq B_{n+1}^{\lambda(R)}(B_{n+1}^{\lambda(Q)}(\max(\boldsymbol{x}))),\ \text{by induction}, \qquad (4.40)\\
&= B_{n+1}^{\lambda(R)+\lambda(Q)}(\max(\boldsymbol{x}))\\
&= B_{n+1}^{\lambda(P)}(\max(\boldsymbol{x})).
\end{aligned}$$

Finally, let P be a LOOP program of the form $P = [Q]\sigma$, where Q is a LOOP-n program. Then for $k \geq \max\{n(Q), \sigma\}$,

$$\begin{aligned} \max(\|[Q]\sigma\|_{k,k}(\boldsymbol{x})) &= \max(\|Q; S\sigma\|_{k,k}^{x_\sigma}(\boldsymbol{x})) \\ &\leq \max(\|Q\|_{k,k}^{x_\sigma}(\boldsymbol{x})) \\ &\leq B_n^{x_\sigma \cdot \lambda(Q)}(\max(\boldsymbol{x})), \text{ by induction,} \\ &\leq B_{n+1}^{\lambda(Q)+2}(\max(\boldsymbol{x})), \text{ see (4.42)} \\ &= B_{n+1}^{\lambda(P)}(\max(\boldsymbol{x})). \end{aligned} \tag{4.41}$$

The last inequality follows from the assertion

$$B_n^{x \cdot a}(y) \leq B_{n+1}^{a+2}(y), \quad x \leq y, \tag{4.42}$$

which can be proved by using the properties of the small Ackermann functions in Proposition 4.1 as follows:

$$\begin{aligned} B_n^{x \cdot a}(y) &\leq B_n^{y \cdot a}(y) \\ &\leq B_n^{y \cdot a}(B_{n+1}(y)) \\ &= B_n^{y \cdot a}(B_n^y(1)) \\ &= B_n^{y(a+1)}(1) \\ &= B_{n+1}(y(a+1)) \\ &\leq B_{n+1}(y \cdot 2^{a+1}) \\ &= B_{n+1}(B_1^{a+1}(y)) \\ &\leq B_{n+1}(B_{n+1}^{a+1}(y)) \\ &= B_{n+1}^{a+2}(y)). \end{aligned}$$

□

Corollary 4.10. *The runtime function of a LOOP-n program is bounded by the n-th small Ackermann function.*

Proof. Let P be a LOOP-n program. By Proposition 4.6, the runtime program $\gamma(P)$ is also LOOP-n. Thus by Proposition 4.9, there is an integer m such that for all inputs $\boldsymbol{x} \in \mathbb{N}_0^k$ with $k \geq n(P)$,

$$\max(\|\gamma(P)\|_{k,k}(\boldsymbol{x})) \leq B_n^m(\max(\boldsymbol{x})). \tag{4.43}$$

But by definition, the runtime program P' of P satisfies $\|P'\|_{k,1}(\boldsymbol{x}) \leq \max(\|\gamma(P)\|_{k,k}(\boldsymbol{x}))$ for all $\boldsymbol{x} \in \mathbb{N}_0$ and so the result follows. □

For sake of completeness, the converse assertion also holds which is attributed to Meyer and Ritchie (1967).

Theorem 4.11. *(Meyer-Ritchie) Let $n \geq 2$. A LOOP computable function f is LOOP-n computable if and only if there is a LOOP-n program P such that $\|P\| = f$ and the associated runtime program P' is bounded by a LOOP-n computable function.*

Finally, the runtime functions provide a way to prove that the LOOP hierarchy is proper.

Proposition 4.12. *For each $n \in \mathbb{N}_0$, the class of LOOP-n functions is a proper subclass of the class of LOOP-$n+1$ functions.*

Proof. By Proposition 4.4, the n-th small Ackermann function B_n is LOOP-n computable. Assume that B_{n+1} is LOOP-n computable. Then by Proposition 4.9, there is an integer $m \geq 0$ such that $B_{n+1}(x) \leq B_n^m(x)$ for all $x \in \mathbb{N}_0$. This contradicts Proposition 4.3, which shows that B_{n+1} grows faster than any power of B_n. □

4.3 Ackermann's Function

As already remarked earlier, there are total computable functions that are not primitive recursive. The most prominent example is *Ackermann's function* $A : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by the equations

$$A(0, y) = y + 1, \tag{4.44}$$

$$A(x + 1, 0) = A(x, 1), \tag{4.45}$$

$$A(x + 1, y + 1) = A(x, A(x + 1, y)). \tag{4.46}$$

Proposition 4.13. *Ackermann's function is a total function on $\mathbb{N}_0^2$.*

Proof. The term $A(0, y)$ is certainly defined for all y. Suppose that $A(x, y)$ is defined for all y. Then $A(x+1, 0) = A(x, 1)$ is defined. Assume that $A(x+1, y)$ is defined. Then by induction, $A(x+1, y+1) = A(x, A(x+1, y))$ is defined. It follows that $A(x+1, y)$ is defined for all y. In turn, it follows that $A(x, y)$ is defined for all x and y. □

Proposition 4.14. *The monadic functions $A_x : \mathbb{N}_0 \to \mathbb{N}_0 : y \mapsto A(x, y)$, $x \in \mathbb{N}_0$, have the form*

- $A_0(y) = y + 1 = \nu(y + 3) - 3$,
- $A_1(y) = y + 2 = f_+(2, y + 3) - 3$,
- $A_2(y) = 2y + 3 = f_\cdot(2, y + 3) - 3$,
- $A_3(y) = 2^{y+3} - 3 = f_{\text{exp}}(2, y + 3) - 3$,
- $A_4(y) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} - 3 = f_{\text{itexp}}(2, y + 3) - 3$, *where there are $y + 3$ instances of the symbol 2 on the right-hand side.*

Proof.

- By definition, $A_0(y) = A(0, y) = y + 1$.
- First, $A_1(0) = A(0, 1) = 2$. Suppose $A_1(y) = y + 2$. Then $A_1(y + 1) = A(1, y + 1) = A(0, A(1, y)) = A(1, y) + 1 = (y + 1) + 2$.
- Plainly, $A_2(0) = A(1, 1) = 3$. Suppose $A_2(y) = 2y+3$. Then $A_2(y+1) = A(2, y+1) = A(1, A(2, y)) = A(2, y) + 2 = (2y + 3) + 2 = 2(y + 1) + 3$.
- Clearly, $A_3(0) = A(2, 1) = 5$. Suppose $A_3(y) = 2^{y+3} - 3$. Then $A_3(y + 1) = A(3, y + 1) = A(2, A(3, y)) = 2 \cdot (2^{y+3} - 3) + 3 = 2^{(y+1)+3} - 3$.

- First, $A_4(0) = A(3,1) = 2^{2^2} - 3$. Suppose that

  $$A_4(y) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} - 3,$$

  where there are $y+3$ instances of the symbol 2 on the right-hand side. Then

  $$A_4(y+1) = A(3, A(4,y)) = 2^{A(4,y)+3} - 3 = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} - 3,$$

  where there are $(y+1)+3$ instances of the symbol 2 on the far right of these equations.

□

The small Ackermann functions can be combined into a dyadic function $A : \mathbb{N}_0^2 \to \mathbb{N}_0$ given by

$$A(x,y) = B_x(y), \quad x, y \in \mathbb{N}_0. \tag{4.47}$$

This is a variant of the original Ackermann function.

Proposition 4.15. *The function A in (4.47) is given by the equations*

$$\begin{aligned} A(0,y) &= B_0(y), && (4.48)\\ A(x+1,0) &= 1, && (4.49)\\ A(x+1,y+1) &= A(x, A(x+1,y)). && (4.50) \end{aligned}$$

Proof. We have $A(0,y) = B_0(y)$, $A(x{+}1,0) = B_{x+1}(0) = B_x^0(1) = 1$, and $A(x{+}1,y{+}1) = B_{x+1}(y{+}1) = B_x^{y+1}(1) = B_x(B_x^y(1)) = B_x(B_{x+1}(y)) = B_x(A(x+1,y) = A(x, A(x+1,y))$. □

The Ackermann function grows very quickly. This can be seen by expanding a simple expression using the defining rules.

Example 4.16.

$$\begin{aligned} A(2,3) &= A(1, A(2,2))\\ &= A(1, A(1, A(2,1)))\\ &= A(1, A(1, A(1, A(2,0))))\\ &= A(1, A(1, A(1,1)))\\ &= A(1, A(1, A(0, A(1,0))))\\ &= A(1, A(1, A(0,1)))\\ &= A(1, A(1,2))\\ &= A(1, A(0, A(1,1)))\\ &= A(1, A(0, A(0, A(1,0))))\\ &= A(1, A(0, A(0,1)))\\ &= A(1, A(0,2))\\ &= A(1,4)\\ &= A(0, A(1,3)) \end{aligned}$$

$$
\begin{aligned}
&= A(0, A(0, A(1, 2)))\\
&= A(0, A(0, A(0, A(1, 1))))\\
&= A(0, A(0, A(0, A(0, A(1, 0)))))\\
&= A(0, A(0, A(0, A(0, 1))))\\
&= A(0, A(0, A(0, 2)))\\
&= A(0, A(0, 4))\\
&= A(0, 6)\\
&= 8.
\end{aligned}
$$

♢

Proposition 4.17. *Ackermann's function is not primitive recursive.*

Proof. Assume that the function A is primitive recursive. Then A is LOOP-n computable for some $n \geq 0$. Thus by Proposition 4.9, there is an integer $p \geq 0$ such that for all $x, y \in \mathbb{N}_0$,

$$A(x, y) \leq B_n^p(\max\{x, y\}). \tag{4.51}$$

But by Proposition 4.3, there is a number $y_0 \geq 0$ such that for all numbers $y \geq y_0$,

$$B_n^p(y) < B_{n+1}(y). \tag{4.52}$$

It can be assumed that $y_0 \geq n + 1$. Taking $x = n + 1$ and $y \geq y_0$ leads to a contradiction:

$$A(n + 1, y) \leq B_n^p(\max\{n + 1, y\}) = B_n^p(y) < B_{n+1}(y) = A(n + 1, y). \tag{4.53}$$

□

The lattice of function classes (under inclusion) considered in this chapter is the following:

$$\begin{tikzcd}
\mathcal{T} \arrow[d, dash] \\
\mathcal{P} = \mathcal{F}_{\text{LOOP}} \arrow[d, dash] \\
\cdots \arrow[d, dash] \\
\mathcal{F}_{\text{LOOP(1)}} \arrow[d, dash] \\
\mathcal{F}_{\text{LOOP(0)}}
\end{tikzcd}$$

All inclusions are strict.

5

Acceptable Programming Systems

Acceptable programming systems form the basis for the development of the theory of computation. This chapter provides several basic theoretical results of computability, such as the existence of universal functions, the parametrization theorem known as smn theorem, and Kleene's normal form theorem. These results make use of the Gödel numbering of partial recursive functions.

5.1 Gödel Numbering of GOTO Programs

In mathematical logic, Gödel numbering refers to a function that assigns to each well-formed formula of some formal language a unique natural number called its Gödel number. This concept was introduced by the logician Kurt Gödel (1906-1978) for the proof of incompleteness of elementary arithmetic (1931). Here Gödel numbering is used to provide an encoding of GOTO programs.

For this, let $\mathbb{N}_0^*$ denote the union of all cartesian products $\mathbb{N}_0^k$, $k \geq 0$. In particular, $\mathbb{N}_0^0 = \{\epsilon\}$, where ϵ is the empty string, and $\mathbb{N}_0^1 = \mathbb{N}_0$. The Cantor pairing function $J_2 : \mathbb{N}_0^2 \to \mathbb{N}_0$ can be used to define an encoding $J : \mathbb{N}_0^* \to \mathbb{N}_0$ as follows:

$$J(\epsilon) = 0, \tag{5.1}$$

$$J(x) = J_2(0, x) + 1, \quad x \in \mathbb{N}_0, \tag{5.2}$$

$$J(\boldsymbol{x}, y) = J_2(J(\boldsymbol{x}), y) + 1, \quad \boldsymbol{x} \in \mathbb{N}_0^*, y \in \mathbb{N}_0. \tag{5.3}$$

Note that the second equation is a special case of the third one, since for each $y \in \mathbb{N}_0$,

$$J(\epsilon, y) = J_2(J(\epsilon), y) + 1 = J_2(0, y) + 1 = J(y). \tag{5.4}$$

Example 5.1. We have

$$J(1,3) = J_2(J(1), 3) + 1 = J_2(J_2(0,1) + 1, 3) + 1 = J_2(2,3) + 1 = 17 + 1 = 18.$$

$\diamondsuit$

Proposition 5.2. *The encoding function J is a primitive recursive bijection.*

Proof. First, claim that J is primitive recursive. Indeed, the function J is primitive recursive for strings of length ≤ 1, since J_2 is primitive recursive. Assume that J is primitive recursive for strings of length $\leq k$, where $k \geq 1$. For strings of length $k+1$, the function J can be written as a composition of primitive recursive functions:

$$J = \nu \circ J_2(J(\pi_1^{(k+1)}, \ldots, \pi_k^{(k+1)}), \pi_{k+1}^{(k+1)}). \tag{5.5}$$

By induction hypothesis, J is primitive recursive for strings of length $\leq k$ and thus the right-hand side is primitive recursive. The claim follows.

Second, let A be the set of all numbers $n \in \mathbb{N}_0$ such that there is a unique string $\boldsymbol{x} \in \mathbb{N}_0^*$ with $J(\boldsymbol{x}) = n$. Claim that $A = \mathbb{N}_0$. Indeed, 0 lies in A since $J(\epsilon) = 0$ and $J(\boldsymbol{x}) > 0$ for all $\boldsymbol{x} \neq \epsilon$ Let $n > 0$ and assume that the assertion holds for all numbers $m < n$. Define

$$u = K_2(n-1) \quad \text{and} \quad v = L_2(n-1). \tag{5.6}$$

Then $J_2(u,v) = J_2(K_2(n-1), L_2(n-1)) = n-1$. By construction, $K_2(z) \leq z$ and $L_2(z) \leq z$ for all $z \in \mathbb{N}_0$. Thus $u = K_2(n-1) < n$ and hence $u \in A$. By induction, there is exactly one string $\boldsymbol{x} \in \mathbb{N}_0^k$ such that $J(\boldsymbol{x}) = u$. Then $J(\boldsymbol{x}, v) = J_2(J(\boldsymbol{x}), v)) + 1 = J_2(u,v) + 1 = n$.

Assume that $J(\boldsymbol{x}, v) = n = J(\boldsymbol{y}, w)$ for some $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{N}_0^*$ and $v, w \in \mathbb{N}_0$. Then by definition, $J_2(J(\boldsymbol{x}), v) = J_2(J(\boldsymbol{y}), w)$. But the Cantor pairing function is bijective and thus $J(\boldsymbol{x}) = J(\boldsymbol{y})$ and $v = w$. Since $J(\boldsymbol{x}) < n$ it follows by induction that $\boldsymbol{x} = \boldsymbol{y}$. Thus $n \in A$ and so by the induction axiom, $A = \mathbb{N}_0$. It follows that J is bijective. □

The encoding function J gives rise to two functions $K, L : \mathbb{N}_0 \to \mathbb{N}_0$ defined as

$$K(n) = K_2(n \dot{-} 1) \quad \text{and} \quad L(n) = L_2(n \dot{-} 1), \quad n \in \mathbb{N}_0. \tag{5.7}$$

Note that the following marginal conditions hold:

$$K(1) = K(0) = L(0) = L(1) = 0. \tag{5.8}$$

Proposition 5.3. *The functions K and L are primitive recursive, and for each number $n \geq 1$, there are unique $\boldsymbol{x} \in \mathbb{N}_0^*$ and $y \in \mathbb{N}_0$ with $J(\boldsymbol{x}, y) = n$ such that*

$$J(\boldsymbol{x}) = K(n) \quad \textit{and} \quad y = L(n). \tag{5.9}$$

Proof. By Proposition 2.34, the functions K_2 and L_2 are primitive recursive and so K and L are primitive recursive, too.

Let $n \geq 1$. By Proposition 5.2, there are unique $\boldsymbol{x} \in \mathbb{N}_0^*$ and $y \in \mathbb{N}_0$ such that $J(\boldsymbol{x}, y) = n$. Thus $J_2(J(\boldsymbol{x}), y) = n-1$. But $J_2(K_2(n-1), L_2(n-1)) = n-1$ and the Cantor pairing function J_2 is bijective. Thus $K_2(n-1) = J(\boldsymbol{x})$ and $L_2(n-1) = y$, and hence $K(n) = J(\boldsymbol{x})$ and $L(n) = y$. □

The length of a string can be determined by its encoding.

Proposition 5.4. *If $\boldsymbol{x}$ is a string in $\mathbb{N}_0^*$ of length k, then k is the smallest number such that*

$$K^k(J(\boldsymbol{x})) = 0.$$

Proof. In view of the empty string, $K^0(J(\epsilon)) = J(\epsilon) = 0$. Let $\boldsymbol{x} = x_1 \ldots x_k$ be a non-empty string. Then $J(\boldsymbol{x}) = n$ for some $n \geq 1$. By Proposition 5.3, $J(x_1 \ldots x_{k-1}) = K(n)$, and by induction, $K^{k-1}(J(x_1 \ldots x_{k-1})) = 0$. Therefore, $K^k(J(\boldsymbol{x})) = K^k(n) = K^{k-1}(K(n)) = K^{k-1}(J(x_1 \ldots x_{k-1})) = 0$, as requested.

By Proposition 5.3, we have $(K \circ J)(\boldsymbol{x}, y) = J(\boldsymbol{x})$ and hence each application of the function K reduces the length of the considered string by 1. In each step the attained value is nonzero as long as the string is non-empty and it becomes zero when the empty string is reached. This completes the proof. □

Example 5.5. Take a number $x \in \mathbb{N}_0$, i.e., a string of length 1. Then $J(x) = n \geq 1$. By (5.4), $J(x) = J(\epsilon, x)$ and therefore by (5.9), $K^1(n) = J(\epsilon) = 0$ and $L(n) = x$. ♢

In view of Proposition 5.4, define the mapping $f : \mathbb{N}_0^* \times \mathbb{N}_0 \to \mathbb{N}_0$ as

$$f : (\boldsymbol{x}, k) \mapsto K^k(J(\boldsymbol{x})). \tag{5.10}$$

Unbounded minimalization of this function yield the *length* function $\lg : \mathbb{N}_0^* \to \mathbb{N}_0$ given by

$$\lg(\boldsymbol{x}) = \mu f(\boldsymbol{x}) = \begin{cases} k \text{ if } k \text{ smallest with } f(\boldsymbol{x}, k) = 0 \text{ and } f(\boldsymbol{x}, i) > 0 \text{ for } 0 \leq i \leq k, \\ \uparrow \text{ otherwise.} \end{cases} \tag{5.11}$$

Proposition 5.6. *The length function* lg *is primitive recursive.*

Proof. The length function is partial recursive, since it is obtained by minimalization of a primitive recursive function. This minimization is bounded with the bound given by the argument $\boldsymbol{x}$ interpreted as natural number because a non-zero natural number is as least as large as its length. Since bounded minimalization of a primitive recursive function is primitive recursive, the result follows. □

Example 5.7. Let $n = 18$. Then $K(18) = K_2(17)$ and $L(18) = L_2(17)$. Since $\frac{1}{2}s(s+1) \leq 17 < \frac{1}{2}(s+1)(s+2)$ is satisfied by $s = 5$, we obtain $K_2(17) = 2$ and $L_2(17) = 3$ (Example 2.33). Moreover, $K^2(18) = K(K(18)) = K(2) = K_2(1)$. But $K_2(1) = 0$ and so $K^2(18) = 0$. It follows that the preimage of $n = 18$ is given by a string of length two. In order to determine the preimage of the number $n = 18$, we have $18 = 17 + 1 = J_2(2, 3) + 1 = J(J^{-1}(2), 3)$, and $2 = 1 + 1 = J_2(0, 1) + 1 = J(1)$. Therefore, $J(1, 3) = J_2(J(1), 3) + 1 = J_2(2, 3) + 1 = 18$. ♢

Proposition 5.8. *Let $n \geq 1$. The inverse value $J^{-1}(n)$ is given by the string*

$$(K^{k-1}(n), L \circ K^{k-2}(n), \ldots, L \circ K(n), L(n)), \tag{5.12}$$

where $k \geq 1$ is the smallest number such that $K^k(n) = 0$.

Proof. Let $x \in \mathbb{N}_0$. Then $J(x) = n \geq 1$ and as shown above, $K(n) = 0$ and $L(n) = x$. Hence, $J^{-1}(n) = L(n)$.

Let $\boldsymbol{x} \in \mathbb{N}_0^k$, $k \geq 1$, and $y \in \mathbb{N}_0$. Then $J(\boldsymbol{x}, y) = n \geq 1$. By Proposition 5.3, $K(n) = J(\boldsymbol{x})$ and $L(n) = y$. Since $K(n) < n$, induction shows that $\boldsymbol{x} = J^{-1}(K(n))$ is given by

$$(K^{k-1}(K(n)), L \circ K^{k-2}(K(n)), \ldots, L(K(n))) = (K^k(n), L \circ K^{k-1}(n), \ldots, L \circ K(n)).$$

Hence, $(\boldsymbol{x}, y)$ has the form

$$(K^k(n), L \circ K^{k-1}(n), \ldots, L \circ K(n), L(n))$$

as required. □

The primitive recursive bijection J allows to encode the *standard GOTO programs*, SGOTO programs for short. These are GOTO programs $P = s_0; s_1; \ldots; s_q$ that have a canonical labelling in the sense that $\lambda(s_l) = l$, $0 \leq l \leq q$. It is clear that for each GOTO program there is a semantically equivalent SGOTO program. SGOTO programs are used in the following, since they will permit a slightly simpler Gödel numbering than arbitrary GOTO programs. In the following, let $\mathcal{P}_{\text{SGOTO}}$ denote the class of all SGOTO programs.

Take an SGOTO program $P = s_0; s_1; \ldots; s_q$. For each $0 \leq l \leq q$, put

$$I(s_l) = \begin{cases} 3 \cdot J(i,k) & \text{if } s_l = (l, x_i \leftarrow x_i + 1, k), \\ 3 \cdot J(i,k) + 1 & \text{if } s_l = (l, x_i \leftarrow x_i - 1, k), \\ 3 \cdot J(i,k,m) + 2 & \text{if } s_l = (l, \texttt{if } x_i = 0, k, m). \end{cases} \tag{5.13}$$

The number $I(s_l)$ is called the *Gödel number* of the instruction s_l, $0 \leq l \leq q$. The function I is primitive recursive, since it is defined by cases and the functions involved are primitive recursive.

Note that the function I allows to identify the l-th instruction of an SGOTO program given its Gödel number e. Indeed, the residue of e modulo 3 provides the type of instruction and the quotient of e modulo 3 gives the encoding of the parameters of the instruction. More concretely, write

$$e = 3n + t, \tag{5.14}$$

where $n = \div(e, 3)$ and $t = \text{mod}(e, 3)$. The length of the string encoded by n is first determined by the minimalization given in Proposition 5.4. It is either two for the increment and decrement instructions, or three for the branching instructions. More concretely, by Proposition 5.8, $J^{-1}(n) = (K(n), L(n))$ for the increment and decrement instructions, and $J^{-1}(n) = (K^2(n), L(K(n)), L(n))$ for the branching instructions. Then the instruction can be decoded as follows:

$$s_l = \begin{cases} (l, x_{K(n)} \leftarrow x_{K(n)} + 1, L(n)) & \text{if } t = 0, \\ (l, x_{K(n)} \leftarrow x_{K(n)} - 1, L(n)) & \text{if } t = 1, \\ (l, \texttt{if } x_{K^2(n)} = 0, L(K(n)), L(n)) & \text{if } t = 2. \end{cases} \tag{5.15}$$

Example 5.9. Suppose the l-th instruction of an SGOTO program is encoded by the number $e = 55$. We have $55 = 3 \cdot 18 + 1$ and so $n = 18$. By Example 5.1, $J(1, 3) = 18$ and therefore the instruction is $s_l = (l, x_1 \leftarrow x_1 - 1, 3)$. ♢

Finally, the *Gödel number* of an SGOTO program $P = s_0; s_1; \ldots; s_q$ is defined as

$$\Gamma(P) = J(I(s_0), I(s_1), \ldots, I(s_q)). \tag{5.16}$$

Proposition 5.10. *The function $\Gamma : \mathcal{P}_{\text{SGOTO}} \to \mathbb{N}_0$ is bijective and primitive recursive.*

Proof. The mapping Γ is bijective since J is bijective and the instructions encoded by I are uniquely determined as shown above. Moreover, the function Γ is a composition of primitive recursive functions and thus is primitive recursive. □

The SGOTO program P with Gödel number e is denoted by P_e. Gödel numbering provides a list of all SGOTO programs

$$P_0, P_1, P_2, \ldots. \tag{5.17}$$

Conversely, each number e can be assigned the SGOTO program P such that $\Gamma(P) = e$. For this, the length of the string encoded by e is first determined by the minimalization given in Proposition 5.4. Suppose the string has length $n + 1$, where $n \geq 0$. Then the task is to find $\boldsymbol{x} \in \mathbb{N}_0^n$ and $y \in \mathbb{N}_0$ such that $J(\boldsymbol{x}, y) = n$. But by (5.9), $K(n) = J(\boldsymbol{x})$ and $L(n) = y$ and so the preimage of n under J can be repeatedly determined. Finally, when the string is given, the preimage (instruction) of each constituent (number) can be established as described in (5.15).

For each number e and each number $n \geq 0$, denote the n-ary partial recursive function computing the SGOTO program with Gödel number e by

$$\phi_e^{(n)} = \|P_e\|_{n,1}. \tag{5.18}$$

If f is an n-ary partial recursive function, each number $e \in \mathbb{N}_0$ with the property $f = \phi_e^{(n)}$ is called an *index* of f. The index of a partial recursive function f provides the Gödel number of an SGOTO program computing it. The list of all SGOTO program in (5.17) yields a list of all n-ary partial recursive functions:

$$\phi_0^{(n)}, \phi_1^{(n)}, \phi_2^{(n)}, \ldots. \tag{5.19}$$

Note that the list contains repetitions, since each n-ary partial recursive function has infinitely many indices.

5.2 Parametrization

The parametrization theorem, also called smn theorem, is a cornerstone of computability theory. It was first proved by Kleene (1943) and refers to computable functions in which some arguments are considered as parameters. The smn theorem does not only tell that the resulting function is computable but also shows how to compute an index for it. A special case is considered first which applies to dyadic computable functions.

Proposition 5.11. *For each dyadic partial recursive function f, there is a monadic primitive recursive function g such that*

$$f(x, \cdot) = \phi_{g(x)}^{(1)}, \quad x \in \mathbb{N}_0. \tag{5.20}$$

Proof. Let $P_e = s_0; s_1; \ldots; s_q$ be an SGOTO program computing the function f. For each number $x \in \mathbb{N}_0$, consider the following SGOTO program Q_x:

$$\begin{array}{llll}
0 & \texttt{if}\ x_1 = 0 & 3 & 1 \\
1 & x_2 \leftarrow x_2 + 1 & 2 & \\
2 & x_1 \leftarrow x_1 - 1 & 0 & \\
3 & x_1 \leftarrow x_1 + 1 & 4 & \\
4 & x_1 \leftarrow x_1 + 1 & 5 & \\
 & \vdots & & \\
2 + x & x_1 \leftarrow x_1 + 1 & 3 + x & \\
 & s_0' & & \\
 & s_1' & & \\
 & \vdots & & \\
 & s_q' & &
\end{array}$$

where $P'_e = s'_0; s'_1; \ldots; s'_q$ is the SGOTO program that is derived from P by replacing each occurring label j with $j + 3 + x$, $0 \leq j \leq q$. The milestones of the computation of Q_x are given by the following configurations:

$$\begin{array}{llll}
0 & y & 0\;0\;0\ldots & \text{init} \\
0 & 0 & y\;0\;0\ldots & \text{step } 3 \\
0 & x & y\;0\;0\ldots & \text{step } 3+x \\
0 & f(x,y) & \ldots\ldots & \text{end}
\end{array}$$

It follows that $\|Q_x\|_{1,1}(y) = f(x, y)$ for all $x, y \in \mathbb{N}_0$.

Take the function $g : \mathbb{N}_0 \to \mathbb{N}_0$ defined by $g(x) = \Gamma(Q_x)$, i.e., $g(x)$ is the Gödel number of the program Q_x. This function is primitive recursive by Proposition 5.10. But by definition, $\phi^{(1)}_{g(x)} = \|Q_x\|_{1,1}$ and thus the result follows. □

This assertion is a special case of the socalled smn theorem. The unimaginative name originates from Kleene's notation $s_{m,n}$ for the primitive recursive function playing a key role later on.

Theorem 5.12. (**smn Theorem**) *For each pair of numbers $m, n \geq 1$, there is an $m+1$-ary primitive recursive function $s_{m,n}$ such that*

$$\phi^{(m+n)}_e(\boldsymbol{x}, \cdot) = \phi^{(n)}_{s_{m,n}(e,\boldsymbol{x})}, \quad \boldsymbol{x} \in \mathbb{N}^m_0,\ e \in \mathbb{N}_0. \tag{5.21}$$

Proof. The idea is quite similar to that of the special case. Take an SGOTO program $P_e = s_0; s_1; \ldots; s_q$ calculating the function $\phi^{(m+n)}_e$. For each input $\boldsymbol{x} \in \mathbb{N}^m_0$, extend the program P_e to an SGOTO program $Q_{e,\boldsymbol{x}}$ providing the following intermediate configurations:

$$\begin{array}{llll}
0 & \boldsymbol{y} & 0\;0\;0\ldots & \text{init} \\
0 & \boldsymbol{0} & \boldsymbol{y}\;0\;0\ldots & \text{reload } \boldsymbol{y} \\
0 & \boldsymbol{x} & \boldsymbol{y}\;0\;0\ldots & \text{generate parameter } \boldsymbol{x} \\
0 & \phi^{(m+n)}_e(\boldsymbol{x},\boldsymbol{y}) & \ldots\ldots & \text{end}
\end{array}$$

It follows that $\|Q_{e,\boldsymbol{x}}\|_{n,1}(\boldsymbol{y}) = \phi^{(m+n)}_e(\boldsymbol{x}, \boldsymbol{y})$ for all $\boldsymbol{x} \in \mathbb{N}^m_0$ and $\boldsymbol{y} \in \mathbb{N}^n_0$.

Consider the function $s_{m,n} : \mathbb{N}^{m+1}_0 \to \mathbb{N}_0$ defined by $s_{m,n}(e, \boldsymbol{x}) = \Gamma(Q_{e,\boldsymbol{x}})$; that is, $s_{m,n}(e, \boldsymbol{x})$ is the Gödel number of the program $Q_{e,\boldsymbol{x}}$. This function is primitive recursive by Proposition 5.10. But by definition, $\phi^{(n)}_{s_{m,n}(e,\boldsymbol{x})} = \|Q_{e,\boldsymbol{x}}\|_{n,1}$ and thus the result follows. □

5.3 Universal Functions

Another basic result of computability theory is the existence of a computable function called universal function that is capable of computing any other computable function. Let $n \geq 1$ be a number. A *universal function* for n-ary partial recursive functions is an $n+1$-ary function $\psi^{(n)}_{\text{univ}} : \mathbb{N}^{n+1}_0 \to \mathbb{N}_0$ such that

$$\psi^{(n)}_{\text{univ}}(e, \cdot) = \phi^{(n)}_e, \quad e \in \mathbb{N}_0. \tag{5.22}$$

Theorem 5.13. *For each arity $n \geq 1$, the universal function $\psi^{(n)}_{\text{univ}}$ exists and is partial recursive.*

Proof. The existence will be proved in several steps. First, define the *one-step function* $E : \mathbb{N}_0^2 \to \mathbb{N}_0^2$ that describes the process to move from one configuration of the URM to the next one during the computation of an SGOTO program. For this, consider the following diagram:

$$\begin{tikzcd}
e,\xi \arrow[r, "E"'] \arrow[d, "G^{-1}"'] & e,\xi' \\
P_e,\omega \arrow[r, "s_l"'] & P_e,\omega' \arrow[u, "G"']
\end{tikzcd}$$

The one-step function E takes a pair (e, ξ) and recovers from the second component ξ the corresponding configuration of the URM given by $\omega = G^{-1}(\xi)$. Then the next instruction s_l, the l-th one, given by the SGOTO program $P = P_e$ with Gödel number e is executed providing a new configuration ω' of the URM which is then encoded as $\xi' = G(\omega')$. In this way, the function E is defined as

$$E(e, \xi) = (e, \xi'). \tag{5.23}$$

More specifically, the given number ξ is a product of prime powers

$$\xi = p_0^l p_1^{\omega_1} p_2^{\omega_2} p_3^{\omega_3} \cdots = 2^l 3^{\omega_1} 5^{\omega_2} 7^{\omega_3} \cdots \tag{5.24}$$

with $\omega_0 = l$ and the associated configuration of the URM is given by the state

$$\omega = (\omega_0, \omega_1, \omega_2, \omega_3, \ldots). \tag{5.25}$$

In particular, $l = \omega_0 = G_0(\xi)$ is the label of the instruction to be executed. In the next step, the preimage of the program's Gödel number e using the length function and Proposition 5.8 is determined:

$$J^{-1}(e) = (\sigma_0, \sigma_1, \ldots, \sigma_q). \tag{5.26}$$

Afterwards the number σ_l is decoded into the associated instruction s_l using (5.14) and (5.15). Then the instruction is executed and the encoding of the next configuration is defined as follows:

- If $s_l = (l, x_i \leftarrow x_i + 1, k)$, the next state of the URM is

$$\omega' = (k, \omega_1, \ldots, \omega_{i-1}, \omega_i + 1, , \omega_{i+1}, \ldots) \tag{5.27}$$

  and the next label is k.
- If $s_l = (l, x_i \leftarrow x_i - 1, k)$, the next state of the URM is

$$\omega' = (k, \omega_1, \ldots, \omega_{i-1}, \omega_i \dot{-} 1, , \omega_{i+1}, \ldots) \tag{5.28}$$

  and the next label is k.
- If $s_l = (l, \texttt{if } x_i = 0, k, m)$, the state of the URM remains unchanged (up to the label) and the next label is either k or l depending on whether the value x_i is zero or not.

In each case, the next state is given by $\xi' = G(\omega')$. Since the function E is defined by cases and all functions occurring are primitive recursive, it follows that E is also primitive recursive.

Second, the execution of the SGOTO program P_e can be described by the *iterated one-step function* $E' : \mathbb{N}_0^3 \to \mathbb{N}_0$ defined as

$$E'(e,\xi,t) = (\pi_2^{(2)} \circ E^t)(e,\xi). \tag{5.29}$$

This function is primitive recursive, since it is given by applying a projection to an iteration of a primitive recursive function.

Third, the computation of the SGOTO program P_e terminates if it reaches a non-existing label. Equivalently, termination is reached if and only if

$$G_0(\xi) \notin L(P_e). \tag{5.30}$$

Note that the set $L(P_e)$ is primitive, since it is finite. Thus the corresponding characteristic function $\chi_{L(P_e)}$ is primitive recursive and hence the above condition is equivalent to

$$\chi_{L(P_e)}(G_0(\xi)) = 0. \tag{5.31}$$

The *runtime function* $Z : \mathbb{N}_0^2 \to \mathbb{N}_0$ assigns to the program P_e and the state $\omega = G^{-1}(\xi)$ the number of steps required to reach termination,

$$Z(e,\xi) = \mu_t(\chi_{L(P_e)}(G_0(E'(e,\xi,t))) = 0), \tag{5.32}$$

where the minimalization $\mu = \mu_t$ is subject to the variable t counting the number of steps. This function is only partial recursive since the minimalization is unbounded, since the program P_e may not terminate and even if it terminates, the number of steps until termination will be unknown.

Fourth, the *residual step function* is defined by the partial function $R : \mathbb{N}_0^2 \to \mathbb{N}_0$ which assigns to each SGOTO program P_e and each initial state $\omega = G^{-1}(\xi)$ the result of computation given by the content of the first register:

$$R(e,\xi) = G_1(E'(e,\xi,Z(e,\xi))). \tag{5.33}$$

This function is partial recursive, since the function Z is partial recursive.

Summing up, the desired partial recursive function is given as

$$\psi_{\text{univ}}^{(n)}(e,\boldsymbol{x}) = R(e, G(0,x_1,\ldots,x_n,0,0,\ldots)), \tag{5.34}$$

which equals $\phi_e^{(n)}(\boldsymbol{x}) = \|P_e\|_{n,1}(\boldsymbol{x})$, where e is a Gödel number and $\boldsymbol{x} = (x_1,\ldots,x_n) \in \mathbb{N}_0^n$ an input. □

The existence of universal functions implies the existence of universal URM programs, and vice versa.

An *acceptable programming system* is considered to be an enumeration of n-ary partial recursive functions $\psi_0, \psi_1, \psi_2, \ldots$ for which both the smn theorem and the theorem about the existence of universal functions hold. For instance, the enumeration of URM (or GOTO) computable functions forms an acceptable programming system.

5.4 Kleene's Normal Form

Kleene (1943) introduced the T predicate that tells whether an SGOTO program will halt when run with a particular input and if so, a corresponding function provides the result of computation. Similar to the smn theorem, the original notation used by Kleene has become standard terminology.

First note that the definition of the universal function allows to define the *extended Kleene set* $S_n \subseteq \mathbb{N}_0^{n+3}$, $n \geq 1$, given as

$$(e, \boldsymbol{x}, z, t) \in S_n \quad :\Longleftrightarrow \quad \chi_{L(P_e)}(G_0(E'(e, \xi_{\boldsymbol{x}}, t))) = 0 \quad \wedge \quad G_1(E'(e, \xi_{\boldsymbol{x}}, t)) = z, \tag{5.35}$$

where $\xi_{\boldsymbol{x}} = G(0, x_1, \ldots, x_n, 0, 0, \ldots)$ is the encoding of the initial state comprising the input $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{N}_0^n$. Note that $(e, \boldsymbol{x}, z, t) \in S_n$ if and only if the program ϕ_e with input $\boldsymbol{x}$ terminates after t steps and the result of computation is z.

Proposition 5.14. *For each arity $n \geq 1$, the set S_n is primitive,*

Proof. The functions used to define the set S_n are primitive recursive. In particular, the characteristic function of $L(P_e)$ is primitive recursive since the label set $L(P_e)$ is finite. □

Let $A \subseteq \mathbb{N}_0^{n+1}$ be a relation. First, the *unbounded minimalization* of A is the function $\mu A : \mathbb{N}_0^n \to \mathbb{N}_0$ defined as

$$\mu A(\boldsymbol{x}) = \mu(\text{csg} \circ \chi_A)(\boldsymbol{x}) = \begin{cases} y \text{ if } (\boldsymbol{x}, y) \in A \text{ and } (\boldsymbol{x}, i) \not\in A \text{ for all } 0 \leq i < y, \\ \uparrow \text{otherwise.} \end{cases} \tag{5.36}$$

We write

$$\mu A(\boldsymbol{x}) = \mu y[(\boldsymbol{x}, y) \in A], \quad \boldsymbol{x} \in \mathbb{N}_0^n. \tag{5.37}$$

Second, the *unbounded existential quantification* of A is the relation $E_A \subseteq \mathbb{N}_0^n$ given by

$$E_A = \{\boldsymbol{x} \mid \exists y \in \mathbb{N}_0 : (\boldsymbol{x}, y) \in A\}. \tag{5.38}$$

Put

$$\exists y[(\boldsymbol{x}, y) \in A] \quad :\Longleftrightarrow \quad \boldsymbol{x} \in E_A, \quad \boldsymbol{x} \in \mathbb{N}_0^n. \tag{5.39}$$

Third, the *unbounded universal quantification* of A is the relation $U_A \subseteq \mathbb{N}_0^n$ defined by

$$U_A = \{\boldsymbol{x} \mid \forall y \in \mathbb{N}_0 : (\boldsymbol{x}, y) \in A\}. \tag{5.40}$$

Set

$$\forall y[(\boldsymbol{x}, y) \in A] \quad :\Longleftrightarrow \quad \boldsymbol{x} \in E_A, \quad \boldsymbol{x} \in \mathbb{N}_0^n. \tag{5.41}$$

Theorem 5.15 (Kleene). *For each arity $n \geq 1$, there is a primitive set $T_n \subseteq \mathbb{N}_0^{n+2}$ called* Kleene set *such that for all $\boldsymbol{x} \in \mathbb{N}_0^n$:*

$$\boldsymbol{x} \in \text{dom}\, \phi_e^{(n)} \Longleftrightarrow (e, \boldsymbol{x}) \in \exists y[(e, \boldsymbol{x}, y) \in T_n]. \tag{5.42}$$

Moreover, for each $\boldsymbol{x} \in \mathbb{N}_0^n$,

$$\begin{aligned} \phi_e^{(n)}(\boldsymbol{x}) &= U(\mu y[(e, \boldsymbol{x}, y) \in T_n]) \\ &= \begin{cases} U(y) & \textit{if } (e, \boldsymbol{x}, y) \in T_n \textit{ and } (e, \boldsymbol{x}, i) \not\in T_n \textit{ for } 0 \leq i < y, \\ \uparrow & \textit{otherwise,} \end{cases} \end{aligned} \tag{5.43}$$

where $U : \mathbb{N}_0 \to \mathbb{N}_0$ is a primitive recursive function.

Proof. Define the relation $T_n \subseteq \mathbb{N}_0^{n+2}$ as follows:

$$(e, \boldsymbol{x}, y) \in T_n \quad :\Longleftrightarrow \quad (e, \boldsymbol{x}, K_2(y), L_2(y)) \in S_n. \tag{5.44}$$

That is, the component y encodes both, the result of computation $z = K_2(y)$ and the number of steps $t = L_2(y)$. Since the set S_n is primitive and the functions K_2 and L_2 are primitive recursive, it follows that the set T_n is primitive.

Let $\boldsymbol{x} \in \mathbb{N}_0^n$ lie in the domain of $\phi_e^{(n)}$. Then $\phi_e^{(n)}(\boldsymbol{x}) = z$ for some $z \in \mathbb{N}_0$. Thus there is a number $t \geq 0$ such that $(e, \boldsymbol{x}, z, t) \in S_n$. Putting $y = J_2(z, t)$ yields $(e, \boldsymbol{x}, y) \in T_n$; that is, $(e, \boldsymbol{x}) \in \exists y[(e, \boldsymbol{x}, y) \in T_n]$.

Conversely, let $(e, \boldsymbol{x}) \in \exists y[(e, \boldsymbol{x}, y) \in T_n]$. Then there is a number $y \geq 0$ such that $(e, \boldsymbol{x}, y) \in T_n$. Thus by definition, $(e, \boldsymbol{x}, K_2(y), L_2(y)) \in S_n$ and hence $\boldsymbol{x}$ belongs to the domain of $\phi_e^{(n)}$.

Finally, let $\boldsymbol{x} \in \operatorname{dom} \phi_e^{(n)}$. Then there is a number $y \geq 0$ such that $(e, \boldsymbol{x}, y) \in T_n$. Define the function $U : \mathbb{N}_0 \to \mathbb{N}_0$ by putting $U(y) = K_2(y)$. The function U is primitive recursive and by definition of S_n yields the result of computation. □

Kleene's normal form implies that any partial recursive function can be defined by using a single instance of the μ (minimalization) operator applied to a primitive recursive function. In the context of programming, this means that any program can be written using a single `while` loop.

6

Turing Machine

The Turing machine has been described by the British mathematician Alan Turing (1912-1954) as a thought experiment representing a computing machine. Despite its simplicity, a Turing machine is a device of universal computation.

6.1 The Machinery

A Turing machine consists of an infinite tape and an associated read/write head connected to a control mechanism. The tape is divided into denumerably many cells, each of which containing a symbol from a tape alphabet. This alphabet contains the special symbol "b" signifying that a cell is blank or empty. The cells are scanned, one at a time, by the read/write head which is able to move in both directions. At any given time instant, the machine will be in one of a finite number of states. The behaviour of the read/write head and the change of the machine's state are governed by the present state of the machine and by the symbol in the cell under scan.

The machine operates on words over an input alphabet. The symbols forming a word are written, in order, in consecutive cells of the tape from left to right. When the machine enters a state, the read/write head scans the symbol in the controlled cell, and writes a symbol from the tape alphabet to this cell. Then the head moves one cell to the left, or one cell to the right, or not at all. After this, the machine enters a new state.

A *Turing machine* is a quintuple $M = (\Sigma, Q, \delta, q_0, q_F)$ consisting of

- a finite alphabet Σ, called *tape alphabet*, containing a distinguished *blank* symbol b; the subset $\Sigma_I = \Sigma \setminus \{b\}$ is called *input alphabet*.
- a finite set Q of *states*,
- a partial function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R, \Lambda\}$, the *state transition function*,
- a *start state* $q_0 \in Q$, and
- a *halt state* $q_F \in Q$ such that $\delta(q_F, \sigma)$ is undefined for all σ in Σ.

The symbols L, R, and Λ are interpreted as *left move*, *right move*, and *no move*, respectively. The tape cells can be numbered by the set of integers $\mathbb{Z}$, and the tape content can be considered as a mapping $\tau : \mathbb{Z} \to \Sigma$ which assigns blank to almost every cell; that is, only a finite portion of the tape contains symbols from the input alphabet.

A *configuration* of a Turing machine M consists of the content of the tape which may be given by the finite portion of the tape containing symbols from the input alphabet, the cell controlled by the read/write head, and the state of the machine. Thus a configuration can be pictured as follows:

$$\begin{array}{c} — \; a_{i_1} \; a_{i_2} \; \ldots \; \overset{}{a_{i_j}} \; \ldots \; a_{i_l} \; — \\ \phantom{— \; a_{i_1} \; a_{i_2} \; \ldots \;} \uparrow \phantom{\; \ldots \; a_{i_l} \; —} \\ \phantom{— \; a_{i_1} \; a_{i_2} \; \ldots \;} q \phantom{\; \ldots \; a_{i_l} \; —} \end{array}$$

where $q \in Q$ is the current state, the cell controlled by the read/write head is marked by the arrow, and all cells to the left of a_{i_1} and to the right of a_{i_l} contain the blank symbol. This configuration will also be written as sequence $(a_{i_1} \ldots a_{i_{j-1}}, q, a_{i_j} \ldots a_{i_l})$.

The equation $\delta(q, a) = (q', a', D)$ means that if the machine is in state $q \in Q$ and reads the symbol $a \in \Sigma$ from the cell controlled by the read/write head, it writes the symbol $a' \in \Sigma$ into this cell, moves to the left if $D = L$, or moves to the right if $D = R$, or moves not at all if $D = \Lambda$, and enters the state $q' \in Q$. Given a configuration (uc, q, av) where $a, c \in \Sigma$, $u, v \in \Sigma^*$, and $q \in Q$, $q \neq q_F$. The configuration *reached in one step* from it is given as follows:

$$(u', q', v') = \begin{cases} (uca', q', v) \text{ if } \delta(q, a) = (q', a', R), \\ (u, q', ca'v) \text{ if } \delta(q, a) = (q', a', L), \\ (uc, q', a'v) \text{ if } \delta(q, a) = (q', a', \Lambda). \end{cases} \tag{6.1}$$

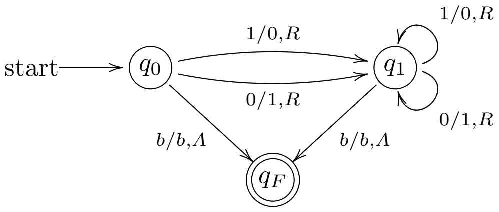
In the following, the notation $(u, q, v) \vdash (u', q', v')$ will signify that (u', q', v') is reached from (u, q, v) in one step.

A *computation* of the Turing machine M is started by writing, in order, the symbols of the input word $\boldsymbol{x} = x_1 x_2 \ldots x_n \in \Sigma_I^*$ in consecutive cells of the tape from left to right, while all other cells contain the blank symbol. Moreover, the machine is initially in the state q_0 and the read/right head controls the leftmost cell of the input; that is, the *initial configuration* can be illustrated as follows:

$$\begin{array}{l} — \; x_1 \; x_2 \; \ldots \; x_n \; — \\ \uparrow \\ q_0 \end{array}$$

Then the machine performs eventually a sequence of transitions as given by the state transition function. If the machine reaches the state q_F, it stops and the output of the computation is given by the collection of symbols on the tape.

Example 6.1. Consider the Turing machine $M = (\Sigma, Q, \delta, q_0, q_F)$ with tape alphabet $\Sigma = \{0, 1, b\}$, state set $Q = \{q_0, q_1, q_F\}$, and state transition function δ given by the following state diagram:



In such a diagram, the encircled nodes represent the states and an arrow joining state q with state q' and bearing the label $a/a', D$ indicates the transition $\delta(q, a) = (q', a', D)$. For instance, the computation of the machine on the binary input 0011 is the following:

$$(b, q_0, 0011) \vdash (1, q_1, 011) \vdash (11, q_1, 11) \vdash (110, q_1, 0) \vdash (1100, q_1, b) \vdash (1100, q_F, b).$$

In general, the machine calculates the bitwise complement of the given binary word. ♢

6.2 Post-Turing Machine

A Turing machine specified by state diagrams is rather hard to follow. Therefore, a program formulation of the Turing machine known as Post-Turing machine invented by Martin Davis (born 1928) will subsequently be considered.

A *Post-Turing machine* uses a binary alphabet $\Sigma = \{1, b\}$, an infinite tape of binary storage locations, and a primitive programming language with instructions for bidirectional movement among the storage locations, alteration of cell content one at a time, and conditional jumps. The instructions are as follows:

- `write` a, where $a \in \Sigma$,
- `move left`,
- `move right`, and
- `if read` a `then goto` A, where $a \in \Sigma$ and A is a label.

A *Post-Turing program* consists of a finite sequence of labelled instructions which are sequentially executed starting with the first instruction. Each instruction may have a label. If so, this label must be unique in the program. Since there is no specific instruction for termination, the machine will stop when it has arrived at a state at which the program contains no instruction telling the machine what to do next.

First, several Post-Turing programs will be introduced to be used as macros later on. The first macro is `move left to next blank:`

$$\begin{array}{rl} A: & \texttt{move left} \\ & \texttt{if read } 1 \texttt{ then goto } A \end{array} \tag{6.2}$$

This program moves the read/write head to next blank on the left. If this program is started in the configuration

$$\begin{array}{cccccccc} — & b & 1 & 1 & 1 & \ldots & 1 & — \\ & & & & & & \uparrow & \end{array}$$

it will end in the configuration

$$\begin{array}{cccccccc} — & b & 1 & 1 & 1 & \ldots & 1 & — \\ & \uparrow & & & & & & \end{array}$$

The macro `move right to next blank` is similarly defined:

$$\begin{array}{rl} A: & \texttt{move right} \\ & \texttt{if read } 1 \texttt{ then goto } A \end{array} \tag{6.3}$$

This program moves the read/write head to next blank on the right. If this program begins in the configuration

$$\begin{array}{cccccccc} — & 1 & 1 & 1 & \ldots & 1 & b & — \\ & \uparrow & & & & & & \end{array}$$

it will stop in the configuration

$$\begin{array}{cccccccc} — & 1 & 1 & 1 & \ldots & 1 & b & — \\ &&&&&& \uparrow & \end{array}$$

The macro `write` $b1$ is defined as

```
write $b$
move right
write 1
move right
```

$$(6.4)$$

If this macro begins in the configuration

$$\begin{array}{cccccc} — & a_{i_0} & a_{i_1} & a_{i_2} & a_{i_3} & — \\ && \uparrow &&& \end{array}$$

it will terminate in the configuration

$$\begin{array}{cccccc} — & a_{i_0} & b & 1 & a_{i_3} & — \\ &&&& \uparrow & \end{array}$$

The macro `move block right` is given as

```
write $b$
move right to next blank
write 1
move right
```

$$(6.5)$$

This program shifts a block of 1's by one cell to the right such that it merges with the subsequent block of 1's to right. If this macro starts in the configuration

$$\begin{array}{cccccccccccccc} — & b & 1 & 1 & 1 & \ldots & 1 & b & 1 & 1 & 1 & \ldots & 1 & b & — \\ && \uparrow &&&&&&&&&&&& \end{array}$$

it will halt in the configuration

$$\begin{array}{cccccccccccccc} — & b & b & 1 & 1 & \ldots & 1 & 1 & 1 & 1 & 1 & \ldots & 1 & b & — \\ &&&&&&&& \uparrow &&&&&& \end{array}$$

The macro `move block left` is analogously defined. The unconditional jump `goto` A stands for the Post-Turing program

```
if read $b$ then goto $A$
if read 1 then goto $A$
```

$$(6.6)$$

Another useful macro is `erase` given as

```
$A$ : if read $b$ then goto $B$
      write $b$
      move left
      goto $A$
$B$ : move left
```

$$(6.7)$$

This program deletes a block of 1's from right to left. If it starts in the configuration

$$\begin{array}{cccccccccc} — & a & b & 1 & 1 & 1 & \ldots & 1 & b & — \\ & & & & & & & \uparrow & & \end{array}$$

where a is an arbitrary symbol, it will stop in the configuration

$$\begin{array}{cccccccccc} — & a & b & b & b & b & \ldots & b & b & — \\ & \uparrow & & & & & & & & \end{array}$$

Finally, the repetition of a statement such as

$$\begin{array}{l} \texttt{move left} \\ \texttt{move left} \\ \texttt{move left} \end{array} \tag{6.8}$$

will be abbreviated by denoting the statement and the number of repetitions in parenthesis such as

$$\texttt{move left}\ (3) \tag{6.9}$$

For instance, the routine `move block right` (2) shifts two consecutive blocks by one cell to the right. It turns the configuration

$$\begin{array}{cccccccccccccccccc} — & b & 1 & 1 & \ldots & 1 & b & 1 & 1 & \ldots & 1 & b & 1 & 1 & \ldots & 1 & b & — \\ & & \uparrow & & & & & & & & & & & & & & & \end{array}$$

into the configuration

$$\begin{array}{cccccccccccccccccc} — & b & b & 1 & \ldots & 1 & 1 & b & 1 & \ldots & 1 & 1 & 1 & 1 & \ldots & 1 & b & — \\ & & & & & & & & & & & & \uparrow & & & & & \end{array}$$

6.3 Turing Computable Functions

The Turing machine has the same computational capabilities as the unrestricted register machine. To prove this, it will be shown that the URM computable functions are computable by Post-Turing programs. For this, let $f : \mathbb{N}_0^k \to \mathbb{N}_0$ be an URM computable function. It may be assumed that there is a GOTO program P that computes the function f and uses the variables $x_1, \ldots, x_n$, where $n \geq k$. The goal is to provide a Post-Turing program P' that simulates the computation of P. The program P' consists of three subroutines:

$$\begin{array}{rl} \textit{start:} & \texttt{initiate} \\ & \texttt{simulate} \\ & \texttt{clean\_up} \end{array} \tag{6.10}$$

The routine `initiate` presets the program for the computation of the function f. An input $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{N}_0^k$ of the function f is encoded on the tape in unary format; that is, a natural number x is represented by a block of $x + 1$ ones and a sequence of k numbers is described by k blocks such that consecutive blocks are separated by one blank. Thus the initial tape looks as follows:

$$— \overbrace{\underset{\uparrow}{1}1\ldots1}^{x_1+1}\, b\, \overbrace{11\ldots1}^{x_2+1}\, b \ldots b\, \overbrace{11\ldots1}^{x_k+1} — \tag{6.11}$$

The idea is that the blocks will correspond to the registers $R_1, R_2, \ldots, R_k$ of the URM. The remaining registers used during the computation will be initialized by zeros. This will be accomplished by the routine `initiate`:

```
init: move right to next blank $(k)$
      write $b1$ $(n-k)$
      move left to next blank $(n)$
      move right
```
(6.12)

If the procedure `initiate` is started in the configuration (6.11), it will end in the configuration

$$— b\, \overbrace{\underset{\uparrow}{1}1\ldots1}^{x_1+1}\, b\, \overbrace{11\ldots1}^{x_2+1}\, b \ldots b\, \overbrace{11\ldots1}^{x_k+1}\, b\,1\,b\,1\,b \ldots b\,1\,b\,1\,b — \tag{6.13}$$

The tape can now be interpreted in a way that it contains the numbers $x_1, \ldots, x_k$ followed by $n-k$ zeros.

The routine `simulate` is the heart of the simulation program. The idea is to simulate each GOTO instruction by a corresponding Post-Turing program such that the Post-Turing programs concatenated in order, constitute the procedure `simulate`. For this, the simulation should keep track of the registers $R_1, R_2, \ldots, R_n$ used to carry out the program P. To this end, the tape will always contain n blocks of ones which will in turn correspond to these registers.

The GOTO instruction $(l, x_i \leftarrow x_i + 1, l')$ is simulated by the Post-Turing program

```
$Al$ : move right to next blank $(i)$
$Bl$ : write 1
       move right
       if read $b$ then goto $Cl$
       write $b$
       move right to next blank
       goto $Bl$
$Cl$ : move left to next blank $(n)$
       move right
       goto $Al'$
```
(6.14)

This program enlarges the ith block by an additional 1 and shifts all blocks to the right by one cell to the right.

The GOTO instruction $(l, x_i \leftarrow x_i \dot{-} 1, l')$ is simulated by the program

```
$Al$ : move right to next blank ($i$)
       move left (2)
       if read 1 then goto $Bl$
       goto $Cl$
$Bl$ : move left to next blank ($i$)
       move right
       move block right ($i-1$)
       write $b$                                          (6.15)
       move left to next blank ($i-1$)
       move right
       goto $Al'$
$Cl$ : move left to next blank ($i-1$)
       move right
       goto $Al'$
```

This subroutine shortens the ith block by a single 1 if it contains at least two 1's which corresponds to a non-zero number. If so, all blocks to the left are shifted by one cell to the right. Otherwise, the blocks are left unchanged.

The GOTO instruction (l, if $x_i = 0, l', l''$) is simulated by the program

```
$Al$ :  move right to next blank ($i$)
        move left (2)
        if read 1 then goto $Bl$
        goto $Cl$
$lBl$ : move left to next blank ($i$)
        move right                                        (6.16)
        goto $Al''$
$Cl$ :  move left to next blank ($i-1$)
        move right
        goto $Al'$
```

This program checks if the ith block contains one or more 1's and jumps accordingly to the label Al' or Al''.

Note that when one of these subroutines ends, the read/write head always points to the first cell of the first block.

Suppose the routine `simulate` starts with the configuration (6.11) and terminates; this will exactly be the case when the input $\boldsymbol{x} = (x_1, \ldots, x_k)$ belongs to the domain of the function f. In this case, the tape will contain in the first block the unary encoding of the result $f(\boldsymbol{x})$:

$$\begin{array}{l} - \, b \, \overbrace{11\ldots1}^{f(\boldsymbol{x})+1} \, b \, \overbrace{11\ldots1}^{y_2+1} \, b \ldots b \, \overbrace{11\ldots1}^{y_n+1} \, b \, - \\ \quad\;\uparrow \end{array} \qquad (6.17)$$

Finally, the subroutine `clean_up` will rectify the tape such that upon termination the tape will only contain $f(\boldsymbol{x})$ ones. This will be achieved by the code

```
clean: move right to next blank $(n)$
       move left
       erase $(n-1)$
       write $b$                                  (6.18)
       move left to next blank
       move right
```

This subroutine produces the tape:

$$\begin{array}{l} — \; b \; \overbrace{11\ldots1}^{f(\boldsymbol{x})} \; b \; — \\ \uparrow \end{array} \tag{6.19}$$

This kind of simulation captures the notion of Post-Turing computability. A function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is *Post-Turing computable* if there is a Post-Turing program P such that for each input $\boldsymbol{x} = (x_1, \ldots, x_k)$ in the domain of f, the program P started with the initial tape (6.11) will terminate with the final tape (6.19); otherwise, the program P will not stop. Summing up, the following result has been established.

Theorem 6.2. *Each GOTO computable function is Post-Turing computable.*

6.4 Gödel Numbering of Post-Turing Programs

This section provides the Gödel numbering of Post-Turing programs similar to that of SGOTO programs. This numbering will be used to show that Post-Turing computable functions are partial recursive.

To this end, let $\Sigma_{s+1} = \{b = a_0, a_1, \ldots, a_s\}$ be the tape alphabet containing the blank symbol b, and let $P = \sigma_0; \sigma_1; \ldots; \sigma_q$ be a Post-Turing program given by a sequence of instructions σ_j, $0 \leq j \leq q$. A *configuration* of the Post-Turing program P consists of the content of the tape, the position of the read/write head, and the instruction σ_j to be performed. Such a configuration can be pictured as follows:

$$\begin{array}{c} — \; a_{i_1} \; a_{i_2} \; \ldots \; a_{i_v} \; \ldots \; a_{i_t} \; — \\ \uparrow \\ \sigma_j \end{array} \tag{6.20}$$

where all cells to the left of a_{i_1} and to the right of a_{i_t} contain the blank symbol.

A *Gödel numbering* of such a configuration can be considered as a triple (u, v, j) consisting of

- the Gödel numbering $u = J(i_1, i_2, \ldots, i_t)$ of the content of the tape,
- the position v of the read/write head, with $1 \leq v \leq t = \mu k(K^k(u) = 0)$, and
- the number j of the next instruction σ_j.

First, define the *one-step function* $E : \mathbb{N}_0^3 \to \mathbb{N}_0^3$ that describes the process of moving from one configuration to the next one during the computation of program P. For this, let $\boldsymbol{z} = (u, v, j)$ be a configuration of the program given as in (6.20). The configuration $\boldsymbol{z}' = (u', v', j')$ is *reached in one step* from $\boldsymbol{z}$, written $\boldsymbol{z} \vdash_P \boldsymbol{z}'$, if one of the following holds:

- If σ_j is `write` a_i,

$$(u',v',j') = (J(i_1,\ldots,i_{v-1},i,i_{v+1},\ldots,i_t),v,j+1). \tag{6.21}$$

- If σ_j is `move left`,

$$(u',v',j') = \begin{cases} (u,v-1,j+1) & \text{if } v > 1, \\ (J(0,i_1,\ldots,i_t),v,j+1) & \text{otherwise.} \end{cases} \tag{6.22}$$

- If σ_j is `move right`,

$$(u',v',j') = \begin{cases} (u,v+1,j+1) & \text{if } v < \mu k(K^k(u)=0), \\ (J(i_1,\ldots,i_t,0),v+1,j+1) & \text{otherwise.} \end{cases} \tag{6.23}$$

- If σ_j is `if read` a_i `then goto` A, where the label A is given as an instruction number,

$$(u',v',j') = \begin{cases} (u,v,A) & \text{if } i_v[=L(K^{t-v}(u)] = i, \\ (u,v,j+1) & \text{otherwise.} \end{cases} \tag{6.24}$$

The function $E : \boldsymbol{z} \mapsto \boldsymbol{z}'$ is defined by cases and primitive recursive operations. It follows that E is primitive recursive.

Second, the execution of the Post-Turing program P is given by a sequence $\boldsymbol{z}_0, \boldsymbol{z}_1, \boldsymbol{z}_2, \ldots$ of configurations such that $\boldsymbol{z}_i \vdash_P \boldsymbol{z}_{i+1}$ for each $i \geq 0$. During this process, a configuration $\boldsymbol{z}_t$ may eventually be reached such that the label of the involved instruction does not belong to the set of instruction numbers $L(P) = \{0,1,\ldots,q\}$. In this case, the program P terminates. Such an event may eventually not happen. In this way, the *runtime function* of P is a partial function $Z_P : \mathbb{N}_0^3 \to \mathbb{N}_0$ given by

$$Z_P : \boldsymbol{z} \mapsto \begin{cases} \min\{t \in \mathbb{N}_0 \mid (\pi_2^{(3)} \circ E_P^t)(\boldsymbol{z}) \notin L(P)\} & \text{if } \{\ldots\} \neq \emptyset, \\ \uparrow & \text{otherwise.} \end{cases} \tag{6.25}$$

The runtime function may not be primitive recursive as it corresponds to an unbounded search process. Claim that the runtime function Z_P is partial recursive. Indeed, the one-step function E_P is primitive recursive and thus its iteration is primitive recursive:

$$E_P' : \mathbb{N}_0^4 \to \mathbb{N}_0^3 : (\boldsymbol{z},t) \mapsto E_P^t(\boldsymbol{z}). \tag{6.26}$$

Moreover, the set $L(P)$ is finite and so the characteristic function $\chi_{L(P)}$ is primitive recursive. Therefore, the following function is also primitive recursive:

$$E_P'' : \mathbb{N}_0^4 \to \mathbb{N}_0 : (\boldsymbol{z},t) \mapsto (\chi_{L(P)} \circ \pi_2^{(3)} \circ E_P')(\boldsymbol{z},t). \tag{6.27}$$

This function has the property that

$$E_P''(\boldsymbol{z},t) = \begin{cases} 1 \text{ if } E_P^t(\boldsymbol{z}) = (u,v,j) \text{ and } j \in L(P), \\ 0 \text{ otherwise.} \end{cases} \tag{6.28}$$

By definition, $Z_P = \mu E_P''$ and thus the claim follows.

The *residual step function* of the GOTO program P is given by the partial function $R_P : \mathbb{N}_0^3 \to \mathbb{N}_0^3$ that maps an initial configuration to the final configuration of the computation, if any:

$$R_P(\boldsymbol{z}) = \begin{cases} E'_P(\boldsymbol{z}, Z_P(\boldsymbol{z})) & \text{if } \boldsymbol{z} \in \operatorname{dom}(Z_P), \\ \uparrow & \text{otherwise.} \end{cases} \tag{6.29}$$

This function is also partial recursive, since for each $\boldsymbol{z} \in \mathbb{N}_0^3$, $R_P(\boldsymbol{z}) = E'_P(\boldsymbol{z}, (\mu E''_P)(\boldsymbol{z}))$.

Define the total functions

$$\alpha_k : \mathbb{N}_0^k \to \mathbb{N}_0^3 : (x_1, \ldots, x_k) \mapsto (J(x_1, \ldots, x_k), 1, 0) \tag{6.30}$$

and

$$\omega_1 : \mathbb{N}_0^3 \to \mathbb{N}_0 : (u, v, j) \mapsto J^{-1}(u). \tag{6.31}$$

Both functions are primitive recursive; α_k and ω_1 provide the initial configuration and the result of the computation, respectively. For each arity $k \in \mathbb{N}_0$, the Post-Turing program P provides the partial recursive function

$$\|P\|_{k,1} = \omega_1 \circ R_P \circ \alpha_k. \tag{6.32}$$

It follows that each Post-Turing computable function is partial recursive. Hence, the Theorems 3.13 and 6.2 yield the following.

Theorem 6.3. *The class of Post-Turing computable functions equals the class of partial recursive functions.*

7

Undecidability

In computability theory, undecidable problems refer to decision problems which are yes-or-no questions on an input set. An undecidable problem does not allow to construct a general algorithm that always leads to a correct yes-or-no answer. The most prominent example is the halting problem. This chapter introduces undecidable sets, semidecidable sets, the theorems of Rice and Rice-Shapiro, and Kleene's recursion theorems.

7.1 Undecidable Sets

A set of natural numbers is decidable, computable or recursive if there is an algorithm which terminates after a finite amount of time and correctly decides whether or not a given input number belongs to the set. More formally, a set A in $\mathbb{N}_0^k$ is called *decidable* if its characteristic function χ_A is recursive. An algorithm for the computation of χ_A is called a *decision procedure* for A. A set A which is not decidable is called *undecidable.*

Example 7.1.

- Every finite set A of natural numbers is computable, since

$$\chi_A(x) = \text{sgn} \circ \sum_{a \in A} \chi_=(a, x), \quad x \in \mathbb{N}_0. \tag{7.1}$$

  In particular, the empty set is computable.
- The entire set of natural numbers is computable, since $\mathbb{N}_0 = \overline{\emptyset}$ (see Proposition 7.2).
- The set of prime numbers is computable (see Proposition 2.38).

◊

Proposition 7.2. *Let A and B be subsets of $\mathbb{N}_0$.*

- *If A is decidable, the complement $\overline{A}$ of A is decidable.*
- *If A and B are decidable, the sets $A \cup B$, $A \cap B$, and $A \setminus B$ are decidable.*

Proof. We have

$$\begin{aligned}
\chi_{\overline{A}} &= \mathrm{csg} \circ \chi_A, &(7.2)\\
\chi_{A\cup B} &= \mathrm{sgn} \circ (\chi_A + \chi_B), &(7.3)\\
\chi_{A\cap B} &= \chi_A \cdot \chi_B, &(7.4)\\
\chi_{A\setminus B} &= \chi_A \cdot \chi_{\overline{B}}. &(7.5)
\end{aligned}$$

□

Now it is time for a first encounter with an undecidable set. The way to prove undecidability is to use *diagonalization* similar to Georg Cantor's (1845-1918) famous proof showing that the set of real numbers is not denumerable. Another way is show undecidability is to apply *reduction* which requires the existence of an undecidable set.

Proposition 7.3. *The set $K = \{x \in \mathbb{N}_0 \mid x \in \operatorname{dom} \phi_x\}$ is undecidable.*

Proof. Assume the set K would be decidable; i.e., the function χ_K would be recursive. Then the function $f : \mathbb{N}_0 \to \mathbb{N}_0$ given by

$$f(x) = \begin{cases} 0 \text{ if } \chi_K(x) = 0, \\ \uparrow \text{ if } \chi_K(x) = 1, \end{cases} \tag{7.6}$$

is partial recursive. To see this, take the function $g : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by $g(x, y) = \chi_K(x)$. The function g is recursive and has the property that $f = \mu g$. Thus f is partial recursive and so has an index e, i.e., $f = \phi_e$. Then $e \in \operatorname{dom} \phi_e$ is equivalent to $f(e) = 0$, which in turn is equivalent to $e \notin K$, which means that $e \notin \operatorname{dom} \phi_e$ contradicting the hypothesis. □

Note that in opposition to the function f used in the proof, the function $h : \mathbb{N}_0 \to \mathbb{N}_0$ defined by exchanging the cases

$$h(x) = \begin{cases} 0 \text{ if } \chi_K(x) = 1, \\ \uparrow \text{ if } \chi_K(x) = 0, \end{cases} \tag{7.7}$$

is partial recursive. To see this, observe that for each $x \in \mathbb{N}_0$,

$$h(x) = 0 \cdot \phi_x(x) = 0 \cdot \psi_{\text{univ}}^{(1)}(x, x). \tag{7.8}$$

Moreover, the related function $h' : \mathbb{N}_0 \to \mathbb{N}_0$ given by

$$h'(x) = \begin{cases} x \text{ if } \chi_K(x) = 1, \\ \uparrow \text{ if } \chi_K(x) = 0, \end{cases} \tag{7.9}$$

is partial recursive. Indeed, for each $x \in \mathbb{N}_0$,

$$h'(x) = x \cdot \mathrm{sgn}(\phi_x(x) + 1) = x \cdot \mathrm{sgn}(\phi_{\text{univ}}^{(1)}(x, x) + 1). \tag{7.10}$$

It is interesting to note that the domain and range of the partial recursive function h' are undecidable sets, since

$$\operatorname{dom} h' = K = \operatorname{ran} h'. \tag{7.11}$$

Now an undecidable set is at hand and so any other set can be proved to be undecidable by using reduction. More specifically, a subset A of $\mathbb{N}_0^k$ is said to be *reducible* to a subset B of $\mathbb{N}_0^l$ if there is a recursive function $f : \mathbb{N}_0^k \to \mathbb{N}_0^l$, called *reduction function*, such that

$$\boldsymbol{x} \in A \quad \Longleftrightarrow \quad f(\boldsymbol{x}) \in B, \quad \boldsymbol{x} \in \mathbb{N}_0^k. \tag{7.12}$$

This assertion is equivalent to

$$\chi_A(\boldsymbol{x}) = \chi_B(f(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathbb{N}_0^k. \tag{7.13}$$

This means that if B is decidable, A is also decidable; or by contraposition, if A is undecidable, B is also undecidable.

Note that a (multi-valued) function $f : \mathbb{N}_0^k \to \mathbb{N}_0^l$ is *recursive* if all coordinate functions $\pi_i^{(l)} \circ f : \mathbb{N}_0^k \to \mathbb{N}_0$, $1 \leq i \leq l$, are recursive.

The *halting problem* is one of the famous undecidabilty results. It states that given a program and an input to the program, decide whether the program finishes or continues to run forever when run with that input. Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs of a Turing machine cannot exist. By Church's thesis, the halting problem is undecidable not only for Turing machines but for any formalism capturing the notion of computability.

Proposition 7.4. *The set* $H = \{(x,y) \in \mathbb{N}_0^2 \mid y \in \operatorname{dom} \phi_x\}$ *is undecidable.*

Proof. The set K can be reduced to the set H by the function $f : \mathbb{N}_0 \to \mathbb{N}_0^2$ given by $x \mapsto (x,x)$. Indeed, $\chi_K(x) = \chi_H(x,x) = \chi_H(f(x))$ for any value $x \in \mathbb{N}_0$. But the set K is undecidable and so is H. □

Next, several interesting undecidability results are presented.

Proposition 7.5. *The set* $C = \{x \in \mathbb{N}_0 \mid \phi_x = c_0^{(1)}\}$ *is undecidable.*

Proof. Take the function $f : \mathbb{N}_0^2 \to \mathbb{N}_0$ given by

$$f(x,y) = \begin{cases} 0 \text{ if } \chi_K(x) = 1, \\ \uparrow \text{ if } \chi_K(x) = 0. \end{cases} \tag{7.14}$$

This function is partial recursive, since it can be written as $f = h \circ \pi_1^{(2)}$, where h is the function given in (7.7). By the smn theorem, there is a monadic recursive function g such that $f(x,y) = \phi_{g(x)}(y)$ for all $x, y \in \mathbb{N}_0$. Consider two cases:

- If $x \in K$, then $f(x,y) = 0$ for all $y \in \mathbb{N}_0$ and so $\phi_{g(x)}(y) = c_0^{(1)}(y)$ for all $y \in \mathbb{N}_0$. Hence, $g(x) \in C$.
- If $x \notin K$, then $f(x,y)$ is undefined for all $y \in \mathbb{N}_0$ and thus $\phi_{g(x)}(y)$ is undefined for all $y \in \mathbb{N}_0$. Hence, $g(x) \notin C$.

It follows that the recursive function g provides a reduction of the set K to the set C. But K is undecidable and so C is also undecidable. □

Proposition 7.6. *The set* $E = \{(x, y) \in \mathbb{N}_0^2 \mid \phi_x = \phi_y\}$ *is undecidable.*

Proof. Let c be an index for the function $c_0^{(1)}$; i.e., $\phi_c = c_0^{(1)}$. Define the function $f : \mathbb{N}_0 \to \mathbb{N}_0^2$ given by $f(x) = (x, c)$. This function is clearly primitive recursive. Moreover, for each $x \in \mathbb{N}_0$, $x \in C$ is equivalent to $\phi_x = c_0^{(1)}$ which in turn is equivalent to $f(x) \in E$. Thus the recursive function f provides a reduction of the set C to the set E. Since C is undecidable, it follows that E is undecidable. □

Proposition 7.7. *For each number* $a \in \mathbb{N}_0$, *the sets* $I_a = \{x \in \mathbb{N}_0 \mid a \in \operatorname{dom} \phi_x\}$ *and* $O_a = \{x \in \mathbb{N}_0 \mid a \in \operatorname{ran} \phi_x\}$ *are undecidable.*

Proof. Take the function $f : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by

$$f(x, y) = \begin{cases} y \text{ if } \chi_K(x) = 1, \\ \uparrow \text{ if } \chi_K(x) = 0. \end{cases} \tag{7.15}$$

This function is partial recursive, since it can be written as

$$f(x, y) = y \cdot \operatorname{sgn}(\phi_x(x) + 1) = y \cdot \operatorname{sgn}(\psi_{\text{univ}}^{(1)}(x, x) + 1). \tag{7.16}$$

By the smn theorem, there is a monadic recursive function g such that

$$f(x, y) = \phi_{g(x)}(y), \quad x, y \in \mathbb{N}_0. \tag{7.17}$$

Consider two cases:

- If $x \in K$, then $f(x, y) = y$ for all $y \in \mathbb{N}_0$ and thus $\operatorname{dom} \phi_{g(x)} = \mathbb{N}_0 = \operatorname{ran} \phi_{g(x)}$.
- If $x \notin K$, then $f(x, y)$ is undefined for all $y \in \mathbb{N}_0$ and so $\operatorname{dom} \phi_{g(x)} = \emptyset = \operatorname{ran} \phi_{g(x)}$.

It follows that for each $a \in \mathbb{N}_0$, $x \in K$ is equivalent to both, $g(x) \in I_a$ and $g(x) \in O_a$. Thus the recursive function g provides a simultaneous reduction of K to both, I_a and O_a. Since the set K is undecidable, the result follows. □

Proposition 7.8. *The set* $T = \{x \in \mathbb{N}_0 \mid \phi_x$ *is total* $\}$ *is undecidable.*

Proof. Assume that T would be decidable. Pick the function $f : \mathbb{N}_0 \to \mathbb{N}_0$ defined as

$$f(x) = \begin{cases} \phi_x(x) + 1 & \text{if } \chi_T(x) = 1, \\ 0 & \text{if } \chi_T(x) = 0. \end{cases} \tag{7.18}$$

This function is recursive, since $\phi_x(x)$ is only evaluated if ϕ_x is total. Thus there is an index $e \in T$ such that $f = \phi_e$. But then $\phi_e(e) = f(e) = \phi_e(e) + 1$ yields a contradiction. □

The undecidability results established so far have a number of practical implications, which will be briefly summarized using the formalism of GOTO programs:

- The problem whether an GOTO program halts with a given input is undecidable.
- The problem whether an GOTO program computes a specific function (here $c_0^{(1)}$) is undecidable.

- The problem whether two GOTO programs and are semantically equivalent, i.e., exhibit the same input-output behaviour, is undecidable.
- The problem whether an GOTO program halts for a specific input is undecidable.
- The problem whether an GOTO program always halts is undecidable.

These undecidability results refer to the input-output behaviour or the semantics of GOTO programs. The following result (1953) due to Henry Gordon Rice (born 1920) is a milestone in computability theory. It states that for any non-trivial property of partial recursive functions, there is no general and effective method to decide whether a partial recursive function has this property or not. Here a property of partial recursive functions is called *trivial* if it holds for all partial recursive functions or for none of them.

Theorem 7.9 (Rice). *Let $\mathcal{A}$ be a class of monadic partial recursive functions given by a non-trivial monadic predicate P; i.e., $\mathcal{A} = \{\phi_x \mid P(x)\}$. Then the corresponding index set*

$$\mathrm{prog}(\mathcal{A}) = \{x \in \mathbb{N}_0 \mid \phi_x \in \mathcal{A}\} \tag{7.19}$$

is undecidable.

Proof. By Proposition 7.2, if a set is decidable, its complement is also decidable. Therefore, it may be assumed that the nowhere-defined function $f_\uparrow$ does not belong to $\mathcal{A}$. Take any function $f \in \mathcal{A}$ and define the function $h : \mathbb{N}_0^2 \to \mathbb{N}_0$ as follows:

$$h(x,y) = \begin{cases} f(y) & \text{if } \chi_K(x) = 1, \\ \uparrow & \text{if } \chi_K(x) = 0. \end{cases} \tag{7.20}$$

This function is partial recursive, since

$$h(x,y) = f(y) \cdot \mathrm{sgn}(\phi_x(x) + 1) = f(y) \cdot \mathrm{sgn}(\psi_{\mathrm{univ}}^{(1)}(x,x) + 1). \tag{7.21}$$

Therefore by the smn theorem, there is a monadic recursive function g such that

$$h(x,y) = \phi_{g(x)}(y), \quad x, y \in \mathbb{N}_0^{.} \tag{7.22}$$

Consider two cases:

- If $x \in K$, then $h(x,y) = f(y)$ for all $y \in \mathbb{N}_0$ and thus $\phi_{g(x)} = f$. Hence, $g(x) \in \mathrm{prog}(\mathcal{A})$.
- If $x \notin K$, then $h(x,y)$ is undefined for all $y \in \mathbb{N}_0$ and so $\phi_{g(x)} = f_\uparrow$. Hence, by hypothesis, $g(x) \notin \mathrm{prog}(\mathcal{A})$.

Therefore, the recursive function g reduces the set K to the set $\mathrm{prog}(\mathcal{A})$. Since the set K is undecidable, the result follows. □

7.2 Semidecidable Sets

A set A of natural numbers is called computably enumerable, semidecidable or provable if there is an algorithm such that the set of input numbers for which the algorithm halts is exactly the set of numbers in A. More specifically, a subset A of $\mathbb{N}_0^k$ is called *semidecidable* if the function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ defined by

$$f(\boldsymbol{x}) = \begin{cases} 1 \text{ if } \boldsymbol{x} \in A, \\ \uparrow \text{ otherwise}, \end{cases} \tag{7.23}$$

is partial recursive.

Proposition 7.10. *A subset A of $\mathbb{N}_0^k$ is semidecidable if and only if the set A is the domain of a k-ary partial recursive function.*

Proof. Let A be semidecidable. Then the corresponding function f given in (7.23) has the property that $\operatorname{dom} f = A$. Conversely, let A be a subset of $\mathbb{N}_0^k$ for which there is a partial computable function $h : \mathbb{N}_0^k \to \mathbb{N}_0$ with the property that $\operatorname{dom} h = A$. Then the function $f = \nu \circ c_0^{(1)} \circ h$ is also partial recursive and coincides with function in (7.23). Hence the set A is semidecidable. □

Example 7.11. The prototype set K is semidecidable as it is the domain of the partial recursive function in (7.7). ♢

A program for the function f given in (7.23) provides a *partial decision procedure* for A:

> Given $\boldsymbol{x} \in \mathbb{N}_0^k$. If $\boldsymbol{x} \in A$, the program started with input $\boldsymbol{x}$ will halt giving a positive answer. Otherwise, the program will not terminate in a finite number of steps.

Proposition 7.12. *Each decidable set is semidecidable.*

Proof. Let A be a decidable subset of $\mathbb{N}_0^k$. Then the function $g : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ defined by

$$g(\boldsymbol{x}, y) = (\text{csg} \circ \chi_A)(\boldsymbol{x}) = \begin{cases} 0 \text{ if } \boldsymbol{x} \in A, \\ 1 \text{ otherwise}, \end{cases} \tag{7.24}$$

is recursive. Thus the function $f = \mu g$ is partial recursive. It has the property that $\mu g(\boldsymbol{x}) = y$ if $g(\boldsymbol{x}, y) = 0$ and $g(\boldsymbol{x}, i) \neq 0$ for all $0 \leq i < y$, and $\mu g(\boldsymbol{x})$ is undefined otherwise. It follows that $\mu g(\boldsymbol{x}) = 0$ if $\boldsymbol{x} \in A$ and $\mu g(\boldsymbol{x})$ is undefined otherwise. Hence, $A = \operatorname{dom} f$ as required. □

Proposition 7.13. *The halting problem is semidecidable.*

Proof. Consider the corresponding set H. The universal function $\psi_{\text{univ}}^{(1)}$ has the property

$$\psi_{\text{univ}}^{(1)}(x, y) = \begin{cases} \phi_x(y) & \text{if } y \in \operatorname{dom} \phi_x, \\ \uparrow & \text{otherwise}. \end{cases} \tag{7.25}$$

It follows that $H = \operatorname{dom} \psi_{\text{univ}}^{(1)}$ as required. □

Proposition 7.14. *Let A be a subset of $\mathbb{N}_0^k$. If A is reducible to a semidecidable set, then A is semidecidable.*

Proof. Suppose A is reducible to a semidecidable subset B of $\mathbb{N}_0^l$. Then there is a recursive function $f : \mathbb{N}_0^k \to \mathbb{N}_0^l$ such that $\boldsymbol{x} \in A$ if and only if $f(\boldsymbol{x}) \in B$. Moreover, there is a partial recursive function $g : \mathbb{N}_0^l \to \mathbb{N}_0$ such that $B = \operatorname{dom} g$. Thus the composite function $g \circ f : \mathbb{N}_0^k \to \mathbb{N}_0$ is partial recursive. Furthermore, for each $\boldsymbol{x} \in \mathbb{N}_0^k$, $\boldsymbol{x} \in A$ is equivalent to $f(\boldsymbol{x}) \in B$ which in turn is equivalent that $g(f(\boldsymbol{x}))$ is defined. Hence, $A = \operatorname{dom} g \circ f$ as required. □

The next assertion states that each semidecidable set results from a decidable one by unbounded existential quantification. That is, a partial decision procedure can be formulated as an unbounded search to satisfy a decidable relation.

Proposition 7.15. *A set A is semidecidable if and only if there is a decidable set B such that $A = \exists y[(\boldsymbol{x}, y) \in B]$.*

Proof. Let B be a decidable subset of $\mathbb{N}_0^{k+1}$ and $A = \exists y[(\boldsymbol{x}, y) \in B]$. Consider the function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ given by

$$f(\boldsymbol{x}) = \mu(\text{csg} \circ \chi_B)(\boldsymbol{x}) = \begin{cases} 0 \text{ if } (\boldsymbol{x}, y) \in B \text{ for some } y \in \mathbb{N}_0, \\ \uparrow \text{otherwise.} \end{cases} \tag{7.26}$$

This function is partial recursive and has the property that $\operatorname{dom} f = A$.

Conversely, let A be a semidecidable subset of $\mathbb{N}_0^k$. Then there is an index e such that $\operatorname{dom} \phi_e^{(k)} = A$. By Kleene's normal form theorem, an element $\boldsymbol{x} \in \mathbb{N}_0^k$ satisfies $\boldsymbol{x} \in A$ if and only if $\boldsymbol{x} \in \exists y[(e, \boldsymbol{x}, y) \in T_k]$, where the index e is kept fixed. Hence, $A = \exists y[(e, \boldsymbol{x}, y) \in T_k]$ as required. □

The next result shows that the class of semidecidable sets is closed under unbounded existential quantification.

Proposition 7.16. *If B is semidecidable, then $A = \exists y[(\boldsymbol{x}, y) \in B]$ is semidecidable.*

Proof. Let B be a semidecidable subset of $\mathbb{N}_0^{k+1}$. By Proposition 7.15, there is a decidable subset C of $\mathbb{N}_0^{k+2}$ such that $B = \exists z[(\boldsymbol{x}, y, z) \in C]$. But the search for a pair (y, z) of numbers with $(\boldsymbol{x}, y, z) \in C$ can be replaced by the search for a number u such that $(\boldsymbol{x}, K_2(u), L_2(u)) \in C$. It follows that $A = \exists u[(\boldsymbol{x}, K_2(u), L_2(u)) \in C]$. Thus by Proposition 7.15, the set A is semidecidable. □

It follows that the class of semidecidable sets is closed under existential quantification. This is not true for the class of decidable sets. To see this, take the Kleene predicate T_1 which is primitive recursive by the Kleene normal form theorem. The prototype set K, which is semidecidable but not decidable, results from T_1 by existential quantification as follows:

$$K = \exists y[(x, x, y) \in T_1]. \tag{7.27}$$

Another useful connection between decidable and semidecidable sets is the following.

Proposition 7.17. *A set A is decidable if and only if A and $\overline{A}$ are semidecidable.*

Proof. If A is decidable, then by Proposition 7.2, the set $\overline{A}$ is also decidable. But each decidable set is semidecidable and so A and $\overline{A}$ are semidecidable.

Conversely, if A and $\overline{A}$ are semidecidable, a decision procedure for A can be established by simultaneously applying the partial decision procedures for A and $\overline{A}$. One of these procedures will provide a positive answer in a finite number of steps giving an answer to the decision procedure for A. □

Example 7.18. The complement of the halting problem is given by the set

$$\overline{H} = \{(x, y) \in \mathbb{N}_0^2 \mid y \notin \operatorname{dom} \phi_x\}. \tag{7.28}$$

This set is not semidecidable, since the set H is semidecidable but not decidable. ♢

Recall that the *graph* of a partial function $f : \mathbb{N}_0^n \to \mathbb{N}_0$ is given as

$$\operatorname{graph}(f) = \{(\boldsymbol{x}, y) \in \mathbb{N}_0^{n+1} \mid f(\boldsymbol{x}) = y\}. \tag{7.29}$$

Proposition 7.19. *A function $f : \mathbb{N}_0^n \to \mathbb{N}_0$ is partial recursive if and only if the graph of f is semidecidable.*

Proof. Suppose the function f is partially recursive. Then there is an index e for f, i.e., $f = \phi_e^{(n)}$. The extended Kleene set S_n used to derive Kleene's normal form shows that $f(\boldsymbol{x}) = y$ is equivalent to $(\boldsymbol{x}, y) \in \exists t[(e, \boldsymbol{x}, y, t) \in S_n]$. Thus the set graph$(f)$ is obtained from the decidable set S_n with e being fixed by existential quantification and so is semidecidable.

Conversely, let the set graph(f) be semidecidable. Then there is a decidable set $A \subseteq \mathbb{N}_0^{n+2}$ such that graph(f) has the form $\exists z[(\boldsymbol{x}, y, z) \in A]$. To compute the function f, take an argument $\boldsymbol{x} \in \mathbb{N}_0^n$ and systematically search for a pair $(y, z) \in \mathbb{N}_0^2$ such that $(\boldsymbol{x}, y, z) \in A$, say by listing the elements of $\mathbb{N}_0^2$ as in (2.42). If such a pair exists, put $f(\boldsymbol{x}) = y$. Otherwise, $f(\boldsymbol{x})$ is undefined. □

7.3 Recursively Enumerable Sets

In this section, we restrict our attention to sets of natural numbers. To this end, the terminology will be changed a bit. A set A of natural numbers is called *recursive* if its characteristic function χ_A is recursive, and a set A of natural numbers is called *recursively enumerable* (*r.e.* for short) if its characteristic function χ_A is partial recursive.

The first result provides the relationship between decidable and recursive sets as well as semidecidable and recursively enumerable sets.

Proposition 7.20. *Let $J_k : \mathbb{N}_0^k \to \mathbb{N}_0$ be a primitive recursive bijection.*

- *A subset A of $\mathbb{N}_0^k$ is decidable if and only if $J_k(A)$ is recursive.*
- *A subset A of $\mathbb{N}_0^k$ is semidecidable if and only if $J_k(A)$ is recursively enumerable.*

Proof. Let $J_k(A)$ be recursive. Then for each number $x \in \mathbb{N}_0$,

$$\begin{aligned}
\chi_{J_k(A)}(x) &= \begin{cases} 1 \text{ if } x \in J_k(A), \\ 0 \text{ otherwise,} \end{cases} \\
&= \begin{cases} 1 \text{ if } \exists \boldsymbol{a} \in \mathbb{N}_0^k :\ J_k(\boldsymbol{a}) = x \wedge \chi_A(\boldsymbol{a}) = 1, \\ 0 \text{ otherwise,} \end{cases} \\
&= \begin{cases} 1 \text{ if } \chi_A \circ J_k^{-1}(x) = 1, \\ 0 \text{ otherwise.} \end{cases}
\end{aligned}$$

Thus $\chi_{J_k(A)} = \chi_A \circ J_k^{-1}$ and hence χ_A is recursive. Conversely, if A is recursive, then $\chi_A = \chi_{J_k(A)} \circ J_k$ and so $\chi_{J_k(A)}$ is recursive.

Second, if A is semidecidable given by the domain of a partial recursive function $f : \mathbb{N}_0^k \to \mathbb{N}_0$, the function $g = f \circ J_k^{-1}$ is partial recursive and has the domain $J_k(A)$. Conversely, if $J_k(A)$ is semidecidable defined by the domain of a partial recursive function $g : \mathbb{N}_0 \to \mathbb{N}_0$, the function $f = g \circ J_k$ is partial recursive with domain A. □

Thus it is sufficient to consider subsets of $\mathbb{N}_0$ instead of subsets of $\mathbb{N}_0^k$ for any $k \geq 1$. Next, closure properties of recursive sets are studied.

Proposition 7.21. *If A and B are recursive sets, the sets $\overline{A}$, $A \cup B$, $A \cap B$, and $A \setminus B$ are recursive.*

Proof. Let the functions χ_A and χ_B be recursive. Then the functions $\chi_{\overline{A}} = \text{csg} \circ \chi_A$, $\chi_{A \cup B} = \text{sgn} \circ (\chi_A + \chi_B)$, $\chi_{A \cap B} = \chi_A \cdot \chi_B$, and $\chi_{A \setminus B} = \chi_A \cdot \chi_{\overline{B}}$ are also recursive. □

The Gödel numbering of SGOTO programs yields an enumeration of all monadic partial recursive functions

$$\phi_0, \phi_1, \phi_2, \ldots \tag{7.30}$$

By taking the domains of these functions, i.e., $D_e = \text{dom}\,\phi_e$, this list provides an enumeration of all recursively enumerable sets

$$D_0, D_1, D_2, \ldots \tag{7.31}$$

Let A be a recursively enumerable set. Then there is a Gödel number $e \in \mathbb{N}_0$ such that $D_e = A$. The number e is called an *index* for A.

Proposition 7.22. *For each set A of natural numbers, the following assertions are equivalent:*

- *A is recursively enumerable.*
- *$A = \emptyset$ or there is a monadic recursive function f with $A = \text{ran}\, f$.*
- *There is a k-ary partial recursive function g with $A = \text{ran}\, g$.*

Proof.

- Let A be a recursively enumerable set. First, let A be the empty set. Then $A = \emptyset = \text{dom} f_\uparrow$ for the nowhere-defined function $f_\uparrow$ and so the empty set is recursively enumerable. Second, let A be nonempty. By the above discussion, the set A has an index e. Fix an element $a \in A$ and use Kleene's normal form theorem to define the monadic function

$$f : x \mapsto \begin{cases} K_2(x) & \text{if } (e, K_2(x), L_2(x)) \in T_1, \\ a & \text{otherwise.} \end{cases} \tag{7.32}$$

  This function is primitive recursive and since $A = \text{dom}\,\phi_e$ it follows that $A = \text{ran}\, f$ as required.
- Let $A = \emptyset$ or $A = \text{ran}\, f$ for some monadic recursive function f. Define the partial recursive function g such that g is the nowhere-defined function if $A = \emptyset$, and $g = f$ if $A \neq \emptyset$. Then $A = \text{ran}\, g$ as required.
- Let g be a k-ary partial recursive function with $A = \text{ran}\, g$. By Proposition 7.19, the set $B = \{(\boldsymbol{x}, y) \in \mathbb{N}_0^{k+1} \mid g(\boldsymbol{x}) = y\}$ is semidecidable and thus by Proposition 7.16, the set $A = \exists x_1 \ldots \exists x_k[(x_1, \ldots, x_k, y) \in B]$ is recursively enumerable. □

This result shows that a non-empty set of natural numbers A is recursively enumerable if and only if there is a monadic recursive function f that allows to enumerate the elements of A, i.e.,

$$A = \{f(0), f(1), f(2), \ldots\}. \tag{7.33}$$

Such a function f is called an *enumerator* for A. As can be seen from the proof, the enumerators can be chosen to be primitive recursive.

Proposition 7.23. *If A and B are recursively enumerable sets, the sets $A \cap B$ and $A \cup B$ are also recursively enumerable.*

Proof. First, let f and g be monadic partial recursive functions where $A = \operatorname{dom} f$ and $B = \operatorname{dom} g$. Then $f \cdot g$ is partial recursive with the property that $\operatorname{dom} f \cdot g = A \cap B$.

Second, let A and B be non-empty sets, and let f and g be monadic recursive functions where $A = \operatorname{ran} f$ and $B = \operatorname{ran} g$. Define the monadic function h as follows,

$$h : x \mapsto \begin{cases} f(\lfloor x/2 \rfloor) & \text{if } x \text{ is even,} \\ g(\lfloor x/2 \rfloor) & \text{otherwise.} \end{cases} \tag{7.34}$$

Thus $h(0) = f(0)$, $h(1) = g(0)$, $h(2) = f(1)$, $h(3) = g(1)$ and so on. The function h defined by cases is recursive and satisfies $\operatorname{ran} h = A \cup B$.

□

Proposition 7.24. *An infinite set A of natural numbers is recursive if and only if the set A has a strictly monotonous enumerator, i.e., $A = \operatorname{ran} f$ with $f(0) < f(1) < f(2) < \ldots$.*

Proof. Let A be an infinite recursive set. Define the monadic function f by minimalization and primitive recursion as follows:

$$f(0) = \mu y[y \in A], \tag{7.35}$$

$$f(n+1) = \mu y[y \in A \wedge y > f(n)]. \tag{7.36}$$

This function is recursive, strictly monotonous and satisfies $\operatorname{ran} f = A$.

Conversely, let f be a strictly monotonous monadic recursive function where $A = \operatorname{ran} f$ is infinite. Then $f(n) = y$ implies $y \geq n$ and thus

$$y \in A \quad \Longleftrightarrow \quad \exists n[n \leq y \wedge f(n) = y]. \tag{7.37}$$

The relation on the right-hand side is decidable and so A is recursive. □

Corollary 7.25. *Each infinite recursively enumerable set contains an infinite recursive subset.*

7.4 Theorem of Rice-Shapiro

The theorem of Rice provides a class of sets that are undecidable. Now we present a similar result for recursively enumerable sets called the theorem of Rice-Shapiro, which was posed by Henry Gordan Rice and proved by Norman Shapiro (born 1932).

For this, a monadic function g is called an *extension* of a monadic function f, written $f \subseteq g$, if $\operatorname{dom} f \subseteq \operatorname{dom} g$ and $f(x) = g(x)$ for all $x \in \operatorname{dom} f$. The relation of extension is an order relation on the set of all monadic functions with smallest element given by the nowhere-defined function. The maximal elements are the monadic total functions.

A monadic function f is called *finite* if its domain is finite. Each finite function f is partial recursive, since

$$f(x) = \sum_{a \in \operatorname{dom} f} \operatorname{csg}(|x - a|) f(a), \quad x \in \mathbb{N}_0. \tag{7.38}$$

Theorem 7.26 (Rice-Shapiro). *Let $\mathcal{A}$ be a class of monadic partial recursive functions whose corresponding index set* $\operatorname{prog}(\mathcal{A}) = \{x \in \mathbb{N}_0 \mid \phi_x \in \mathcal{A}\}$ *is recursively enumerable. Then a monadic partial recursive function f lies in $\mathcal{A}$ if and only if there is a finite function $g \in \mathcal{A}$ such that $g \subseteq f$.*

Proof. First, let $f \in \mathcal{A}$ and assume that no finite function g which is extended by f lies in $\mathcal{A}$. Take the recursively enumerable set $K = \{x \mid x \in \operatorname{dom} \phi_x\}$, let e be an index for K, and let P_e be a GOTO program that computes ϕ_e. Define the function

$$g : (z,t) \mapsto \begin{cases} \uparrow & \text{if } P_e \text{ computes } \phi_e(z) \text{ in } \leq t \text{ steps}, \\ f(t) & \text{otherwise.} \end{cases} \tag{7.39}$$

The function g is partial recursive. Thus by the smn theorem, there is a monadic recursive function s such that

$$g(z,t) = \phi_{s(z)}(t), \quad t, z \in \mathbb{N}_0. \tag{7.40}$$

Hence, $\phi_{s(z)} \subseteq f$ for each $z \in \mathbb{N}_0$. Consider two cases:

- If $z \in K$, the program $P_e(z)$ halts after, say t_0 steps. Then

$$\phi_{s(z)}(t) = \begin{cases} \uparrow & \text{if } t_0 \leq t, \\ f(t) & \text{otherwise.} \end{cases} \tag{7.41}$$

  Thus $\phi_{s(z)}$ is finite and hence, by hypothesis, $\phi_{s(z)}$ does not belong to $\mathcal{A}$.
- If $z \notin K$, the program $P_e(z)$ does not halt and so $\phi_{s(z)} = f$ which implies that $\phi_{s(z)} \in \mathcal{A}$.

It follows that the function s reduces the non-recursively enumerable set $\overline{K}$ to the set $\operatorname{prog}(\mathcal{A})$ and hence the set $\operatorname{prog}(\mathcal{A})$ is not recursively enumerable. A contradiction.

Conversely, let f be a monadic partial recursive function that does not belong to $\mathcal{A}$ and let g be a finite function in $\mathcal{A}$ with $g \subseteq f$. Define the function

$$h : (z,t) \mapsto \begin{cases} f(t) & \text{if } t \in \operatorname{dom} g \text{ or } z \in K, \\ \uparrow & \text{otherwise.} \end{cases} \tag{7.42}$$

The function h is partial recursive. Thus by the smn theorem, there is a monadic recursive function s such that

$$h(z,t) = \phi_{s(z)}(t), \quad t, z \in \mathbb{N}_0. \tag{7.43}$$

Consider two cases:

- If $z \in K$, $\phi_{s(z)} = f$ and so $\phi_{s(z)} \notin \mathcal{A}$.
- If $z \notin K$, $\phi_{s(z)}(t) = f(t) = g(t)$ for all $t \in \operatorname{dom} g$ and $\phi_{s(z)}$ is undefined elsewhere. Hence, $\phi_{s(z)} \in \mathcal{A}$.

It follows that the function s provides a reduction of the non-recursively enumerable set $\overline{K}$ to the set $\operatorname{prog}(\mathcal{A})$ and hence the set $\operatorname{prog}(\mathcal{A})$ is not recursively enumerable. A contradiction. □

Note that in applications the contraposition of the above assertion is used. That is, if the following condition does not hold,

$$\text{for each monadic partial recursive function } f: \ f \in \mathcal{A} \iff g \subseteq f \text{ for some finite } g \in \mathcal{A}, \tag{7.44}$$

the index set $\operatorname{prog}(\mathcal{A})$ will not be recursively enumerable.

Corollary 7.27. *Let* $\text{prog}(\mathcal{A}) = \{x \in \mathbb{N}_0 \mid \phi_x \in \mathcal{A}\}$ *be recursively enumerable. Then any extension of a function in* $\mathcal{A}$ *lies itself in* $\mathcal{A}$.

Proof. Let h be an extension of a function $f \in \mathcal{A}$. By the theorem of Rice-Shapiro, there is a finite function g which extends f. But then also h extends g and so it follows by the theorem of Rice-Shapiro that h lies in $\mathcal{A}$. □

Corollary 7.28. *Let* $\text{prog}(\mathcal{A}) = \{x \in \mathbb{N}_0 \mid \phi_x \in \mathcal{A}\}$ *be recursively enumerable. If the nowhere-defined function is in* $\mathcal{A}$*, all monadic partial recursive functions lie in* $\mathcal{A}$.

Proof. Each monadic computable function extends the nowhere-defined function and so by the theorem of Rice-Shaprio lies in $\mathcal{A}$. □

Note that the theorem of Rice is a consequence of the theorem of Rice-Shapiro. To see this, let $\mathcal{A}$ be a set of monadic partial recursive functions. Suppose the set $\text{prog}(\mathcal{A})$ would be decidable. Then by Proposition 7.17, both $\text{prog}(\mathcal{A})$ and its complement are recursive enumerable. Without restriction, we may assume that the set $\mathcal{A}$ contains the nowhere-defined function. Then by Corollary 7.28, the set $\mathcal{A}$ contains all monadic partial recursive functions and hence is non-trivial.

Example 7.29. The set $\mathcal{A} = \{\phi_x \mid \phi_x \text{ bijective}\}$ is not recursively enumerable. Indeed, suppose $\text{prog}(\mathcal{A})$ would be recursive enumerable. Then by the theorem of Rice-Shapiro, the set $\text{prog}(\mathcal{A})$ would contain a finite function. But finite functions are not bijective and so cannot belong to $\mathcal{A}$. A contradiction. ♢

7.5 Recursion Theory

Kleene's recursion theorem (1938) is a fixed-point result about the collection of indices for partial recursive functions. It states roughly that for any recursive function manipulating SGOTO programs there is an SGOTO program which is invariant under this manipulation. This result can be stated with or without parameters.

For the indexing of partial recursive functions, the following convention is used: If $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is a function such that $f(\boldsymbol{x})$ is undefined, then $\phi_{f(\boldsymbol{x})}$ is the nowhere-defined function. Using this convention, the application of a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ to the sequence of indices of n-ary partial recursive functions $(\phi_e)_{e\in\mathbb{N}_0}$ provides an *operator* $\Phi_f : (\phi_e)_{e\in\mathbb{N}_0} \to (\phi_{f(e)})_{e\in\mathbb{N}_0}$. The subsequent considerations assume that the collection of partial recursive functions $(\phi_e)_{e\in\mathbb{N}_0}$ in question forms an acceptable programming system.

Theorem 7.30 (Kleene). *For each monadic recursive function* f*, there is a number* n *such that*

$$\phi_n^{(1)} = \phi_{f(n)}^{(1)}. \tag{7.45}$$

Proof. Consider the function $g : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by

$$g(x,y) = \begin{cases} \phi^{(1)}_{f(\phi_x^{(1)}(x))}(y) & \text{if } x \in K, \\ \uparrow & \text{otherwise.} \end{cases} \tag{7.46}$$

This function is partial recursive, since $\phi^{(1)}_{f(\phi^{(1)}_x(x))}$ is well-defined for each $x \in K$. By the above convention, we can write for all $x \in \mathbb{N}_0$,

$$g(x, \cdot) = \phi^{(1)}_{f(\phi^{(1)}_x(x))}. \tag{7.47}$$

Thus by the smn theorem, there is a primitive recursive function $s : \mathbb{N}_0 \to \mathbb{N}_0$ such that for all $x \in \mathbb{N}_0$,

$$g(x, \cdot) = \phi^{(1)}_{s(x)}. \tag{7.48}$$

Note that if $f(\phi^{(1)}_x(x))$ is undefined, $s(x)$ will index a function that is nowhere defined. Since s is computable, there is an index m for s; i.e., $s = \phi^{(1)}_m$. Putting $x = m$ and $n = \phi^{(1)}_m(m)$, we obtain

$$\phi^{(1)}_{f(n)} = \phi^{(1)}_{f(\phi^{(1)}_m(m))} = \phi^{(1)}_{s(m)} = \phi^{(1)}_n. \tag{7.49}$$

□

The index n in Eq. (7.45), which is computable from an index for the function f, is called a *fixed point* for the monadic recursive function f and the corresponding operator Φ_f.

Example 7.31. Consider the increment function and the squaring function. By the recursion theorem, there are numbers m and n such that $\phi^{(1)}_m = \phi^{(1)}_{m+1}$ and $\phi^{(1)}_n = \phi^{(1)}_{n^2}$. ♢

Corollary 7.32. *For each dyadic partial recursive function f, there is a number e such that*

$$\phi^{(1)}_e = f(e, \cdot). \tag{7.50}$$

Proof. Let f be a dyadic partial recursive function. By the smn theorem, there is a primitive recursive function $s : \mathbb{N}_0 \to \mathbb{N}_0$ such that for all $x \in \mathbb{N}_0$,

$$f(x, \cdot) = \phi^{(1)}_{s(x)}. \tag{7.51}$$

Thus by the recursion theorem, there is a number e such that

$$\phi^{(1)}_e = \phi^{(1)}_{s(e)}. \tag{7.52}$$

Hence, for all $y \in \mathbb{N}_0$,

$$f(e, y) = \phi^{(1)}_e(y). \tag{7.53}$$

□

Corollary 7.33. *For each monadic recursive function f and each number $k \geq 0$, there is an index $n > k$ such that*

$$\phi^{(1)}_{f(n)} = \phi^{(1)}_n. \tag{7.54}$$

Proof. Let e be an index with $\phi_i^{(1)} \neq \phi_e^{(1)}$ for all $0 \leq i \leq k$. Define the function $g : \mathbb{N}_0 \to \mathbb{N}_0$ by

$$g(x) = \begin{cases} e & \text{if } x \leq k, \\ f(x) & \text{otherwise.} \end{cases} \tag{7.55}$$

This function is recursive. Thus by the recursion theorem, there is an index n such that $\phi_{g(n)}^{(1)} = \phi_n^{(1)}$. By definition, this cannot hold for $n \leq k$. Thus $n > k$ and hence $g(n) = f(n)$. □

Example 7.34. Consider the projection function $q : \mathbb{N}_0^2 \to \mathbb{N}_0$ given by $q(x,y) = x$. By Corollary 7.32, there is an index e such that $\phi_e^{(1)}(y) = q(e,y) = e$ for all $y \in \mathbb{N}_0$. That is, the function $\phi_e^{(1)}$ outputs its own index when applied to any input value. Such indices are called *quines.* ♢

Example 7.35. Take the primitive recursive function $f : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x,y) \mapsto y^x$. By Corollary 7.32, there is an index n such that $\phi_n^{(1)}(y) = y^n$. ♢

Example 7.36. The recursion theorem can be used to prove the undecidability of the halting problem. To see this, claim that the complement of the halting problem $\overline{H}$ is not semidecidable (Example 7.18). Indeed, suppose $\overline{H}$ would be semidecidable. Then by Proposition 7.10, there is a dyadic partial recursive function f such that $\overline{H} = \text{dom} f$. Moreover, by Corollary 7.32, there is an index e such that $\phi_e^{(1)} = f(e,\cdot)$. Thus $\phi_e(y)$ is undefined if and only if $(e,y) \in \overline{H}$. But $(e,y) \in \overline{H}$ if and only if $(e,y) \in \text{dom} f$ or equivalently $\phi_e^{(1)}(y)$ is defined. A contradiction. This proves the claim. Since $\overline{H}$ is not semidecidable, it follows from Proposition 7.17 that the halting problem H cannot be decidable. ♢

Theorem 7.37 (Kleene). *For each dyadic recursive function f, there is a monadic recursive function h such that for all $y \in \mathbb{N}_0$,*

$$\phi_{h(y)}^{(1)} = \phi_{f(h(y),y)}^{(1)}. \tag{7.56}$$

Proof. Consider the function $g : \mathbb{N}_0^3 \to \mathbb{N}_0$ defined by

$$g(x,y,z) = \begin{cases} \phi_{\phi_x^{(2)}(x,y)}^{(1)}(z) & \text{if } (x,y) \in \text{dom}\, \phi_x^{(2)}, \\ \uparrow & \text{otherwise.} \end{cases} \tag{7.57}$$

This function is partial recursive, since $\phi_{\phi_x^{(2)}(x,y)}^{(1)}$ is well-defined for each $(x,y) \in \text{dom}\, \phi_x^{(2)}$. By the above convention, we can write for all $x, y \in \mathbb{N}_0$,

$$g(x,y,\cdot) = \phi_{\phi_x^{(2)}(x,y)}^{(1)}. \tag{7.58}$$

By the smn theorem, there is a primitive recursive function $s : \mathbb{N}_0^2 \to \mathbb{N}_0$ such that for all $x, y \in \mathbb{N}_0$,

$$g(x,y,\cdot) = \phi_{s(x,y)}^{(1)}. \tag{7.59}$$

Take the function $f' : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by $f'(x,y) = f(s(x,y),y)$. It is clear that f' is recursive and thus there is an index e for f'; i.e., $f' = \phi_e^{(2)}$. Define the primitive recursive function $h : \mathbb{N}_0 \to \mathbb{N}_0$ by $h(y) = s(e,y)$ for all $y \in \mathbb{N}_0$. Then for all $y \in \mathbb{N}_0$,

$$\phi_{h(y)}^{(1)} = \phi_{s(e,y)}^{(1)} = \phi_{\phi_e^{(2)}(e,y)}^{(1)} = \phi_{f(s(e,y),y)}^{(1)} = \phi_{f(h(y),y)}^{(1)}. \tag{7.60}$$

□

The above results can be easily generalized to the case of computable functions of higher arity.

8

Word Problems

The undecidability of the halting problem has many consequences not only in computability theory but also in other branches of science. The word problems encountered in abstract algebra and formal language theory belong to the most prominent undecidability results. Hilbert's tenth problem is treated at the end of the chapter.

8.1 Semi-Thue Systems

The word problem for a set is the algorithmic problem of deciding whether two given representatives stand for the same element. In abstract algebra and formal language theory, sets have a presentation given by generators and relations which allows the word problem for a set to be decribed by utilizing its presentation.

A *string rewriting system*, historically called *semi-Thue system*, is a pair (Σ, R) where Σ is an alphabet and R is a dyadic relation on non-empty strings over Σ, i.e., $R \subseteq \Sigma^+ \times \Sigma^+$. Each element $(u, v) \in R$ is called a *rewriting rule* and is written as $u \to v$. Semi-Thue systems were introduced by the Norwegian mathematician Axel Thue (1863-1922) in 1914.

The rewriting rules can be naturally extended to strings in Σ^* by allowing substrings to be rewritten accordingly. More specifically, the *one-step rewriting relation* $\to_R$ induced by R on Σ^+ is a dyadic relation on Σ^+ such that for any strings s and t in Σ^+,

$$s \to_R t \quad :\Longleftrightarrow \quad s = xuy,\, t = xvy,\, u \to v \text{ for some } x, y \in \Sigma^*,\; u, v \in \Sigma^+. \tag{8.1}$$

That is, a string s is rewritten by a string t when there is a rewriting rule $u \to v$ such that s contains u as a substring and this substring is replaced by v giving the string t.

The pair $(\Sigma, \to_R)$ is called an *abstract rewriting system.* Such a system allows to form a finite or infinite sequence of strings which is produced by starting with an initial string $s_0 \in \Sigma^+$ and repeatedly rewriting it by using one-step rewriting. A *zero-or-more steps rewriting* or *derivation* like this is captured by the reflexive transitive closure of $\to_R$ denoted by $\overset{*}{\to}_R$. That is, for any strings $s, t \in \Sigma^+$, $s \overset{*}{\to}_R t$ if and only if $s = t$ or there is a finite sequence $s_0, s_1, \ldots, s_m$ of elements in Σ^+ such that $s_0 = s$, $s_i \to_R s_{i+1}$ for $0 \leq i \leq m-1$, and $s_m = t$.

Example 8.1. Take the semi-Thue system (Σ, R) with $\Sigma = \{a, b\}$ and $R = \{(ab, bb), (ab, a), (b, aba)\}$. The derivation $abb \rightarrow_R ab \rightarrow_R bb \rightarrow_R baba \rightarrow_R bbba$ shows that $abb \overset{*}{\rightarrow}_R bbba$. ♢

The *word problem* for semi-Thue systems asks for a semi-Thue system (Σ, R) and two strings $s, t \in \Sigma^+$, whether or not the string s can be transformed into the string t by applying the rules from R; that is, $s \overset{*}{\rightarrow}_R t$.

This problem is undecidable. To see this, the halting problem for SGOTO-2 programs will be reduced to this word problem. For this, let $P = s_0; s_1; \ldots; s_{n-1}$ be an SGOTO-2 program consisting of n instructions. We may assume that the label n, called *exit label*, does not address an instruction and is the only label signifying termination.

A *configuration* of the two-register machine is given by a triple (j, x, y), where $0 \leq j \leq n-1$ is the actual instruction number, x is the content of the first register, and y is the content of the second register. These numbers can be encoded in unary format as follows:

$$\overline{x} = \overbrace{LL\ldots L}^{x} \quad \text{and} \quad \overline{y} = \overbrace{LL\ldots L}^{y}. \tag{8.2}$$

In this way, each configuration of the two-register machine can be written as a string

$$a\overline{x}j\overline{y}b \tag{8.3}$$

over the alphabet $\Sigma = \{a, b, 0, 1, 2, \ldots, n-1, n, L\}$.

Define a semi-Thue system (Σ, R_P) that simulates the mode of operation of the SGOTO-2 program P. For this, each SGOTO-2 instruction is assigned an appropriate rewriting rule as follows:

GOTO-2 instructions	rewriting rules
$(j, x_1 \leftarrow x_1 + 1, k)$	(j, Lk)
$(j, x_2 \leftarrow x_2 + 1, k)$	(j, kL)
$(j, x_1 \leftarrow x_1 - 1, k)$	(Lj, k), (aj, ak)
$(j, x_2 \leftarrow x_2 - 1, k)$	(jL, k), (jb, kb)
$(j, \texttt{if} \;\; x_1 = 0, k, l)$	(Lj, Ll), (aj, ak)
$(j, \texttt{if} \;\; x_2 = 0, k, l)$	(jL, lL), (jb, kb)

(8.4)

Moreover, the semi-Thue system contains two clean-up rewriting rules

$$(Ln, n) \quad \text{and} \quad (anL, an). \tag{8.5}$$

Example 8.2. Consider the SGOTO-2 program P:

$$\begin{array}{l} (0, \texttt{if} \;\; x_1 = 0, 3, 1) \\ (1, x_1 \leftarrow x_1 - 1, 2) \\ (2, x_1 \leftarrow x_2 - 1, 0) \\ (3, \texttt{if} \;\; x_2 = 0, 5, 4) \\ (4, x_2 \leftarrow x_2 + 1, 3) \end{array}$$

This program computes the partial function

$$\|P\|_{2,1}(x, y) = \begin{cases} 0 \text{ if } x \geq y, \\ \uparrow \text{otherwise.} \end{cases} \tag{8.6}$$

The corresponding semi-Thue system over the alphabet $\Sigma = \{a, b, 0, 1, 2, 3, 4, 5, L\}$ consists of the following rewriting rules:

$$(L0, L1), (a0, a3), (L1, 2), (a1, a2), (2L, 0), (2b, 0b), (3L, 4L), (3b, 5b), (4, 3L), (L5, 5), (a5L, a5). \quad (8.7)$$

Here is a sample computation in which the registers initially hold the values $x = 3$ and $y = 2$:

SGOTO-2 program	semi-Thue system
$(0, 3, 2)$	$aLLL0LLb$
$(1, 3, 2)$	$aLLL1LLb$
$(2, 2, 2)$	$aLL2LLb$
$(0, 2, 1)$	$aLL0Lb$
$(1, 2, 1)$	$aLL1Lb$
$(2, 1, 1)$	$aL2Lb$
$(0, 1, 0)$	$aL0b$
$(1, 1, 0)$	$aL1b$
$(2, 0, 0)$	$a2b$
$(0, 0, 0)$	$a0b$
$(3, 0, 0)$	$a3b$
$(5, 0, 0)$	$a5b$

$\diamondsuit$

The contruction leads immediately to the following observation.

Lemma 8.3. *A configuration (j, x, y) of the SGOTO-2 program P with $j < n$ leads to the successor configuration (k, u, v) if and only if the semi-Thue system (Σ, R_P) provides the one-step derivation*

$$a\overline{x}j\overline{y}b \rightarrow_{R_P} a\overline{u}k\overline{v}b. \quad (8.8)$$

The iterated application of this statement implies the following.

Proposition 8.4. *A configuration (j, x, y) of the SGOTO-2 program P with $j < n$ leads to the configuration (k, u, v) if and only if the semi-Thue system (Σ, R_P) yields the derivation*

$$a\overline{x}j\overline{y}b \overset{*}{\rightarrow}_{R_P} a\overline{u}k\overline{v}b. \quad (8.9)$$

Moreover, if the SGOTO-2 program terminates, its final configuration is of the form (n, x, y). The corresponding word in the semi-Thue system is $a\overline{x}n\overline{y}b$ which can be further rewritten according to the clean-up rules (8.5) as follows:

$$a\overline{x}n\overline{y}b \overset{*}{\rightarrow}_{R_P} anb. \quad (8.10)$$

This establishes the following result.

Proposition 8.5. *An SGOTO-2 program P started in the configuration $(0, x, y)$ halts if and only if in the corresponding semi-Thue system,*

$$a\overline{x}0\overline{y}b \overset{*}{\rightarrow}_{R_P} anb. \quad (8.11)$$

This proposition yields an effective reduction of the halting problem for SGOTO-2 programs to the word problem for semi-Thue systems. But the halting problem for SGOTO-2 programs is undecidable and thus we have established the following.

Theorem 8.6. *The word problem for semi-Thue systems is undecidable.*

8.2 Thue Systems

Thue systems form a subclass of semi-Thue systems. A *Thue system* is a semi-Thue system (Σ, R) whose relation R is symmetric, i.e., if $u \to v \in R$ then $v \to u \in R$. In a Thue system, the reflexive transitive closure $\overset{*}{\to}_R$ of the one-step rewriting relation $\to_R$ is also symmetric and thus an equivalence relation on Σ^+.

The *word problem* asks for a Thue system (Σ, R) and two strings $s, t \in \Sigma^+$, whether or not the string s can be transformed into the string t by applying the rules from R; that is, $s \overset{*}{\to}_R t$.

This problem is also undecidable. To see this, Thue systems will be related to semi-Thue systems. For this, let (Σ, R) be a semi-Thue system. The *symmetric closure* of the rewriting relation R is the symmetric relation

$$R^{(s)} = R \cup R^{-1}, \tag{8.12}$$

where $R^{-1} = \{(v, u) \mid (u, v) \in R\}$ is the *inverse relation* of R. The set $R^{(s)}$ is the smallest symmetric relation containing R, and the pair $(\Sigma, R^{(s)})$ is a Thue system. The relation $\overset{*}{\to}_{R^{(s)}}$ is thus the reflexive transitive and symmetric closure of $\to_R$ and hence an equivalence relation on Σ^+. That is, for any strings $s, t \in \Sigma^+$, $s \overset{*}{\to}_{R^{(s)}} t$ if and only if $s = t$ or there is a finite sequence $s_0, s_1, \ldots, s_m$ of elements in Σ^+ such that $s = s_0$, $s_i \to_R s_{i+1}$ or $s_{i+1} \to_R s_i$ for $0 \leq i \leq m - 1$, and $s_m = t$. Note that if $s \overset{*}{\to}_{R^{(s)}} t$ holds in a Thue system $(\Sigma, R^{(s)})$, neither $s \overset{*}{\to}_R t$ nor $t \overset{*}{\to}_R s$ need to be valid in the corresponding semi-Thue system (Σ, R).

Example 8.7. Consider the semi-Thue system (Σ, R) with $\Sigma = \{a, b\}$ and $R = \{(ab, b), (ba, a)\}$. The corresponding Thue system is $(\Sigma, R^{(s)})$, where $R^{(s)} = R \cup \{(b, ab), (a, ba)\}$.

In the semi-Thue system, rewriting is strictly antitone leading to smaller strings, i.e., if $u \overset{*}{\to}_R v$ and $u \neq v$, then $|u| > |v|$, e.g., $aabb \to_R abb \to_R bb$.

On the other hand, in the Thue system, $aabb \to_{R^{(s)}} abb \to_{R^{(s)}} abab$, but in the semi-Thue system neither $aabb \overset{*}{\to}_R abab$ nor $abab \overset{*}{\to}_R aabb$. ♢

Theorem 8.8 (Post). *Let P be an SGOTO-2 program with n instructions and exit label n, let (Σ, R_P) be the corresponding semi-Thue system, and let $(\Sigma, R_P^{(s)})$ be the associated Thue system. For each configuration (j, x, y) of the program P,*

$$a\overline{x}j\overline{y}b \overset{*}{\to}_{R_P} anb \quad \Longleftrightarrow \quad a\overline{x}j\overline{y}b \overset{*}{\to}_{R_P^{(s)}} anb. \tag{8.13}$$

Proof. The direction from left-to-right holds since $R \subseteq R^{(s)}$. Conversely, let (j, x, y) be a configuration of the program P. By hypothesis, there is a rewriting sequence in the Thue system such that

$$s_0 = a\overline{x}j\overline{y}b \to_{R_P^{(s)}} s_1 \to_{R_P^{(s)}} \ldots \to_{R_P^{(s)}} s_q = anb, \tag{8.14}$$

where it may be assumed that the length q of the derivation is minimal. It is clear that each occurring string s_i corresponds to a configuration of the program P, $0 \leq i \leq q$.

Suppose the derivation (8.14) cannot be established by the semi-Thue system. That is, the sequence contains a rewriting step $s_p \leftarrow_{R_P} s_{p+1}$, $0 \leq p \leq q - 1$. The index p can be chosen to be maximal with this property. Since there is no rewriting rule applicable to $s_q = anb$, we have $p + 1 < q$. This leads to the following situation:

$$s_p \leftarrow_{R_P} s_{p+1} \rightarrow_{R_P} s_{p+2}. \tag{8.15}$$

However, the string s_{p+1} encodes a configuration of P and there is at most one rewriting rule applicable to it. Thus the words s_p and s_{p+2} must be identical and hence the derivation (8.14) can be shortened by deleting the string s_{p+1} contradicting the assumption. □

The above result is due to the Jewish logician Emil Post (1897-1954) and provides an effective reduction of the derivations in semi-Thue system to derivations in Thue systems. But the word problem for semi-Thue systems is undecidable and thus we obtain the following.

Theorem 8.9. *The word problem for Thue systems is undecidable.*

8.3 Semigroups

Word problems are also encountered in abstract algebra. For this, note that each Thue system gives rise to a semigroup in a natural way. To see this, let $(\Sigma, R^{(s)})$ be a Thue system. We already know that the rewriting relation $\overset{*}{\rightarrow}_{R^{(s)}}$ on Σ^+ is an equivalence relation. The equivalence class of a string $s \in \Sigma^+$ is the subset $[s]$ of all strings in Σ^+ that can be derived from s by a finite number of rewriting steps; i.e.,

$$[s] = \{t \in \Sigma^+ \mid s \overset{*}{\rightarrow}_{R^{(s)}} t\}. \tag{8.16}$$

Note that for all strings $s, t \in \Sigma^+$, we have

$$[s] = [t] \quad \Longleftrightarrow \quad s \overset{*}{\rightarrow}_{R^{(s)}} t. \tag{8.17}$$

Consider the corresponding quotient set given by the set of equivalence classes

$$S = S(\Sigma, R^{(s)}) = \{[s] \mid s \in \Sigma^+\}. \tag{8.18}$$

Proposition 8.10. *The quotient set $S = S(\Sigma, R^{(s)})$ forms a semigroup under the operation*

$$[s] \cdot [t] = [st], \quad s, t \in \Sigma^+. \tag{8.19}$$

Proof. Claim that the operation is well-defined. Indeed, let $[s] = [s']$ and $[t] = [t']$, where $s, s', t, t' \in \Sigma^+$. Then by (8.17), $s \overset{*}{\rightarrow}_{R^{(s)}} s'$ and $t \overset{*}{\rightarrow}_{R^{(s)}} t'$. Thus $st \overset{*}{\rightarrow}_{R^{(s)}} s't \overset{*}{\rightarrow}_{R^{(s)}} s't'$ and hence by (8.17), $[st] = [s't']$.

Moreover, let $r, s, t \in \Sigma^+$. Then by (8.19) and the associativity of the concatenation of words in Σ^+,

$$[r] \cdot ([s] \cdot [t]) = [r] \cdot [st] = [r(st)] = [(rs)t] = [rs] \cdot [t] = ([r] \cdot [s]) \cdot [t].$$

□

The semigroup $S = S(\Sigma, R^{(s)})$ has a presentation in terms of generators and relations

$$S = \langle a_1, \ldots, a_n \mid u_1 = v_1, \ldots, u_m = v_m \rangle,$$

where $\Sigma = \{a_1, \ldots, a_n\}$ is the alphabet and $R = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ is the rule set. The calculation of the elements of the semigroup S will be illustrated by two examples. The appendix provides further details.

Example 8.11. Consider the Thue system $(\Sigma, R^{(s)})$, where $\Sigma = \{a, b\}$ and $R = \{(ab, b), (ba, a)\}$. The semigroup $S = S(\Sigma, R^{(s)})$ has the presentation

$$S = \langle a, b \mid b = ab, a = ba \rangle.$$

For instance, the derivations $a \rightarrow_{R^{(s)}} ba \rightarrow_{R^{(s)}} aba \rightarrow_{R^{(s)}} aa$ and $b \rightarrow_{R^{(s)}} ab \rightarrow_{R^{(s)}} bab \rightarrow_{R^{(s)}} bb$ give rise in the semigroup S to the equations $[a] = [aa]$ and $[b] = [bb]$, respectively.

Any word of S is of the form $[a^{i_1}b^{j_1} \ldots a^{i_k}b^{j_k}]$, where $i_1, \ldots, i_k, j_1, \ldots, j_k \geq 0$ such that at least one of these numbers is non-zero. In view of the relations $[a] = [aa]$ and $[b] = [bb]$, each word of S is an alternating sequence of a's and b's. More specifically,

$$[baba \ldots ba] = [a] = [abab \ldots aba]$$

and

$$[abab \ldots ab] = [b] = [baba \ldots bab].$$

Thus the semigroup S consists only of two elements $[a]$ and $[b]$ which are idempotent, i.e., $[a] \cdot [a] = [a]$ and $[b] \cdot [b] = [b]$, and satisfy $[a] \cdot [b] = [b]$ and $[b] \cdot [a] = [a]$. Therefore, the multiplication table of S is as follows:

$\cdot$	$[a]$	$[b]$
$[a]$	$[a]$	$[b]$
$[b]$	$[a]$	$[b]$

$\diamondsuit$

Example 8.12. Consider the Thue system $(\Sigma, R^{(s)})$ with alphabet $\Sigma = \{a, b\}$ and rule set given by $R = \{(aaa, aa), (bbb, bb), (ab, ba)\}$. The semigroup $S = S(\Sigma, R^{(s)})$ has the presentation

$$S = \langle a, b \mid aa = aaa, bb = bbb, ab = ba \rangle.$$

Any word of S is of the form $[a^{i_1}b^{j_1} \ldots a^{i_k}b^{j_k}]$, where $i_1, \ldots, i_k, j_1, \ldots, j_k \geq 0$ such that at least one of these numbers is non-zero. First, the relation $[ab] = [ba]$ implies that each word of S has the shape $[a^i b^j]$, where $i, j \geq 0$ and $i + j \geq 1$. Second, the relations $[aa] = [aaa]$ and $[bb] = [bbb]$ entail that the semigroup S consists of eight elements: $[a], [b], [aa], [ab], [bb], [aab], [abb], [aabb]$. $\diamondsuit$

The *word problem* for the semigroup $S = S(\Sigma, R^{(s)})$ asks whether or not arbitrary strings $s, t \in \Sigma^+$ describe the same semigroup element $[s] = [t]$. By (8.19), there is an effective reduction of the word problem for Thue systems to the word problem for semigroups. This leads to the following result which was independently established by Emil Post (1987-1954) and Andrey Markov Jr. (1903-1979).

Theorem 8.13. *The word problem for semigroups is undecidable.*

8.4 Post's Correspondence Problem

The Post correspondence problem is an undecidable problem that was introduced by Emil Post in 1946. Due to its simplicity it is often used in proofs of undecidability.

A *Post correspondence system* (PCS) over an alphabet Σ having at least two symbols is a finite set of pairs of elements in Σ^+,

$$\Pi = \{(\alpha_i, \beta_i) \in \Sigma^+ \times \Sigma^+ \mid 1 \leq i \leq l\}. \tag{8.20}$$

For each finite sequence $\boldsymbol{i} = (i_1, \ldots, i_r) \in \{1, \ldots, l\}^+$ of indices, define the (left) string

$$\alpha(\boldsymbol{i}) = \alpha_{i_1} \circ \alpha_{i_2} \circ \cdots \circ \alpha_{i_r} \tag{8.21}$$

and the (right) string

$$\beta(\boldsymbol{i}) = \beta_{i_1} \circ \beta_{i_2} \circ \cdots \circ \beta_{i_r}, \tag{8.22}$$

where $\circ$ denotes the concatenation of strings.

A *solution* of the PCS Π is a sequence $\boldsymbol{i} \in \{1, \ldots, l\}^+$ of indices such that $\alpha(\boldsymbol{i}) = \beta(\boldsymbol{i})$. Note that the alphabet Σ is required to have at least two symbols, since the problem is decidable if the alphabet has only one element. Moreover, if there is a solution of a PCS, the concatenation of the solution is also a solution. Thus if a PCS has one solution, it has infinitely many.

Example 8.14. The PCS $\Pi = \{(\alpha_1, \beta_1) = (aa, a), (\alpha_2, \beta_2) = (ba, ab), (\alpha_3, \beta_3) = (c, ac)\}$ over the alphabet $\Sigma = \{a, b, c\}$ has the solution $\boldsymbol{i} = (1, 2, 3)$, since

$$\begin{aligned} \alpha(\boldsymbol{i}) &= \alpha_1 \circ \alpha_2 \circ \alpha_3 = aa \circ ba \circ c \\ &= aabac \\ &= a \circ ab \circ ac = \beta_1 \circ \beta_2 \circ \beta_3 = \beta(\boldsymbol{i}). \end{aligned}$$

Note that each sequence of the shape $\boldsymbol{i} = (1, 2, \ldots, 2, 3)$ provides a solution. ♢

The *word problem* for Post correspondence systems asks whether a Post correspondence system has a solution or not. This problem is undecidable.

To see this, the word problem for semi-Thue systems will be reduced to this problem. To this end, let (Σ, R) be a semi-Thue system, where $\Sigma = \{a_1, \ldots, a_n\}$ and $R = \{(u_i, v_i) \mid 1 \leq i \leq m\}$. Take a copy of the alphabet Σ given by $\Sigma' = \{a'_1, \ldots, a'_n\}$. In this way, each string $s = a_{i_1} a_{i_2} \ldots a_{i_r}$ over Σ can be assigned a copy $s' = a'_{i_1} a'_{i_2} \ldots a'_{i_r}$ over Σ'.

Put $q = m + n$ and let $s, t \in \Sigma^+$. Define the PCS $\Pi = \Pi(\Sigma, R, s, t)$ over the extended alphabet

$$\overline{\Sigma} = \Sigma \cup \Sigma' \cup \{x, y, z\} \tag{8.23}$$

by the following $l = 2q + 4$ pairs:

$$\begin{array}{llll} (\alpha_i, \beta_i) & = (u_i, v'_i),\ 1 \leq i \leq m, & (\alpha_{m+i}, \beta_{m+i}) & = (a_i, a'_i),\ 1 \leq i \leq n, \\ (\alpha_{q+i}, \beta_{q+i}) & = (u'_i, v_i),\ 1 \leq i \leq m, & (\alpha_{q+m+i}, \beta_{q+m+i}) & = (a'_i, a_i),\ 1 \leq i \leq n, \\ (\alpha_{2q+1}, \beta_{2q+1}) & = (y, z), & (\alpha_{2q+2}, \beta_{2q+2}) & = (z, y), \\ (\alpha_{2q+3}, \beta_{2q+3}) & = (x, xsy), & (\alpha_{2q+4}, \beta_{2q+4}) & = (ztx, x). \end{array} \tag{8.24}$$

Example 8.15. (Cont'd) Take the semi-Thue system (Σ, R) with alphabet $\Sigma = \{a, b, c\}$ and rule set $R = \{(aa, a), (ba, ab), (c, ac)\}$. The corresponding PCS $\Pi = \Pi(\Sigma, R, s, t)$ over the alphabet $\overline{\Sigma} = \{a, b, c, a', b', c', x, y, z\}$ consists of the following pairs:

$$\begin{array}{lll}
(\alpha_1,\beta_1) = (aa,a'), & (\alpha_2,\beta_2) = (ba,a'b'), & (\alpha_3,\beta_3) = (c,a'c'), \\
(\alpha_4,\beta_4) = (a,a'), & (\alpha_5,\beta_5) = (b,b'), & (\alpha_6,\beta_6) = (c,c'), \\
(\alpha_7,\beta_7) = (a'a',a), & (\alpha_8,\beta_8) = (b'a',ab), & (\alpha_9,\beta_9) = (c',ac), \\
(\alpha_{10},\beta_{10}) = (a',a), & (\alpha_{11},\beta_{11}) = (b',b), & (\alpha_{12},\beta_{12}) = (c',c), \\
(\alpha_{13},\beta_{13}) = (y,z), & (\alpha_{14},\beta_{14}) = (z,y), & \\
(\alpha_{15},\beta_{15}) = (x,xsy), & (\alpha_{16},\beta_{16}) = (ztx,x). &
\end{array}$$

$\diamondsuit$

Lemma 8.16. *If $s,t \in \Sigma^+$ with $s = t$ or $s \rightarrow_R t$, there is a sequence $\boldsymbol{i} \in \{1,\ldots,q\}^+$ of indices such that $\alpha(\boldsymbol{i}) = s$ and $\beta(\boldsymbol{i}) = t'$, and there is a sequence $\boldsymbol{i}' \in \{q+1,\ldots,2q\}^+$ of indices such that $\alpha(\boldsymbol{i}') = s'$ and $\beta(\boldsymbol{i}') = t$.*

Proof. If $s = t$, the pairs (a_i, a_i') provide a sequence $\boldsymbol{i}$ with $\alpha(\boldsymbol{i}) = s$ and $\beta(\boldsymbol{i}) = s'$. Similarly, the pairs (a_i', a_i) yield a sequence $\boldsymbol{i}'$ with $\alpha(\boldsymbol{i}') = s'$ and $\beta(\boldsymbol{i}') = s$.

If $s \rightarrow_R t$, then $s = xu_iy$ and $t = xv_iy$ for some $x,y \in \Sigma^*$ and $1 \leq i \leq m$. Write $x = x_1 \ldots x_k$ and $y = y_1 \ldots y_l$ as words over Σ. Then the sequence $\boldsymbol{i}$ given by the sequence of pairs

$$(x_1,x_1'),\ldots,(x_k,x_k'),(u_i,v_i'),(y_1,y_1'),\ldots,(y_l,y_l') \tag{8.25}$$

satisfies $\alpha(\boldsymbol{i}) = s$ and $\beta(\boldsymbol{i}) = t'$. Similarly, the sequence $\boldsymbol{i}'$ defined by the sequence of pairs

$$(x_1',x_1),\ldots,(x_k',x_k),(u_i',v_i),(y_1',y_1),\ldots,(y_l',y_l) \tag{8.26}$$

fulfills $\alpha(\boldsymbol{i}') = s'$ and $\beta(\boldsymbol{i}') = t$. □

Example 8.17. (Cont'd) Let $s = aaac$ and $t = aaaac$. Then the sequences $\boldsymbol{i} = (4,4,4,6)$ and $\boldsymbol{i}' = (10,10,10,12)$ provide

$$\begin{aligned}
\alpha(\boldsymbol{i}) &= \alpha_4 \circ \alpha_4 \circ \alpha_4 \circ \alpha_6 = a \circ a \circ a \circ c = aaac = s, \\
\beta(\boldsymbol{i}) &= \beta_4 \circ \beta_4 \circ \beta_4 \circ \beta_6 = a' \circ a' \circ a' \circ c' = a'a'a'c' = s', \\
\alpha(\boldsymbol{i}') &= \alpha_{10} \circ \alpha_{10} \circ \alpha_{10} \circ \alpha_{12} = a' \circ a' \circ a' \circ c' = a'a'a'c' = s', \\
\beta(\boldsymbol{i}') &= \beta_{10} \circ \beta_{10} \circ \beta_{10} \circ \beta_{12} = a \circ a \circ a \circ c = aaac = s.
\end{aligned}$$

Moreover, $s \rightarrow_R t$ by the rule $(c, ac) \in R$, and the sequences $\boldsymbol{i} = (4,4,4,3)$ and $\boldsymbol{i}' = (10,10,10,9)$ yield

$$\begin{aligned}
\alpha(\boldsymbol{i}) &= \alpha_4 \circ \alpha_4 \circ \alpha_4 \circ \alpha_3 = a \circ a \circ a \circ c = aaac = s, \\
\beta(\boldsymbol{i}) &= \beta_4 \circ \beta_4 \circ \beta_4 \circ \beta_3 = a' \circ a' \circ a' \circ a'c' = a'a'a'a'c' = t', \\
\alpha(\boldsymbol{i}') &= \alpha_{10} \circ \alpha_{10} \circ \alpha_{10} \circ \alpha_9 = a' \circ a' \circ a' \circ c' = a'a'a'c' = s', \\
\beta(\boldsymbol{i}') &= \beta_{10} \circ \beta_{10} \circ \beta_{10} \circ \alpha_9 = a \circ a \circ a \circ ac = aaaac = t.
\end{aligned}$$

$\diamondsuit$

Note that if the PCS $\Pi = \Pi(\Sigma, R, s, t)$ has a solution $\alpha(\boldsymbol{i}) = \beta(\boldsymbol{i})$ with $\boldsymbol{i} = (i_1, \ldots, i_r)$, the solution starts with $i_1 = 2q + 3$ and ends with $i_r = 2q + 4$, i.e., the corresponding string has the prefix xsy and the postfix ztx. The reason is that by construction $(\alpha_{2q+3}, \beta_{2q+3}) = (x, xsy)$ is the only pair whose components have the same prefix and $(\alpha_{2q+4}, \beta_{2q+4}) = (ztx, x)$ is the only pair whose components have the same postfix.

Proposition 8.18. *Let $s, t \in \Sigma^+$. If there is a derivation $s \overset{*}{\rightarrow}_R t$ in the semi-Thue system (Σ, R), the PCS $\Pi = \Pi(\Sigma, R, s, t)$ has a solution.*

Proof. Let $s \overset{*}{\rightarrow}_R t$ with

$$s = s_1 \rightarrow_R s_2 \rightarrow_R \ldots \rightarrow_R s_{k-1} \rightarrow_R s_k = t, \tag{8.27}$$

where $s_i \rightarrow_R s_{i+1}$, $1 \leq i \leq k-1$. We may assume that the number of words k is odd; otherwise, the terminal word may be appended once (without effect), i.e., $s = s_1, s_2, \ldots, s_k = t, s_{k+1} = t$. Then by Lemma 8.16, for each j, $1 \leq j < k$, there is a sequence $\boldsymbol{i}^{(j)}$ of indices such that $\alpha(\boldsymbol{i}^{(j)}) = s_j$ and $\beta(\boldsymbol{i}^{(j)}) = s'_{j+1}$ if j is odd and $\alpha(\boldsymbol{i}^{(j)}) = s'_j$ and $\beta(\boldsymbol{i}^{(j)}) = s_{j+1}$ if j is even.

Claim that a solution of the PCS is given by the sequence

$$\boldsymbol{i} = (2q+3, \boldsymbol{i}^{(1)}, 2q+1, \boldsymbol{i}^{(2)}, 2q+2, \boldsymbol{i}^{(3)}, 2q+1, \ldots, \boldsymbol{i}^{(k-2)}, 2q+1, \boldsymbol{i}^{(k-1)}, 2q+4). \tag{8.28}$$

Indeed, this sequence can be evaluated as follows:

$$\begin{array}{lccccccccccccc}
\alpha(\boldsymbol{i}): & x & s_1 & y & s'_2 & z & s_3 & y & \ldots & s_{k-2} & y & s'_{k-1} & zs_kx \\
 & . & . & . & . & . & . & . & & . & . & . & . \\
\boldsymbol{i}: & 2q+3 & \boldsymbol{i}^{(1)} & 2q+1 & \boldsymbol{i}^{(2)} & 2q+2 & \boldsymbol{i}^{(3)} & 2q+1 & \ldots & \boldsymbol{i}^{(k-2)} & 2q+1 & \boldsymbol{i}^{(k-1)} & 2q+4 \\
 & . & . & . & . & . & . & . & & . & . & . & . \\
\beta(\boldsymbol{i}): & xs_1y & s'_2 & z & s_3 & y & s'_4 & z & \ldots & s'_{k-1} & z & s_k & x
\end{array} \tag{8.29}$$

This proves the claim. □

Example 8.19. (Cont'd) Consider the derivation (with an even number of words)

$$s = aaba \rightarrow_R aaab \rightarrow_R aab \rightarrow_R ab = t$$

given in order by the rules (ba, ab), (aa, a), and (aa, a). An associated solution of the PCS $\Pi = \Pi(\Sigma, R, aaba, ab)$ is defined by the sequence

$$\boldsymbol{i} = (15, 4, 4, 2, 13, 7, 10, 11, 14, 1, 5, 13, 10, 11, 16),$$

where

$$\begin{array}{lccccccccc}
\alpha(\boldsymbol{i}): & x & aaba & y & a'a'a'b' & z & aab & y & a'b' & zabx \\
 & . & . & . & . & . & . & . & . & . \\
\boldsymbol{i}: & 15 & (4,4,2) & 13 & (7,10,11) & 14 & (1,5) & 13 & (10,11) & 16 \\
 & . & . & . & . & . & . & . & . & . \\
\beta(\boldsymbol{i}): & xaabay & a'a'a'b' & z & aab & y & a'b' & z & ab & x
\end{array}$$

◇

Example 8.20. (Cont'd) Consider the derivation (with an odd number of words)

$$s = baaac \rightarrow_R abaac \rightarrow_R aabac \rightarrow_R aaabc \rightarrow_R aaabac = t$$

given in order by the rules (ba, ab), (ba, ab), (ba, ab), and (c, ac). A corresponding solution of the PCS $\Pi = \Pi(\Sigma, R, baaac, aaabac)$ is defined by the sequence

$$\boldsymbol{i} = (15, 4, 4, 2, 6, 13, 7, 10, 11, 12, 14, 1, 5, 6, 13, 10, 11, 9, 16),$$

where

$$\begin{array}{llllllllll}
\alpha(\boldsymbol{i}): & x & baaac & y & a'b'a'a'c' & z & aabac & y & a'a'a'b'c' & zaaabacx \\
 & . & . & . & . & . & . & . & . & . \\
\boldsymbol{i}: & 15 & (2,4,4,6) & 13 & (10,8,10,12) & 14 & (4,4,2,6) & 13 & (10,10,10,11,9) & 16 \\
 & . & . & . & . & . & . & . & . & . \\
\beta(\boldsymbol{i}): & xbaaacy & a'b'a'a'c' & z & aabac & y & a'a'a'b'c' & z & aaabac & x
\end{array}$$

◇

Proposition 8.21. *Let $s, t \in \Sigma^+$. If the PCS $\Pi = \Pi(\Sigma, R, s, t)$ has a solution, there is a derivation $s \overset{*}{\rightarrow}_R t$ in the semi-Thue system (Σ, R).*

Proof. Let $\boldsymbol{i} = (i_1, \ldots, i_r)$ be a solution of the PCS Π. The observation prior to Proposition 8.18 shows that the string $w = \alpha(\boldsymbol{i}) = \beta(\boldsymbol{i})$ must have the form $w = xsy \ldots ztx$ with $s = s_1$. Then by the construction of the PCS Π, the string $\boldsymbol{i}$ must have the form given in (8.29). From this, we conclude that $s = s_1 \rightarrow_R s_2 \rightarrow_R \ldots \rightarrow_R s_k = t$, as required. □

Propositions 8.18 and 8.21 provide an effective reduction of the word problem for semi-Thue systems to the word problem for Post correspondence systems. But the word problem for semi-Thue systems is undecidable. This gives rise to the following result.

Theorem 8.22. *The word problem for Post correspondence systems is undecidable.*

The Post correspondence problem can be used to prove an important undecidability result for context-free languages. The context-free languages form a principal class of formal languages that are particularly useful for the construction of compilers for programming languages.

A *context-free grammar* is a quadruple $G = (\Sigma, V, R, S)$, where Σ is a finite set of *terminals*, V is a finite set of *non-terminals* or *variables*, $S \in V$ is the *start symbol*, and R is a finite subset of $V \times (\Sigma \cup V)^*$ whose elements are called *rewriting rules*. The alphabets Σ and V are assumed to be disjoint. The pair $(\Sigma \cup V, R)$ can be viewed as a semi-Thue system, but here rewriting rules of the form (v, ϵ) with $v \in V$ are allowed.

The *language* of a context-free grammar $G = (\Sigma, V, R, S)$ consists of all terminal strings that can be derived from the start symbol, i.e.,

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\rightarrow}_R w\}. \tag{8.30}$$

The language $L(G)$ generated by a context-free grammar G is called *context-free*.

Example 8.23. The archetypical context-free language

$$L(G) = \{a^n b^n a^k \mid n \geq 1, k \geq 1\} \tag{8.31}$$

is generated by the grammar G, where $\Sigma = \{a, b\}$, $V = \{S, A, B\}$, S is the start symbol, and $R = \{(S, AB), (A, aAb), (A, ab), (B, Ba), (B, a)\}$. For instance, the string $aaabbbaa$ is derived as follows:

$$S \rightarrow_R AB \rightarrow_R ABa \rightarrow_R Aaa \rightarrow_R aAbaa \rightarrow_R aaAbbaa \rightarrow_R aaabbbaa.$$

◇

Example 8.24. An *arithmetic expression* in a programming language is a well-formed formula given by a combination of constants, variables, and operators. A typical context-free grammar describing (simple) arithmetic expressions consists of the following rewriting rules:

$$\begin{aligned}
\text{expr} &\to \text{expr op expr}\\
\text{expr} &\to (\text{expr})\\
\text{expr} &\to -\text{expr}\\
\text{expr} &\to \text{id}\\
\text{op} &\to +\\
\text{op} &\to -\\
\text{op} &\to *\\
\text{op} &\to /\\
\text{op} &\to \uparrow
\end{aligned}$$

The terminals of this grammar are id (identifier), $+$, $-$, $*$, $/$, $\uparrow$, (, and), and the non-terminals are expr and op, where expr is the start symbol. ♢

The problem whether two context-free grammars generate disjoint languages or not is undecidable. To see this, the word problem for Post correspondence systems will be reduced to this problem. For this, let (Σ,Π) be a PCS, where Σ contains (without restriction) no numbers and $\Pi = \{(\alpha_i,\beta_i) \mid 1 \leq i \leq m\}$. Define two grammars G_α and G_β such that both have the same alphabet of terminals $\Sigma \cup \{1,\ldots,m\}$, the same alphabet of non-terminals $V = \{S\}$ and therefore S as common start symbol. Moreover, the rewriting rules of G_α are

$$R_\alpha = \{(S, i\alpha_i \mid 1 \leq i \leq m\} \cup \{(S, iS\alpha_i \mid 1 \leq i \leq m\} \tag{8.32}$$

and the rewriting rules of G_β are

$$R_\beta = \{(S, i\beta_i \mid 1 \leq i \leq m\} \cup \{(S, iS\beta_i \mid 1 \leq i \leq m\}. \tag{8.33}$$

The construction gives immediately the following result.

Proposition 8.25. *The languages of the grammars G_α and G_β are*

$$L(G_\alpha) = \{i_r \ldots i_1 \alpha_{i_1} \ldots \alpha_{i_r} \mid 1 \leq i_1,\ldots,i_r \leq m, r \geq 1\} \tag{8.34}$$

and

$$L(G_\beta) = \{i_r \ldots i_1 \beta_{i_1} \ldots \beta_{i_r} \mid 1 \leq i_1,\ldots,i_r \leq m, r \geq 1\}, \tag{8.35}$$

respectively. Moreover,

$$L(G_\alpha) \cap L(G_\beta) = \{i_r \ldots i_1 \alpha_{i_1} \ldots \alpha_{i_r} \mid \boldsymbol{i} = (i_1,\ldots,i_r) \textit{ solves PCS } (\Sigma,\Pi)\}. \tag{8.36}$$

Example 8.26. (Cont'd) Take the PCS (Σ,Π), where $\Sigma = \{a,b,c\}$ and $\Pi = \{(aa,a),(ba,ab),(c,ac)\}$. The corresponding grammars G_α and G_β are given by the respective sets of rewriting rules

$$R_\alpha = \{(S, 1aa), (S, 2ba), (S, 3c), (S, 1Saa), (S, 2Sba), (S, 3Sc)\} \tag{8.37}$$

and

$$R_\beta = \{(S, 1a), (S, 2ab), (S, 3ac), (S, 1Sa), (S, 2Sab), (S, 3Sac)\}. \tag{8.38}$$

A solution of the PCS is $\boldsymbol{i} = (1, 2, 2, 3)$, since

$$\begin{aligned}\alpha(\boldsymbol{i}) &= \alpha_1 \circ \alpha_2 \circ \alpha_2 \circ \alpha_3 = aa \circ ba \circ ba \circ c \\ &= aababac \\ &= a \circ ab \circ ab \circ ac = \beta_1 \circ \beta_2 \circ \beta_2 \circ \beta_3 = \beta(\boldsymbol{i}).\end{aligned}$$

The corresponding derivations in G_α and G_β are

$$S \to 3Sc \to 32Sbac \to 322Sbabac \to 3221ababac$$

and

$$S \to 3Sac \to 32Sabac \to 322Sababac \to 3221ababac,$$

respectively. Hence, the intersection $L(G_\alpha) \cap L(G_\beta)$ is non-empty. ♢

Proposition 8.25 provides an effective reduction of the word problem for Post correspondence systems to the problem whether the intersection of two context-free languages is empty or not. But the word problem for Post correspondence systems is undecidable and therefore we obtain the following result.

Theorem 8.27. *The problem to decide whether the intersection of two context-free languages is empty or not is undecidable.*

8.5 Diophantine Sets

David Hilbert (1862-1943) presented a list of 23 mathematical problems at the International Mathematical Congress in Paris in 1900. The tenth problem can be stated as follows:

> Given a diophantine equation with a finite number of unknowns and with integral coefficients. Devise a procedure that determines in a finite number of steps whether the equation is solvable in integers.

In 1970, a result in mathematical logic known as Matiyasevich's theorem settled the problem negatively.

Let $\mathbb{Z}[X_1, X_2, \ldots, X_n]$ denote the commutative polynomial ring in the unknowns $X_1, X_2, \ldots, X_n$ with integer coefficients. Each polynomial p in $\mathbb{Z}[X_1, X_2, \ldots, X_n]$ gives rise to a *diophantine equation*

$$p(X_1, X_2, \ldots, X_n) = 0 \tag{8.39}$$

asking for *integer* solutions of this equation. By the fundamental theorem of algebra, every non-constant single-variable polynomial with complex coefficients has at least one complex root, or equivalently, the field of complex numbers is algebraically closed.

Example 8.28. Linear diophantine equations have the form $a_1X_1 + \ldots + a_nX_n = b$. If b is the greatest common divisor of $a_1, \ldots, a_n$ (or a multiple of it), the equation has an infinite number of solutions. This is Bezout's theorem and the solutions can be found by applying the extended Euclidean algorithm. On the other hand, if b is not a multiple of the greatest common divisor of $a_1, \ldots, a_n$, the diophantine equation has no solution. ♢

Let p be a polynomial in $\mathbb{Z}[X_1, \ldots, X_n]$. The *natural variety* of p is the zero set of p in $\mathbb{N}_0^n$,

$$V(p) = \{(x_1, \ldots, x_n) \in \mathbb{N}_0^n \mid p(x_1, \ldots, x_n) = 0\}. \tag{8.40}$$

Each polynomial function is defined by composition of addition and multiplication of integers and so leads to the following result. For this, it is assumed that addition and multiplication of integers are computable functions.

Proposition 8.29. *Each natural variety is a decidable set.*

A *diophantine set* results from a natural variety by existential quantification. More specifically, let p be a polynomial in $\mathbb{Z}[X_1, \ldots, X_n, Y_1, \ldots, Y_m]$. A diophantine set is an n-ary relation

$$\{(x_1, \ldots, x_n) \in \mathbb{N}_0^n \mid p(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0 \text{ for some } y_1, \ldots, y_m \in \mathbb{N}_0\}, \tag{8.41}$$

which will subsequently be denoted by

$$\exists y_1 \ldots \exists y_m [p(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0]. \tag{8.42}$$

Proposition 7.15 yields the following assertion.

Proposition 8.30. *Each diophantine set is semidecidable.*

Example 8.31.

- The set of positive integers is diophantine, since it is given by $\{x \mid \exists y[x = y + 1]\}$.
- The predicates $\leq$ and $<$ are diophantine, since $x \leq y$ if and only if $\exists z[y = x + z]$, and $x < y$ if and only if $\exists z[y = x + z + 1]$.
- The predicate $a \equiv b \bmod c$ is diophantine, since it can be written as $\exists x[(a - b)^2 = c^2x^2]$. ♢

The converse of the above proposition shown by Yuri Matiyasevich (born 1947) in 1970 is also valid, but the proof is not constructive.

Theorem 8.32. *Each semidecidable set is diophantine.*

That is, for each semidecidable set A in $\mathbb{N}_0^n$ there is a polynomial p in $\mathbb{Z}[X_1, \ldots, X_n, Y_1, \ldots, Y_m]$ such that

$$A = \exists y_1 \ldots \exists y_m [p(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0]. \tag{8.43}$$

The negative solution of Hilbert's tenth problem can be proved by using the four-square theorem due to Joseph-Louis Lagrange (1736-1813). For this, an identity due to Leonhard Euler (1707-1783) is needed which is proved by multiplying out and checking:

$$\begin{aligned}&(a^2 + b^2 + c^2 + d^2)(t^2 + u^2 + v^2 + w^2) = \qquad (8.44)\\ &\quad (at + bu + cv + dw)^2 + (au - bt + cw - dv)^2 + (av - ct - bw + du)^2 + (aw - dt + bv - cu)^2.\end{aligned}$$

This equation implies that the set of numbers which are the sum of four squares is closed under multiplication.

Theorem 8.33 (Lagrange). *Each natural number can be written as a sum of four squares.*

For instance, $3 = 1^2 + 1^2 + 1^2 + 0^2$, $14 = 3^2 + 2^2 + 1^2 + 0^2$, and $39 = 5^2 + 3^2 + 2^2 + 1^2$.

Proof. By the above remark it is enough to show that all primes are the sum of four squares. Since $2 = 1^2 + 1^2 + 0^2 + 0^2$, the result is true for 2.

Let p be an odd prime. First, claim that there is some number m with $0 < m < p$ such that mp is a sum of four squares. Indeed, consider the $p + 1$ numbers a^2 and $-1 - b^2$ where $0 \leq a, b \leq (p - 1)/2$. Two of these numbers must have the same remainder when divided by p. However, a^2 and c^2 have the same remainder when divided by p if and only if p divides $a^2 - c^2$; that is, p divides $a + c$ or $a - c$. Thus the numbers a^2 must all have different remainders. Similarly, the numbers $-1 - b^2$ have different remainders. It follows that there must be a and b such that a^2 and $-1 - b^2$ have the same remainder when divided by p; equivalently, $a^2 + b^2 + 1^2 + 0^2$ is divisible by p. But a and b are at most $(p - 1)/2$ and so $a^2 + b^2 + 1^2 + 0^2$ has the form mp, where $0 < m < p$. This proves the claim.

Second, claim that if mp is the sum of four squares with $1 < m < p$, there is a number n with $1 \leq n < m$ such that np is also the sum of four squares. Indeed, let $mp = x_1^2 + x_2^2 + x_3^2 + x_4^2$ and suppose first that m is even. Then either each x_i is even, or they are all odd, or exactly two of them are even. In the last case, it may be assumed that x_1 and x_2 are even. In all three cases each of $x_1 \pm x_2$ and $x_3 \pm x_4$ are even. So $(m/2)p$ can be written as $((x_1 + x_2)/2)^2 + ((x_1 - x_2)/2)^2 + ((x_3 + x_4)/2)^2 + ((x_3 - x_4)/2)^2$, as required.

Next let m be odd. Define numbers y_i by $x_i \equiv y_i \bmod m$ and $|y_i| < m/2$. Then $y_1^2 + y_2^2 + y_3^2 + y_4^2 \equiv x_1^2 + x_2^2 + x_3^2 - x_4^2 \bmod m$ and so $y_1^2 + y_2^2 + y_3^2 + y_4^2 = nm$ for some number $n \geq 0$. The case $n = 0$ is impossible, since this would make every y_i zero and so would make every x_i divisible by m. But then mp would be divisible by m^2, which is impossible since p is a prime and $1 < m < p$.

Clearly, $n < m$ since each y_i is less than $m/2$. Note that $m^2np = (x_1^2 + x_2^2 + x_3^2 - x_4^2)(y_1^2 + y_2^2 + y_3^2 + y_4^2)$. Use Euler's identity to write m^2np as a sum of four squares. Claim that each integer involved in this representation is divisibe by m. Indeed, one of the involved squares is $(x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4)^2$. But the sum $x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$ is congruent mod m to $y_1^2 + y_2^2 + y_3^2 + y_4^2$, since $x_i \equiv y_i \bmod m$. However, $y_1^2 + y_2^2 + y_3^2 + y_4^2 \equiv 0 \bmod m$, as needed. Similarly, the other three integers involved are divisible by m. Now m^2np is the sum of four squares each of which is divisible by m^2. It follows that np itself is the sum of four squares, as required.

Finally, the process to write a multiple mp of p as a sum of four primes iterates leading to smaller multiples np of p. This process will end by reaching p. □

Let p be a polynomial in $\mathbb{Z}[X_1, \ldots, X_n]$. Define the integral polynomial q in the unknowns $T_1, \ldots, T_n$, $U_1, \ldots, U_n$, $V_1 \ldots, V_n$, $W_1, \ldots, W_n$ such that

$$\begin{aligned} q(T_1, T_2, \ldots, T_n, U_1, U_2, \ldots, U_n, V_1, V_2, \ldots, V_n, W_1, W_2, \ldots, W_n) = \\ p(T_1^2 + U_1^2 + V_1^2 + W_1^2, T_2^2 + U_2^2 + V_2^2 + W_2^2, \ldots, T_n^2 + U_n^2 + V_n^2 + W_n^2). \end{aligned} \tag{8.45}$$

Let $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{N}_0^n$ be a solution of the diophantine equation

$$p(X_1, \ldots, X_n) = 0 \tag{8.46}$$

and let $\boldsymbol{t} = (t_1, \ldots, t_n)$, $\boldsymbol{u} = (u_1, \ldots, u_n)$, $\boldsymbol{v} = (v_1, \ldots, v_n)$, $\boldsymbol{w} = (w_1, \ldots, w_n)$ be elements of $\mathbb{Z}^n$ such that by the theorem of Lagrange,

$$x_i = t_i^2 + u_i^2 + v_i^2 + w_i^2, \quad 1 \leq i \leq n. \tag{8.47}$$

Then $(\boldsymbol{t}, \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \in \mathbb{Z}^{4n}$ is a solution of the diophantine equation

$$q(T_1, \ldots, T_n, U_1, \ldots, U_n, V_1, \ldots, V_n, W_1, \ldots, W_n) = 0. \tag{8.48}$$

Conversely, if $(\boldsymbol{t}, \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \in \mathbb{Z}^{4n}$ is a solution of the diophantine equation (8.48), then $\boldsymbol{x} \in \mathbb{N}_0^n$ defined as in (8.47) provides an integer solution of the diophantine equation (8.46).

It follows that if there is an effective procedure to decide whether the diophantine equation (8.48) has *integer* solutions, there is one to decide if the diophantine equation (8.46) has *non-negative integer* solutions.

Theorem 8.34. *Hilbert's tenth problem is undecidable.*

Proof. The set $K = \{x \mid x \in \operatorname{dom} \phi_x\}$ is recursively enumerable and so by Matiyasevich's result there is a polynomial p in $\mathbb{Z}[X, Y_1, \ldots, Y_m]$ such that

$$K = \exists y_1 \ldots \exists y_m [p(x, y_1, \ldots, y_m) = 0]. \tag{8.49}$$

Suppose there is an effective procedure to decide whether or not a diophantine equation has non-negative integer solutions. Then the question whether a number $x \in \mathbb{N}_0$ lies in K or not can be decided by finding a non-negative integer solution of the equation $p(x, Y_1, \ldots, Y_m) = 0$. However, this contradicts the undecidability of K. □

9

Computational Complexity Theory

The previous chapters centered around the problem to decide what is mechanically computable. From the practical point of view, however, another question is what can be computed by an ordinary computer with reasonable resources. This will naturally lead to the central open challenge of computational complexity theory and simultaneously to one of the most important questions in mathematics and science at large, namely if the class of polynomial-time computable problems equals the class of problems whose solutions are polynomial-time verifiable. This chapter gives a brief introduction into this subject from the view point of modern complexity theory. In the following, all functions considered will be recursive and we will be particularly interested in the resource of time needed for the computation.

9.1 Efficient Computations

The Turing machine is still used in complexity theory as a basic model of computation despite its freaky nature. One reason is that the Turing machine is a universal computing model and so by Church's thesis able to simulate all physically realizable computational tasks. Another one is that it can serve as a starting point for other computational models like the multi-tape or nondeterministic Turing machines.

Multi-Tape Turing Machines

A *Turing machine with $k \geq 2$ tapes* is similarly defined as the one-tape one. A Turing machine with k tapes, where $k \geq 2$, has an *input tape* and $k-1$ working tapes. Each tape is equipped with a tape head so that the machine can access one symbol per tape at a time. The machine can only read symbols from the input tape which is thus a read-only tape. The working tapes are read-write tapes onto which symbols can also be written, and the last working tape is designated as *output tape* onto which the result of a computation is written.

The operation of a multi-tape Turing machine is similar to that of the one-tape one. The transition function $\delta : Q \times \Sigma^k \to Q \times \Sigma^{k-1} \times \{L, R, \Lambda\}$ describes the operations performed in each step: If the machine is in state $q \in Q$, $(\sigma_1, \sigma_2, \ldots, \sigma_k)$ are the symbols being read in the k tapes, and

$$\delta(q, (\sigma_1, \sigma_2, \ldots, \sigma_k)) = (q', (\sigma_2', \ldots, \sigma_k'), z),$$

where $z \in \{L, R, \Lambda\}$, then at the next step the symbol σ_i will be replaced by σ_i' on the working tapes, $2 \leq i \leq k$, the machine will enter the state q' and the k tape heads will move according to the value of z.

Example 9.1. A string $x \in \{0,1\}^*$ is a *palindrome* if it reads the same from the left as from the right, like 11011 and 010010.

Claim that there is a Turing machine which on input $x \in \{0,1\}^*$ decides in $3 \cdot |x| + 1$ steps whether the string is a palindrome. Indeed, take a 3-tape Turing machine (input, work, and output) with alphabet $\Sigma = \{b, 0, 1\}$. First, the machine copies the input string x to the work tape in reverse order. This requires $2 \cdot |x|$ steps since the tape head has first to move to the right end of the input word and then copy its symbols from right to left. Second, it checks from left to right if the symbols on the input and work tape are equal. For this, it enters a new state if two symbols are not equal and halts there. Simultaneously reading the strings necessites $|x|$ steps. The machine can output 1 or 0 depending on whether the string is a palindrome or not. ♢

In the sequel, the running time and storage space of a Turing machine will be viewed as a function of the input, and inputs of the same length will be treated in the same way. The key idea to measure time and space as a function of the input was developed by Hartmanis and Stearns in the early 1960's. This was the starting point of the theory of computational complexity.

The set of palindroms over the alphabet $\{0,1\}$ forms a binary language. A *binary language* is a subset of the free monoid $\{0,1\}^*$. Note that each binary language L can be described by a *Boolean function* $f : \{0,1\}^* \to \{0,1\}$ given as the characteristic function of the language L, i.e., for each string $x \in \{0,1\}^*$, $x \in L$ if and only if $f(x) = 1$.

Let $T, S : \mathbb{N}_0 \to \mathbb{N}_0$ be functions. We say that a Turing machine M *decides* a binary language L in time $T(n)$ and uses space $S(n)$ if for each string $x \in \{0,1\}^*$, the machine M halts on input x after $T(|x|)$ steps, visits at most $S(|x|)$ work-tape cells, and outputs 1 if and only if $x \in L$. For instance, the above Turing machine for palindromes runs in $3n + 1$ steps and uses $2n + 1$ tape cells.

A function $T : \mathbb{N}_0 \to \mathbb{N}_0$ is *time constructible* if $T(n) \geq n$ and there is a Turing machine which computes the function $x \mapsto \{T(|x|)\}_2$ in time $T(n)$. Here $\{m\}_2$ denotes the binary representation of the number m. Note that the constraint $T(n) \geq n$ will allow the machine to read the whole input. All time bounds that will subsequently be encountered are time constructible such as n, $n \log n$, n^2, and 2^n. Time bounds that are not time constructible like $\log n$ or $\sqrt{n}$ may lead to abnormal results.

Our definition of Turing machine is robust in such a way that details of the definition do not matter as long as the model is not substantially changed. Two examples of robustness are given whose proofs are left to the reader. First, the size of the alphabet is considered.

Proposition 9.2. *Let L be a binary language and $T : \mathbb{N}_0 \to \mathbb{N}_0$ be a function. If T is time constructible and L is decidable in time $T(n)$ by a Turing machine with alphabet Σ, then L is decidable in time $4n \cdot \log |\Sigma| \cdot T(n)$ by a Turing machine with alphabet $\{b, 0, 1\}$.*

Second, multi-tape Turing machines are related to single-tape ones which have been studied in Chapter 6.

Proposition 9.3. *Let L be a binary language and $T : \mathbb{N}_0 \to \mathbb{N}_0$ be a function. If T is time constructible and L is decidable in time $T(n)$ by a Turing machine using k tapes, then L is decidable in time $5kT(n)^2$ by a single-tape Turing machine.*

It follows that the exact model of computation does not matter as long as the polynomial factor in the running time can be ignored.

Landau Notation

In computational complexity, when running time and working space are considered, constants will not be taken into account; e.g., it will not be distinguished if a machine runs in time $10n^3$ or n^3. For this, the Landau (or big- and little-Oh) notation is used describing the limiting behavior of a function when the argument tends to infinity.

Given two functions f and g from $\mathbb{N}_0$ to $\mathbb{N}_0$. We say that (1) $f = O(g)$ if there there is a positive constant c such that $f(n) \leq c \cdot g(n)$ for all sufficiently large values of n, (2) $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$, and (3) $f = o(g)$ if for each $\epsilon > 0$, $f(n) \leq \epsilon \cdot g(n)$ for every sufficiently large value of n.

Example 9.4. If $f(n) = 10n^2+n+1$ and $g(n) = n^2$, then $f = O(g)$ and $f = \Theta(g)$. If $f(n) = 100n \log n$ and $g(n) = n^2$, then $f = O(g)$ and $f = o(g)$. ♢

By abuse of notation, the Landau symbol can appear in different places of an equation or inequality, such as in $(n + 1)^2 = n^2 + O(n)$ and $n^{O(1)} = O(e^n)$.

The Class P

A *complexity class* is a class of functions which can be computed within a given frame of resources. The first and most important complexity class is P that makes precise the notion of *efficient computation.* To this end, it will be convenient to restrict the following considerations to binary languages which allow to study decision problems.

We say that a Turing machine M *decides* a binary language L in polynomial time if there is a polynomial function $p : \mathbb{N}_0 \to \mathbb{N}$ such that for each string $x \in \{0, 1\}^*$, the machine M halts on input x after $p(|x|)$ steps (by reaching the halting state) and outputs 1 if and only if $x \in L$. Thus the machine M computes the characteristic function of the language L in polynomial time.

Let $T : \mathbb{N}_0 \to \mathbb{N}_0$ be a time constructible function. The class DTIME($T(n)$) consists of all binary languages that can be decided by a Turing machine in time $O(T(n))$. Here the letter D stands for deterministic, since there are other kinds of Turing machines like nonderministic ones which will be considered later on.

The complexity class P captures the notion of efficient computation or polynomial running time and is defined as the union of the classes DTIME(n^c) for any $c \geq 1$; that is,

$$P = \bigcup_{c \geq 1} \mathrm{DTIME}(n^c). \tag{9.1}$$

Example 9.5. The problem of *graph connectivity* is to decide for a given graph G and two vertices u, v in G if u and v are connected in G. This problem lies in P. Indeed, a *depth-first search* can be used to traverse the graph starting with the vertex u. At the beginning, the vertex u is visited and all other vertices are unvisited. In each step, the most recently visited vertex is taken and one of its adjacent vertices which is unvisited is marked visited. If no such vertex exists, then backtracking will eventually lead to the most recently visited vertex which has some unvisited adjacent vertex. The procedure ends after at most $\binom{n}{2}$ steps; this is the number of edges in a fully connected graph with n vertices. Then all vertices are either visited or will never be visited. ♢

Note that the class P refers to decision problems. So we cannot say that ”integer addition” lies in P. But we may consider a decision version that belongs to P, namely, the following language:

$$\{(x, i) \mid \text{the } i\text{th bit of } x + y \text{ is equal to } 1\}.$$

Dealing with decision problems alone can be rather limiting. However, computational problems like computing non-Boolean functions, search problems, optimization problems, interaction, and others can generally be expressed as decision problems.

The class P is intended to capture the notion of efficient computation. One may question if decision problems in DTIME(n^{99}) are efficiently computable in the real world because n^{99} is a huge number even for smaller values of n. However, in practice, we can usually find algorithms for a problem that work in shorter time like n^2 or n^3.

The class P only involves algorithms that compute a function on all possible inputs. This *worst-case scenario* is sometimes criticized since in practice not all possible inputs may arise and the algorithms need only be efficient on inputs that are relevant. A partial answer to this issue is the *average-case scenario* which defines an analog of P for the case of real-life distributions. For instance, the Simplex algorithm developed by George B. Dantzig (1914-2005) in 1947 is a method for solving linear programming problems. The algorithm has an exponential worst-case behaviour but its average-case behaviour is polynomial.

9.2 Efficiently Verifiable Computations

The classes P and NP belong to the most important practical computational complexity classes. Cook's hypothesis asks whether or not the two classes are equal.

The Class NP

The class P captures the notion of efficient computation. In contrast to this, the class NP will capture the notion of efficiently verifiable solution.

We say that a Turing machine M *verifies* a binary language L in polynomial time if there is a polynomial function $p : \mathbb{N}_0 \to \mathbb{N}_0$ such that for each string $x \in \{0,1\}^*$, there is a string $w = w(x) \in \{0,1\}^*$ of length at most $p(|x|)$ such that the machine M halts on input x and w after $p(|x|)$ steps (by reaching the halting state) and outputs 1 if and only if $x \in L$. The above string $w = w(x)$ is a *certificate* or *witness* for x. Thus the machine M verifies in some sense the characteristic function of the language L.

Example 9.6. Here are some interesting decision problems that lie in the class NP:

- *Independent set:* Given a graph G and a number k, decide if G contains an *independent set* (i.e., a subset of vertices with no edges between them) of at least k elements. The corresponding language consists of all pairs (G, k), where G is a graph that contains an independent set of size at least k. The certificate is a set of at least k vertices forming an independent set in G. Note that if the graph has n vertices, a set of k vertices in G can be encoded in $O(k \log n)$ bits. Thus a witness w has at most $O(n \log n)$ bits which is polynomial in the size of the representation of G. Moreover, checking that $m \geq k$ vertices form an independent set in G can be done in polynomial time depending on the representation of the graph. For instance, if the graph is given by an adjacency matrix, then checking all pairs of m vertices requires $\binom{m}{2}$ steps.

- *Clique:* Given a graph G and a number k, decide if G contains a *clique* (i.e., a subset of vertices in which each pair of vertices is connected by an edge) of at least k elements. A certificate is a set of at least k vertices comprising a clique.
- *Graph connectivity:* Given a graph G and two vertices u, v in G, decide if u and v are connected by a path in G. The certificate is a path between u and v.
- *Vertex cover:* Given a graph G and a number k, decide if G has a *vertex cover* (i.e., a subset of vertices such that each edge is incident to at least one vertex in the set) of size at most k. The witness is a set of vertices of size at most k that forms a vertex cover.
- *Graph isomorphism:* Given two $n \times n$ adjacency matrices A_1 and A_2, decide if A_1 and A_2 define the same graph up to some relabelling of the vertices. The certificate is an $n \times n$ permutation matrix P such that $PA_1 = A_2$.
- *Subset sum:* Given a list of n numbers $a_1, \ldots, a_n$ and a number t, decide if there is a subset of numbers that sum up to t. The witness is a list of numbers adding to t.
- *Linear programming:* Given a list of m linear inequalities with rational coefficients over n variables $x_1, \ldots, x_n$, determine if there is an assignment of rational numbers to the variables that satisfies all the inequalities. The certificate is such an assignment.
- *Integer programming:* Given a list of m linear inequalities with rational coefficients over n variables $x_1, \ldots, x_n$, determine if there is an assignment of zeros and ones to the variables which fulfills all the inequalities. The witness is such an assignment.
- *Travelling salesman:* Given a road map with n cities and a number k, decide if there is a closed path that visits each city exactly once and has total length at most k. The certificate is the sequence of cities in such a tour which is also called a *Hamiltonian circuit.*
- *Integer factoring:* Given three numbers N, a, and b, decide if N has a prime factor in the interval $[a, b]$. The witness is the corresponding factor. ♢

In the above list, the problems of *connectivity* and *linear programming* lie in P. For connectivity, this is clear from 9.5, and for linear programming, this is highly nontrivial and follows from the ellipsoid method of Leonid Khachiyan (1952-2005) developed in 1979. The other problems in the list are not known to be in P. The problems *independent set*, *clique*, *vertex cover*, *subset sum*, *integer programming*, and *travelling salesman* are NP-complete which means that they are not in P unless P = NP. The problems of *graph isomorphism* and *integer factoring* are potential candidates for being NP-intermediates, i.e., languages that are neither NP-complete nor in P provided that P ≠ NP.

In order to study the relationship between the classes P and NP, we consider the exponential-time analog of the class P.

$$\mathrm{EXP} = \bigcup_{c \geq 1} \mathrm{DTIME}(2^{n^c}). \tag{9.2}$$

Proposition 9.7. *We have P ⊆ NP ⊆ EXP.*

Proof. Let L be a binary language in P which is decidable in polynomial time by a Turing machine M. By taking for each input the empty certificate $w = \epsilon$, the machine M verifies L in polynomial time and so the language L lies in NP.

Let L be a binary language in NP that can be verified in polynomial time with polynomial function p by a Turing machine M. Then the language L can be decided in time $2^{O(p(n))}$ by enumerating all possible strings w and using M to check if $w = w(x)$ is a witness for the input x. □

The proof shows that each problem in NP can be solved in exponential time by exhaustive search for a certificate. The use of certificates in the definition of the class NP captures a far-reaching episode beyond mathematics, namely that the correctness of an answer is often much easier to perceive than conceiving the answer itself. Examples include verifying a proof versus establishing one, hearing a sonata versus composing one, and appreciating a design such as a car or a building versus making one.

Nondeterministic Turing Machines

The class NP can also be defined in terms of nondeterministic Turing machines which were used in the original definition of NP. This acronym stands for nondeterministic polynomial time.

A *nondeterministic Turing machine* M is similarly defined as the deterministic one in Chapter 6. However, it has a specific accepting state q_a and two transition functions δ_0 and δ_1 such that in each computational step it makes an arbitrary choice what function to apply. This captures the concept of nondeterminism.

We say that a nondeterministic Turing machine M *decides* a binary language L in polynomial time if there is a polynomial function $p : \mathbb{N}_0 \to \mathbb{N}$ such that for each string $x \in \{0,1\}^*$, the machine M halts on input x in $p(|x|)$ steps, and $x \in L$ if and only if there is a sequence of choices of the transition functions such that the machine stops in the accepting state. Note that if $x \notin L$, there is no sequence of choices of the transition functions such that the machine terminates in the accepting state. Instead, the machine will halt in the halting state.

Let $T : \mathbb{N}_0 \to \mathbb{N}_0$ be a time constructible function. The class NTIME($T(n)$) consists of all binary languages that can be decided by a nondeterministic Turing machine in time $O(T(n))$. This definition leads to an alternative characterization of the class NP as the class of all languages calculated by polynomial-time nondeterministic Turing machines. However, note that nondeterministic Turing machines are not intended to model any kind of physically realizable computational process.

Proposition 9.8. *We have* $NP = \bigcup_{c\geq 1} NTIME(n^c)$.

Proof. Let L be a binary language in NP, and let M be a Turing machine that verifies L in polynomial time. For each string $x \in L$, the corresponding witness $w = w(x)$ can be used to provide a sequence of choices of the transition functions such that the associated nondeterministic Turing machine reaches the accepting state. It follows that L lies NTIME(n^c) for some $c \geq 1$.

Conversely, let L be a binary language which is decided by a nondeterministic Turing machine N in polynomial time. For each string $x \in L$, there is a sequence of nondeterministic choices of the transition functions such that the machine N on input x reaches the accepting state. This sequence can serve as a certificate $w = w(x)$ for the string x. Thus the language L belongs to NP. □

9.3 Reducibility and NP-Completeness

Formal languages can be related to each other by the notion of reduction. A binary language L is *reducible* to another binary language L' denoted $L \leq_p L'$, if there is a function $p : \{0,1\}^* \to \{0,1\}^*$ computable in polynomial time such that for each string $x \in \{0,1\}^*$, $x \in L$ if and only if $p(x) \in L'$.

The definitions immediately give the following result.

Proposition 9.9. *If* $L' \in P$ *and* $L \leq_p L'$, *then* $L \in P$.

Proof. A Turing machine M that decides the language L can be defined in such a way that it first transforms an input string x to the string $p(x)$ in polynomial time and then it uses the Turing machine M' which decides the language L' in polynomial time. □

The notion of reducibility allows to define two important subclasses of languages in NP. A binary language L is *NP-hard* if $L' \leq_p L$ for each language L' in NP. Moreover, L is *NP-complete* if L is NP-hard and L lies in NP.

Proposition 9.10. *Let L, L', and L'' be binary languages.*

- *If $L \leq_p L'$ and $L' \leq_p L''$, then $L \leq_p L''$.*
- *If L is NP-hard and $L \in P$, then $P = NP$.*
- *If L is NP-complete, then $L \in P$ if and only if $P = NP$.*

Proof. First, let p be a polynomial-time reduction from L to L' and let p' be a polynomial-time reduction from L' to L''. Then the mapping $x \mapsto p'(p(x))$ provides a polynomial-time reduction from L to L''.

Second, let L be an NP-hard language in P and let L' lie in NP. Then $L' \leq_p L$ and so there is a polynomial-time reduction p such that $x \in L'$ if and only if $p(x) \in L$. But $p(x) \in L$ can be decided in polynomial time and so $x \in L$ also can. Thus L' belongs to P and hence NP is a subclass of P. The converse follows from 9.7.

Third, let L be an NP-complete language. If L lies in P, then the second assertion implies that the classes P and NP are equal. Conversely, if the classes P and NP are equal, then L also lies in P. □

The NP-complete problems are the hardest problems in the class NP in the sense that they can be tackled by a polynomial-time algorithm if and only if the classes P and NP are equal.

Example 9.11. The *independent set* problem can be reduced to the *clique* problem. To see this, let G be a graph with vertex set V and edge set E. The *complementary graph* of G is a graph G', whose vertex set is V and whose edge set is the complement of the edge set E. Then it is clear that a graph G and its complementary graph G' have the property that a vertex subset U of G is independent in G if and only if U is a clique in G'. The reduction can be accomplished by converting the $n \times n$ adjacency matrix of G to the $n \times n$ adjacency matrix of G' in a such a way that all off-diagonal entries are switched from 1 to 0 and vice versa. This can be accomplished in $O(n^2)$ steps. ♢

The consequences of P = NP would be mind-blowing. It would mean that computers could quickly find proofs for mathematical statements for which a verification exists. However, many researchers believe that the classes P and NP are distinct since decades of effort have brought no evidence that efficient algorithms for NP-complete problems exist.

The concept of NP-completeness was independently discovered by Stephen Cook and Leonid Lewin in the early 1970's. Shortly afterwards, Richard Karp demonstrated that many problems of practical interest are NP-complete. Today, several thousands of problems in various fields are known to be NP-complete.

Satisfiability Problem

One of the first studied NP-complete problems came from mathematical logic. In propositional calculus, the basic Boolean operations are Not ($\neg$), And ($\wedge$), and Or ($\vee$) as defined in Table 9.1.

x	y	$\neg x$	$x \wedge y$	$x \vee y$
0	0	1	0	0
0	1		0	1
1	0	0	0	1
1	1		1	1

Table 9.1. Boolean operations: Not, And, and Or.

Using these operations, *Boolean formulas* over variables $x_1, \ldots, x_n$ can be inductively defined:

- Each variable is a Boolean formula.
- If ϕ and ψ are Boolean formulas, then $\neg(\phi)$, $(\phi \wedge \psi)$, and $(\phi \vee \psi)$ are also Boolean formulas.

It is assumed that Not has higher precedence than And and Or. This allows Boolean formulas to be slightly simplified for better readability; e.g., $((\neg x) \vee y)$ can also be written as $(\neg x \vee y)$ or $\neg x \vee y$.

The variables of a Boolean formula can be assigned truth values 0 or 1. A Boolean formula ϕ over the variables $x_1, \ldots, x_n$ is *satisfiable* if there is an assignment $a = (a_1, \ldots, a_n) \in \{0,1\}^n$ such that the substitution $x_i = a_i$ for $1 \leq i \leq n$ and the corresponding expansion $\phi(a)$ of the formula yields $\phi(a) = 1$. For instance, the formula $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$ is satisfiable since the assignment $x_1 = 1$ and $x_2 = 0$ (or $x_1 = 0$ and $x_2 = 1$) gives the value 1.

A Boolean formula over variables $x_1, \ldots, x_n$ is in *conjunctive normal form* (CNF) if it is composed of And's of Or's of variables and their negations. The Or's of variables are called *clauses* and the variables and their negations are called *literals.* For any number $k \geq 1$, a k-CNF is a CNF in which all clauses contain at most k literals. For example, the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2)$ is a 3-CNF. It is well-known that each Boolean function and so each combinatorial circuit, i.e., a digital circuit whose output only depends on the input and not on internal states given by registers or other kind of memory, can be described by a suitable CNF.

Proposition 9.12. *For each Boolean function $f : \{0,1\}^n \to \{0,1\}$, there is an n-CNF ϕ in n variables such that for each assignment $a \in \{0,1\}^n$, $\phi(a) = f(a)$.*

Proof. For each assignment $a \in \{0,1\}^n$, there is a clause $C_a(x_1, \ldots, x_n)$ in n variables such that $C_a(a_1, \ldots, a_n) = 0$ and $C_a(b_1, \ldots, b_n) = 1$ for each assignment $b \in \{0,1\}^n$ different from a. This clause is defined as $y_1 \vee \ldots \vee y_n$, where $y_i = \neg x_i$ if $a_i = 1$, and $y_i = x_i$ if $a_i = 0$, $1 \leq i \leq n$. For instance, the assignment $a = (1,0,1)$ gives the clause $C_a = \neg x_1 \vee x_2 \vee \neg x_3$.

Let ϕ be defined as the And of the clauses C_a for which $f(a)$ takes on the value 0, i.e.,

$$\phi(x_1, \ldots, x_n) = \bigwedge_{\substack{a \\ f(a)=0}} C_a(x_1, \ldots, x_n). \tag{9.3}$$

By definition, ϕ is an n-CNF. Moreover, if $f(b) = 0$, then $C_b(b) = 0$ and so $\phi(b) = 0$. On the other hand, if $f(b) = 1$, then $C_a(b) = 1$ for each a with the property $f(a) = 0$ and so $\phi(b) = 1$. It follows that for each assignment b, $\phi(b) = f(b)$. □

The concept of NP-completeness is based on the following pioneering result. For this, let L_{SAT} denote the language of all satisfiable CNF and in particular let $L_{3\mathrm{SAT}}$ be the language of all satisfiable 3-CNF. Both, L_{SAT} and $L_{3\mathrm{SAT}}$ lie in NP since each satisfying assignment can be used as a witness that the formula is satisfiable.

Theorem 9.13 (Cook-Lewin). *The language L_{SAT} is NP-complete.*

The proof requires to show that *each* language L in NP is reducible to the language L_{SAT}. This necessitates a polynomial-time transformation that converts each string $x \in \{0,1\}^*$ into a CNF ϕ_x in such a way that $x \in L$ if and only if ϕ_x is satisfiable. However, nothing is known about the language L except that it belongs to NP and so the reduction can only use the definition of computation and its expression by a Boolean formula.

Proof. Suppose L is a language in NP. Then by 9.8 there is a nondeterministic Turing machine M that decides L in polynomial time $p(n)$. The computation of the machine M can be described by the following set of Boolean variables:

- For each state $q \in Q$ and number $0 \leq j \leq p(n)$, let $s(q,j)$ be true if M is in state q during the jth step of computation.
- For all numbers $0 \leq i, j \leq p(n)$, let $h(i,j)$ be true if the tape head is at cell i during step j.
- For each tape symbol t and numbers $0 \leq i, j \leq p(n)$, let $c(i,j,t)$ be true if cell i contains the symbol t during the jth step.
- For each number $0 \leq i \leq p(n)$ and each number $0 \leq j \leq p(n) - 1$, let $u(i,j)$ be true if cell i is unchanged from step j to step $j+1$.

These variables can be used to describe the computation of the machine M as follows:

- The machine is in exactly one state during any step of the computation; i.e., for each number $0 \leq j \leq p(n)$,
$$\bigvee_{q \in Q} s(q,j)$$
and for each state $q \in Q$,
$$s(q,j) \Rightarrow \bigwedge_{q' \in Q \setminus \{q\}} \neg s(q',j).$$
- The machine's tape head is in exactly one location during any step of the computation; that is, for each number $0 \leq j \leq p(n)$,
$$\bigvee_{i=0}^{p(n)} h(i,j)$$
and for each number $0 \leq k \leq p(n)$,
$$h(k,j) \Rightarrow \bigwedge_{i \neq k} \neg h(i,j).$$

- Each cell of the machine contains exactly one tape symbol during any step of the computation; i.e., for all numbers $0 \leq i, j \leq p(n)$,
$$\bigvee_t c(i,j,t)$$
and for each tape symbol t,
$$c(i,j,t) \Rightarrow \bigwedge_{t' \neq t} \neg c(i,j,t').$$
- If a cell remains unchanged from one step to the next, the cell's content will remain the same; that is, for all numbers $0 \leq i \leq p(n)$ and $0 \leq j \leq p(n)-1$ and each tape symbol t,
$$c(i,j,t) \wedge u(i,j) \Rightarrow c(i,j+1,t).$$
- The computation on input x begins in the initial state q_0 with tape head located at cell 0, and the first $p(n)+1$ cells contain the word $xb^{p(n)+1-|x|}$; i.e., for all numbers $0 \leq i \leq p(n)$ and $0 \leq j \leq p(n)-1$,
$$s(q_0,0) \wedge h(0,0) \wedge c(0,0,x_1) \wedge \ldots \wedge c(n-1,0,x_n) \wedge c(n,0,b) \wedge \ldots \wedge c(p(n),0,b),$$
where $x = x_1 \ldots x_n$ is the input string and b is the blank symbol.
- Given a configuration, the next configuration is determined by applying one of the transition functions δ_0 or δ_1; that is, for each tape symbol t and each state $q \in Q$,
$$\delta_0(q,t) = (q_0', t_0, d_0) \quad \text{and} \quad \delta_1(q,t) = (q_1', t_1, d_1).$$
Then for all numbers $0 \leq i \leq p(n)$ and $0 \leq j \leq p(n)-1$, for each tape symbol t, and each state $q \in Q$,
$$\begin{aligned}&s(q,j) \wedge h(i,j) \wedge c(i,j,t) \Rightarrow \\ &\bigwedge_{k \neq i} u(k,j) \wedge \bigvee_{m=0,1} [s(q_m', j+1) \wedge h(\eta_m(i), j+1) \wedge c(i,j+1,t_m)],\end{aligned}$$
where for each $m \in \{0,1\}$,
$$\eta_m(i) = \begin{cases} i-1 \text{ if } d_m = L \text{ and } i > 0, \\ i \quad\;\; \text{if } d_m = L \text{ and } i = 0 \text{ or } d_m = R \text{ and } i = p(n) \text{ or } d_m = \Lambda, \\ i+1 \text{ if } d_m = R \text{ and } i < p(n). \end{cases}$$
- The terminal configuration is accepting if possible; i.e., $s(q_a, p(n))$.

Let ϕ_x be the And of the above seven Boolean formulas. For this, note that the implication $a \Rightarrow b$ can be equivalently written as $\neg a \vee b$. For each input string x, $x \in L$ means that there is a computation of the machine M in polynomial time $p(n)$ that reaches the accepting state q_a. This in turn is equivalent to the existence of an assignment of the formula ϕ_x. □

In particular, the 3SAT problem has gained specific attention since it has been the starting point for proving that several other problems are NP-complete.

Corollary 9.14. *The language* L_{3SAT} *is NP-complete.*

Proof. We already know that $L_{3\mathrm{SAT}}$ lies in NP. So it remains to show that each CNF ϕ can be mapped to a 3-CNF ψ such that ϕ is satisfiable if and only if ψ is. The idea is to replace each clause $C = y_1 \vee y_2 \vee y_3 \vee \ldots \vee y_k$ with $k > 3$ literals by an And of the two clauses $C_1 = y_1 \vee y_2 \vee z$ and $C_2 = y_3 \vee \ldots \vee y_k \vee \neg z$, where z is a new variable. If C is true, there is an assignment to z such that both C_1 and C_2 are true. On the other hand, if C is false, then either C_1 or C_2 is false no matter of the assignment of z. The above transformation can be repeatedly applied to convert a CNF ϕ into an equivalent 3-CNF ψ in polynomial-time. □

9.4 Some NP-Complete Languages

After some NP-complete languages like L_{SAT} and $L_{3\mathrm{SAT}}$ have become available, reduction has been used to show that other languages are NP-complete. This approach has quickly led to several thousands of NP-complete languages. Here are two important examples.

The language L_{IntProg} consists of all satisfiable 0/1 integer programs as defined in 9.6; i.e., a finite set of linear inequalities with rational coefficients over variables $x_1, \ldots, x_n$ lies in L_{IntProg} if there is an assignment of values 0 or 1 to the variables $x_1, \ldots, x_n$ which satisfies the inequalities.

Theorem 9.15. *The language L_{IntProg} is NP-complete.*

Proof. The language L_{IntProg} belongs to NP since each assignment can be taken as a certificate. Moreover, the language L_{SAT} can be reduced to L_{IntProg}. To see this, note that each CNF can be converted into an integer program by expressing each clause as an inequality. For instance, the clause $x_1 \vee \neg x_2 \vee x_3$ represents the inequality $x_1 + (1 - x_2) + x_3 \geq 1$. □

The reduction of the satisfiability problem to a graph theoretic problem can be a cumbersome task. The language L_{Clique} consists of all pairs (G, k), where G is a graph and k is a number such that G has a clique with at least k elements.

Theorem 9.16. *The language L_{Clique} is NP-complete.*

Proof. The language L_{Clique} lies in NP since for any pair (G, k) a given set of at least k vertices can be used as a certificate. Furthermore, the language L_{SAT} can be reduced to L_{Clique}. To see this, take a CNF $\phi = \phi_1 \wedge \ldots \wedge \phi_k$ consisting of k clauses, where each clause $\phi_i = l_{i,1} \vee \ldots \vee l_{i,k_i}$ consists of k_i literals $l_{i,j}$, $1 \leq i \leq k$, $1 \leq j \leq k_i$. Consider the graph $G = (V, E)$ with vertex set

$$V = \{[i, j] \mid 1 \leq i \leq k, 1 \leq j \leq k_i\}$$

and edge set

$$E = \{\{[r, s], [u, v]\} \mid r \neq u, l_{r,s} \neq \neg l_{u,v}\}.$$

For instance, the 3-CNF $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$ gives rise to the graph in 9.1.

Suppose that the CNF ϕ is satisfiable. Then there is an assignment a such that $\phi(a) = 1$. Thus $\phi_i(a) = 1$ for each $1 \leq i \leq k$ and so $l_{i,j_i}(a) = 1$ for some literal l_{i,j_i} in ϕ_i, $1 \leq j_i \leq k_i$. Claim that the set $\{[i, j_i] \mid 1 \leq i \leq k\}$ forms a clique. Indeed, if the vertices $[i, j_i]$ and $[u, j_u]$ are not connected for some indices i, u with $i \neq u$, then $l_{i,j_i}(a) = \neg l_{u,j_u}(a)$ contradicting the hypothesis. Thus the given set forms a clique with k elements.

Conversely, assume that the graph G has a clique with k elements. By construction of the graph, the clique must have the form $\{[i, j_i] \mid 1 \leq i \leq k\}$. Define an assignment a with $l_{i,j_i}(a) = 1$, $1 \leq i \leq k$. It is well-defined since for connected vertices $[i, j_i]$ and $[u, j_u]$, $i \neq u$, the literals l_{i,j_i} and l_{u,j_u} are not complementary to each other. This assignment satisfies the clauses ϕ_i, $1 \leq i \leq k$, and hence the CNF ϕ. □

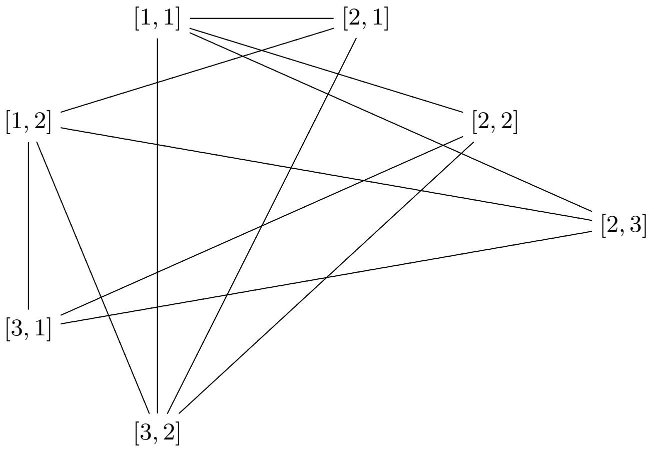


Fig. 9.1. The graph G corresponding to a 3-CNF ϕ.

A

Semigroups and Monoids

In abstract algebra, a semigroup is an algebraic structure with an associative dyadic operation. In particular, a monoid is a semigroup with an identity element. Semigroups and monoids play an important role in the fundamental aspects of computer science. Notably, the full transformation monoid captures the notion of function composition. Each finitely generated free monoid can be represented by the set of strings over a finite alphabet, the syntactic monoid can be used to describe finite state automata, the trace monoid forms a basis for process calculi, and the history monoid furnishes a way to depict concurrent computation. This appendix provides the background on semigroups and monoids required for reading the text.

A *semigroup* is an algebraic structure consisting of a non-empty set S and a dyadic operation

$$\cdot : S \times S \to S : (x, y) \mapsto x \cdot y \tag{A.1}$$

that satisfies the *associative law*; i.e., for all $r, s, t \in S$,

$$(r \cdot s) \cdot t = r \cdot (s \cdot t). \tag{A.2}$$

A semigroup S is *commutative* if it satisfies the *commutative law*; i.e., for all $s, t \in S$,

$$s \cdot t = t \cdot s. \tag{A.3}$$

A semigroup S with operation $\cdot$ is also written as a pair $(S, \cdot)$. The juxtaposition $x \cdot y$ is also denoted as xy. The number of elements of a semigroup S is called the *order* of S.

Example A.1.

- Each singleton set $S = \{s\}$ forms a commutative semigroup under the operation $s \cdot s = s$.
- The set of natural numbers $\mathbb{N}$ forms a commutative semigroup under addition, or multiplication.
- The set of integers $\mathbb{Z}$ forms a commutative semigroup under minimum, or maximum. $\diamondsuit$

A *monoid* is a semigroup M with a distinguished element $e \in M$, called *identity element*, which satisfies for all $m \in M$,

$$m \cdot e = m = e \cdot m. \tag{A.4}$$

A monoid is *commutative* if its operation is commutative. A monoid M with operation $\cdot$ and identity element e is also written as $(M, \cdot, e)$.

Example A.2.

- Any semigroup S can be made into a monoid by adjoining an element e not in S and defining $s \cdot e = s = e \cdot s$ for all $s \in S \cup \{e\}$.
- The set of natural numbers $\mathbb{N}_0$ forms a commutative monoid under addition (with identity element 0), or multiplication (with identity element 1).
- The power set $P(X)$ of a set X forms a commutative monoid under intersection (with identity element X), or union (with identity element $\emptyset$).
- The set of all integral 2×2 matrices

  $$\mathbb{Z}^{2\times 2} = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \mid a, b, c, d \in \mathbb{Z} \right\} \tag{A.5}$$

  forms a commutative monoid under matrix addition

  $$\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} a+a' & b+b' \\ c+c' & d+d' \end{pmatrix} \tag{A.6}$$

  with the zero matrix O as identity element. Likewise, the set of all integral 2×2 matrices $\mathbb{Z}^{2\times 2}$ forms a monoid under matrix multiplication

  $$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix} \tag{A.7}$$

  with the unit matrix I as identity element. This monoid is non-commutative. ♢

In view of the above observation, we will concentrate on monoids instead of semigroups in the sequel.

An *alphabet* denotes always a finite non-empty set, and the elements of an alphabet are called *symbols* or *characters*. A *string* or *word of length* $n \geq 0$ over an alphabet Σ is a sequence $x = (x_1, \ldots, x_n)$ with components in Σ. A sequence $x = (x_1, \ldots, x_n)$ is also written without delimiters as $x = x_1 \ldots x_n$. If $n = 0$, then x is the *empty string* denoted by ϵ.

Proposition A.3. *Let Σ be an alphabet. The set Σ^* of all finite strings over Σ forms a monoid under concatenation of strings and the empty string ϵ is the identity element.*

The monoid Σ^* is called *word monoid* over Σ.

Example A.4. Let $\Sigma = \{1\}$. The unary word monoid Σ^* is commutative and consists of all strings

$$\epsilon,\ 1,\ 11,\ 111,\ 1111,\ 11111, \ldots.$$

Let $\Sigma = \{0, 1\}$. The binary word monoid Σ^* is non-commutative and contains the strings

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0001, \ldots.$$

♢

Proposition A.5. *Let X be a non-empty set. The set of all mappings on X*

$$T_X = \{f \mid f : X \to X\} \tag{A.8}$$

forms a monoid under function composition

$$(f \circ g)(x) = f(g(x)), \quad x \in X, \tag{A.9}$$

and the identity function $\mathrm{id} = \mathrm{id}_X : X \to X : x \mapsto x$ *is the identity element.*

Proof. The function composition is associative and we have for all $f \in T_X$, $f \circ \mathrm{id} = f = \mathrm{id} \circ f$. □

The monoid $(T_X, \circ, \mathrm{id})$ is called the *full transformation monoid* of X.

Example A.6. Let $X = \{1, 2, 3\}$. The mappings

$$f = \begin{pmatrix} 1\,2\,3 \\ 2\,2\,1 \end{pmatrix} \quad \text{and} \quad g = \begin{pmatrix} 1\,2\,3 \\ 2\,1\,3 \end{pmatrix}$$

in T_X satisfy

$$f \circ g = \begin{pmatrix} 1\,2\,3 \\ 2\,2\,1 \end{pmatrix} \quad \text{and} \quad g \circ f = \begin{pmatrix} 1\,2\,3 \\ 1\,1\,2 \end{pmatrix}.$$

◊

Proposition A.7. *Let $(M, \cdot, e)$ and $(M', \cdot', e')$ be monoids. The direct product set $M \times M'$ forms a monoid under the component-wise operation*

$$(x, x') \circ (y, y') = (x \cdot y, x' \cdot' y'), \quad x, y \in M,\, x', y' \in M', \tag{A.10}$$

and the pair (e, e') is the identity element. The monoid $M \times M'$ is commutative if M and M' are commutative.

Proof. Let $x, y, z \in M$ and $x', y', z' \in M'$. In view of associativity,

$$\begin{aligned}(x, x') \circ [(y, y') \circ (z, z')] &= (x, x') \circ (y \cdot z, y' \cdot' z') \\ &= (x \cdot (y \cdot z), x' \cdot' (y' \cdot' z')) \\ &= ((x \cdot y) \cdot z, (x' \cdot' y') \cdot' z') \\ &= (x \cdot y, x' \cdot' y') \circ (z, z') \\ &= [(x, x') \circ (y, y')] \circ (z, z').\end{aligned}$$

In view of commutativity,

$$(x, x') \circ (y, y') = (x \cdot y, x' \cdot' y') = (y \cdot x, y' \cdot' x') = (y, y') \circ (x, x').$$

In view of the identity element,

$$\begin{aligned}(x, x') \circ (e, e') &= (x \cdot e, x' \cdot' e') = (x, x'), \\ (e, e') \circ (x, x') &= (e \cdot x, e' \cdot' x') = (x, x').\end{aligned}$$

□

A *submonoid* of a monoid $(M, \cdot, e)$ is a non-empty subset U of M which contains the identity element e and is closed under the monoid operation; i.e., for all $u, v \in U$, $u \cdot v \in U$.

Example A.8.

- Each monoid $(M, \cdot, e)$ has two trivial submonoids, M and $\{e\}$.
- Let Σ_1 and Σ_2 be alphabets. If Σ_1 is a subset of Σ_2, then Σ_1^* is a submonoid of Σ_2^*.

- The set of all upper triangular integral 2 × 2 matrices

$$U = \left\{ \begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \mid a, b, c \in \mathbb{Z} \right\} \tag{A.11}$$

forms a submonoid of the matrix monoid $(\mathbb{Z}^{2\times 2}, +, O)$, since U is closed under matrix addition

$$\begin{pmatrix} a & b \\ 0 & c \end{pmatrix} + \begin{pmatrix} d & e \\ 0 & f \end{pmatrix} = \begin{pmatrix} a+d & b+e \\ 0 & c+f \end{pmatrix} \tag{A.12}$$

and contains the unit element O. Likewise, the set of all upper triangular integral 2 × 2 matrices U forms a submonoid of the matrix monoid $(\mathbb{Z}^{2\times 2}, \cdot, I)$, since U is closed under matrix multiplication

$$\begin{pmatrix} a & b \\ 0 & c \end{pmatrix} \begin{pmatrix} d & e \\ 0 & f \end{pmatrix} = \begin{pmatrix} ad & ae+bf \\ 0 & cf \end{pmatrix} \tag{A.13}$$

and contains the unit element I. ♢

Proposition A.9. *Let $(M, \cdot, e)$ be a monoid. The mapping*

$$\ell_x : M \to M : y \mapsto xy \tag{A.14}$$

is called left multiplication *with $x \in M$. The set of all left multiplications with elements from M*

$$L_M = \{\ell_x \mid x \in M\} \tag{A.15}$$

forms a submonoid of the full transformation monoid $(T_X, \circ, \mathrm{id})$.

Proof. Let $x, y, z \in M$. Then $\ell_x \circ \ell_y = \ell_{xy}$ since for all $z \in M$,

$$(\ell_{xy})(z) = (xy)z = x(yz) = \ell_x(yz) = \ell_x(\ell_y(z)) = (\ell_x \circ \ell_y)(z),$$

and $\ell_e = \mathrm{id}$ since for all $z \in M$,

$$\ell_e(z) = ez = z = \mathrm{id}(z).$$

□

Each submonoid of a full transformation monoid is called a *transformation monoid.*

Example A.10. Consider the word monoid Σ^* over the latin alphabet Σ. The word `face` induces the left multiplication $\ell_{\texttt{face}} : \Sigma^* \to \Sigma^*$ with $\ell_{\texttt{face}}(\texttt{book}) = \texttt{facebook}$ and $\ell_{\texttt{face}}(\texttt{it}) = \texttt{faceit}$. ♢

Proposition A.11. *Let M be a monoid and X be a subset of M. The set of all finite products of elements of X,*

$$\langle X \rangle = \{x_1 \cdots x_n \mid x_1, \ldots, x_n \in X, n \geq 0\}, \tag{A.16}$$

forms a submonoid of M. It is the smallest submonoid of M containing X.

Proof. The product of two finite products of elements of X is also a finite product, and the empty product $(n = 0)$ gives the identity element of M. Thus $\langle X \rangle$ is a submonoid of M. Moreover, each submonoid U of M containing X must also contain all finite products of elements of X; that is, $\langle X \rangle \subseteq U$. □

For each finite set $X = \{x_1, \ldots, x_n\}$ write $\langle x_1, \ldots, x_n \rangle$ instead of $\langle X \rangle$.

Let M be a monoid and X be a subset of M. Then M is said to be *generated by* X if $M = \langle X \rangle$. In this case, X is a *generating set* of M. In particular, M is *finitely generated* if M has a finite generating set X.

Example A.12.

- For each monoid $(M, \cdot, e)$, we have $\langle \emptyset \rangle = \{e\}$ und $\langle M \rangle = M$.
- The monoid $(\mathbb{N}_0, +, 0)$ is generated by $X = \{1\}$, since $n = n \cdot 1 = 1 + \ldots + 1$ for all $n \in \mathbb{N}_0$.
- The monoid $(\mathbb{N}, \cdot, 1)$ is generated by the set of prime numbers, since by the fundamental theorem of arithmetic, each positive natural number is a (unique) product of prime numbers.
- The word monoid $(\Sigma^*, \circ, \epsilon)$ is generated by the underlying alphabet Σ.
- Consider the matrix monoid $(\mathbb{Z}^{2\times 2}, \cdot, I)$. The powers of the matrix

  $$A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

  are

  $$A^2 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, \quad A^3 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \quad A^4 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \text{and} \quad A^5 = A.$$

  Thus $\langle A \rangle = \{I, A, A^2, A^3\}$ is a (cyclic) submonoid of order 4. ♢

Let $(M, \cdot, e)$ and $(M', \cdot', e')$ be monoids. A *homomorphism* is a mapping $\phi : M \to M'$ which assigns the identity elements

$$\phi(e) = e' \tag{A.17}$$

and commutes with the semigroup operations, i.e., for all $x, y \in M$,

$$\phi(x \cdot y) = \phi(x) \cdot' \phi(y). \tag{A.18}$$

A homomorphism $\phi : M \to M'$ between monoids is a *monomorphism* if ϕ is one-to-one, an *epimorphism* if ϕ is onto, an *isomorphism* if ϕ is one-to-one and onto, an *endomorphism* if $M = M'$, and an *automorphism* if ϕ is an endomorphism and an isomorphism. In particular, two monoids M and M' are *isomorphic* if there is an isomorphism $\phi : M \to M'$. Isomorphic monoids have the same structure up to relabelling of the elements.

Example A.13.

- The power mapping

  $$\phi : \mathbb{N}_0 \to \mathbb{N} : n \mapsto 2^n \tag{A.19}$$

  is a homomorphism from $(\mathbb{N}_0, +, 0)$ into $(\mathbb{N}, \cdot, 1)$, since for all $m, n \in \mathbb{N}_0$,

  $$\phi(m + n) = 2^{m+n} = 2^m \cdot 2^n = \phi(m) \cdot \phi(n)$$

  and $\phi(0) = 2^0 = 1$. This homomorphism is a monomorphism.
- The length mapping

  $$\phi : \Sigma^* \to \mathbb{N}_0 : x \mapsto |x| \tag{A.20}$$

  is a homomorphism from $(\Sigma^*, \cdot, \epsilon)$ onto $(\mathbb{N}_0, +, 0)$, since for all $x, y \in \Sigma^*$,

  $$\phi(xy) = |xy| = |x| + |y| = \phi(x) + \phi(y)$$

  and $\phi(\epsilon) = |\epsilon| = 0$. This homomorphism is an epimorphism.

- The mapping
$$\phi : P(X) \to P(X) : A \mapsto X \setminus A \tag{A.21}$$
is a homomorphism from $(P(X), \cap, X)$ onto $(P(X), \cup, \emptyset)$, since for all $A, B \in P(X)$,
$$\phi(A \cap B) = X \setminus (A \cap B) = (X \setminus A) \cup (X \setminus B) = \phi(A) \cup \phi(B)$$
and $\phi(X) = X \setminus X = \emptyset$. This homomorphism is an isomorphism.
- The mapping
$$\phi : \mathbb{N}_0 \to \mathbb{Z}^{2\times 2} : n \mapsto \begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix} \tag{A.22}$$
is a homomorphism from $(\mathbb{N}_0, +, 0)$ into $(\mathbb{Z}^{2\times 2}, \cdot, I)$, since for all $m, n \in \mathbb{N}_0$,
$$\phi(m+n) = \begin{pmatrix} 1 & m+n \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & m \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix} = \phi(m) + \phi(n)$$
and $\phi(0) = I$. This homomorphism is a monomorphism.
- The mapping
$$\phi : \mathbb{Z}^{2\times 2} \to \mathbb{Z} : A \mapsto \det(A), \tag{A.23}$$
where $\det(A) = ad - bc$ denotes the determinant of the matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, is a homorphisms from $(\mathbb{Z}^{2\times 2}, \cdot, I)$ onto $(\mathbb{Z}, \cdot, 1)$. This homomorphism is an epimorphism.

$\diamondsuit$

Lemma A.14. *Let $\phi : M \to M'$ be a homomorphism between monoids. The set*
$$\operatorname{im}(\phi) = \{\phi(x) \mid x \in M\} \tag{A.24}$$
is a submonoid of M', and the set
$$\ker(\phi) = \{x \in M \mid \phi(x) = e'\} \tag{A.25}$$
is a submonoid of M.

Proof. Let $x, y \in M$. Then $\phi(x) \cdot' \phi(y) = \phi(x \cdot y) \in \operatorname{im}(\phi)$. Moreover, $e' = \phi(e) \in \operatorname{im}(\alpha)$. Therefore, $\operatorname{im}(\phi)$ is a submonoid of M'.

Let $x, y \in \ker(\phi)$. Then $\phi(x \cdot y) = \phi(x) \cdot' \phi(y) = e' \cdot' e' = e'$ and so $x \cdot y \in \ker(\phi)$. Moreover, $\phi(e) = e'$ and thus $e \in \ker(\phi)$. It follows that $\ker(\phi)$ is a submonoid of M. □

The monoid $\operatorname{im}(\phi)$ is called the *image* of ϕ, and the monoid $\ker(\phi)$ is called the *kernel* of ϕ. Note that $\ker(\phi)$ can be the trivial monoid $\{e\}$ even if the homomorphism ϕ is not a monomorphism (e.g., length function in Example A.13).

Proposition A.15. *Each monoid M is isomorphic to the monoid of left multiplications L_M.*

Proof. Claim that the mapping

$$\phi : M \to L_M : x \mapsto \ell_x$$

is an isomorphism. Indeed, for all $x, y \in M$,

$$\phi(xy) = \ell_{xy} = \ell_x \circ \ell_y = \phi(x)\phi(y)$$

and

$$\phi(e) = \ell_e = \mathrm{id}.$$

It is clear that the mapping is onto. It is also one-to-one, since $\ell_x = \ell_y$ with $x, y \in M$ implies by substitution of the identity element e:

$$x = ex = \ell_x(e) = \ell_y(e) = ey = y.$$

□

Example A.16. Consider the monoid $M = \{a, e, b\}$ (corresponding to the multiplicative monoid $\{-1, 0, 1\}$) with the multiplication table

$\cdot$	a	e	b
a	b	e	a
e	e	e	e
b	a	e	b

The transformation monoid of left multiplications L_M consists of the elements

$$\ell_a = \begin{pmatrix} a & e & b \\ b & e & a \end{pmatrix}, \ \ell_e = \begin{pmatrix} a & e & b \\ e & e & e \end{pmatrix}, \ \ell_b = \begin{pmatrix} a & e & b \\ a & e & b \end{pmatrix},$$

where the left multiplications correspond one-to-one with the rows of the multiplication table of M. The multiplication table of L_M is as follows:

$\cdot$	ℓ_a	ℓ_e	ℓ_b
ℓ_a	ℓ_b	ℓ_e	ℓ_a
ℓ_e	ℓ_e	ℓ_e	ℓ_e
ℓ_b	ℓ_a	ℓ_e	ℓ_b

♢

By the Propositions A.9 and A.15, we obtain the following result.

Theorem A.17. *Each monoid M is isomorphic to a transformation monoid.*

Thus the theory of monoids can be considered as the theory of transformations.

Lemma A.18. *If $\phi : M \to M'$ und $\psi : M' \to M''$ are homomorphisms between monoids, the composition $\psi\phi : M \to M''$ is also a homomorphism.*

Proof. The composition of mappings is a mapping. Moreover, for all $x, y \in M$,

$$(\psi\phi)(x \cdot y) = \psi(\phi(x \cdot y)) = \psi(\phi(x) \cdot' \phi(y)) = \psi(\phi(x)) \cdot'' \psi(\phi(y)) = (\psi\phi)(x) \cdot'' (\psi\phi)(y)$$

and $(\psi\phi)(e) = \psi(\phi(e)) = \psi(e') = e''$. □

Proposition A.19. *The set of all endomorphisms* End(M) *of a monoid M forms a monoid under the composition of mappings.*

Proof. By Lemma A.18, the composition of endomorphisms $\phi : M \to M$ and $\psi : M \to M$ is an endomorphism $\psi\phi : M \to M$. Thus the set of all endomorphisms of M is closed under composition. Moreover, the composition of mappings is associative and the identity mapping $\mathrm{id} : M \to M$ is the identity element. □

Therefore, the endomorphism monoid $\mathrm{End}(M)$ is a submonoid of the full transformation monoid T_M.

Proposition A.20. *Let M and M' be monoids. If $\phi : M \to M'$ is an isomorphism, the inverse mapping $\phi^{-1} : M' \to M$ is also an isomorphism.*

Proof. The inverse mapping ϕ^{-1} exists, since ϕ is bijective. Moreover, the mapping ϕ^{-1} is also bijective and we have $\phi\phi^{-1} = \mathrm{id} = \phi^{-1}\phi$. Since ϕ is a homomorphism, we have for all $x, y \in M'$,

$$\phi(\phi^{-1}(x) \cdot \phi^{-1}(y)) = \phi(\phi^{-1}(x)) \cdot' \phi(\phi^{-1}(y)) = x \cdot' y$$

and therefore $\phi^{-1}(x) \cdot \phi^{-1}(y) = \phi^{-1}(x \cdot' y)$. Moreover, $\phi(e) = e'$ implies $\phi^{-1}(e') = e$. □

Example A.21. The exponential function

$$\exp : (\mathbb{R}, +, 0) \to (\mathbb{R}_{>0}, \cdot, 1) : x \mapsto \exp(x) = e^x \tag{A.26}$$

is an isomorphism, since this mapping is bijective,

$$\exp(x + y) = e^{x+y} = e^x \cdot e^y = \exp(x) \cdot \exp(y), \quad x, y \in \mathbb{R},$$

and $\exp(0) = e^0 = 1$. The inverse isomorphism is the logarithm function

$$\log : (\mathbb{R}_{>0}, \cdot, 1) \to (\mathbb{R}, +, 0) : x \mapsto \log(x) \tag{A.27}$$

This mapping is bijective,

$$\log(x \cdot y) = \log(x) + \log(y), \quad x, y \in \mathbb{R}_{>0},$$

and $\log(1) = 0$. ♢

A monoid M is called *free* if M has a generating set X such that each element of M can be uniquely written as a product of elements of X. In this case, the set X is called a *basis* of M.

Proposition A.22. *The word monoid Σ^* over Σ is free with basis Σ.*

Proof. By definition, $\Sigma^* = \langle \Sigma \rangle$. Suppose the word $x \in \Sigma^*$ has two representations

$$x_1 x_2 \ldots x_m = x = y_1 y_2 \ldots y_n, \quad x_i, y_j \in \Sigma.$$

Each word is a sequence or mapping

$$\begin{pmatrix} 1 & 2 & \ldots & m \\ x_1 & x_2 & \ldots & x_m \end{pmatrix} = x = \begin{pmatrix} 1 & 2 & \ldots & n \\ y_1 & y_2 & \ldots & y_n \end{pmatrix}.$$

Thus by the equality of mappings $m = n$ and $x_i = y_i$ for $1 \leq i \leq n$. □

Theorem A.23 (Universal Property). *Let M be a free monoid with basis X. For each monoid M' and each mapping $\phi_0 : X \to M'$ there is a unique homomorphism $\phi : M \to M'$ extending ϕ_0.*

The universal property provides the following commutative diagram:

$$\begin{tikzcd} X \arrow[r, "\phi_0"] \arrow[d, "\mathrm{id}"] & M' \\ M \arrow[ur, dashed, "\phi"'] & \end{tikzcd}$$

Proof. Since M is a free monoid with basis X, each element $x \in M$ has a unique representation $x = x_1 x_2 \ldots x_n$, where $x_i \in X$. Thus the mapping $\phi_0 : X \to M'$ can be extended to a mapping $\phi : M \to M'$ such that

$$\phi(x_1 x_2 \ldots x_n) = \phi_0(x_1)\phi_0(x_2)\ldots\phi_0(x_n), \quad x_i \in X.$$

The uniqueness property ensures that ϕ is a mapping. This mapping is a homomorphism, since for all $x = x_1 \ldots x_m \in \Sigma^*$ and $y = y_1 \ldots y_n \in \Sigma^*$ with $x_i, y_j \in \Sigma$, we have

$$\phi(xy) = \phi(x_1 \ldots x_m y_1 \ldots y_n) = \phi_0(x_1)\ldots\phi_0(x_m)\phi_0(y_1)\ldots\phi_0(y_n) = \phi(x_1 \ldots x_m)\phi(y_1 \ldots y_n) = \phi(x)\phi(y).$$

Let $\psi : M \to M'$ be another homomorphism extending ϕ_0. Then for each $x = x_1 \ldots x_n \in \Sigma^*$ with $x_i \in \Sigma$,

$$\phi(x_1 x_2 \ldots x_n) = \phi_0(x_1)\phi_0(x_2)\ldots\phi_0(x_n) = \psi(x_1)\psi(x_2)\ldots\psi(x_n) = \psi(x_1 x_2 \ldots x_n).$$

Therefore, $\phi = \psi$. □

Example A.24.

- The trivial monoid $M = \{e\}$ is free with basis $X = \emptyset$.
- The monoid $(\mathbb{N}_0, +, 0)$ is free with basis $X = \{1\}$, since each natural number n can be uniquely written as $n = n \cdot 1 = 1 + \ldots + 1$.
- The monoid $(\mathbb{N}, \cdot, 1)$ is not free. Suppose it would be free with basis X. Take the word monoid Σ^* with basis $\Sigma = X$ und the mapping $\phi_0 : X \to \Sigma^* : n \mapsto n$. By the universal property, there is a unique homomorphism $\phi : \mathbb{N} \to \Sigma^*$ extending ϕ_0. The basis X contains at least two elements, since $\mathbb{N}$ is more than the power of one number. If say $2, 3 \in X$, then

  $$23 = \phi_0(2)\phi_0(3) = \phi(2 \cdot 3) = \phi(6) = \phi(3 \cdot 2) = \phi_0(3)\phi_0(2) = 32.$$

  But 23 and 32 are different as strings. A contradiction.
- Each finite non-trivial monoid M cannot be free. Suppose M would be free with basis X. Take the word monoid Σ^* with basis $\Sigma = X$ and the mapping $\phi_0 : X \to \Sigma^* : x \mapsto x$. By the universal property, there is a unique homomorphism $\phi : M \to \Sigma^*$ extending ϕ_0. Since M is non-trivial, there is an element $x \neq e$ in M. Consider the powers $x, x^2, x^3, \ldots$. Since M is finite, there are numbers i, j such that $i < j$ and $x^i = x^j$. Then $x^i = \phi(x^i) = \phi(x^j) = x^j$. But x^i and x^j are distinct as strings in Σ^*. A contradiction. ♢

Proposition A.25. *Let M and M' be free monoids with finite bases X and X', respectively. If $|X| = |X'|$, then M and M' are isomorphic.*

Proof. Since X and X' have the same finite cardinality, we can choose bijections $\phi_0 : X \to X'$ and $\psi_0 : X' \to X$ which are inverse to each other. By the universal property, there are homomorphisms $\phi : M \to M'$ and $\psi : M' \to M$ extending ϕ_0 and ψ_0, respectively. Then for each $x = x_1 \ldots x_m \in M$ with $x_i \in X$,

$$(\psi\phi)(x) = (\psi\phi)(x_1 \ldots x_m) = \psi(\phi_0(x_1)\ldots\phi_0(x_m)) = \psi_0(\phi_0(x_1))\ldots\psi_0(\phi_0(x_m)) = x_1 \ldots x_m = x.$$

Thus $\psi\phi = \mathrm{id}_M$ and similarly $\phi\psi = \mathrm{id}_{M'}$. Therefore, ϕ and ψ are isomorphisms, as required. □

By the Propositions A.22 and A.25, and Example A.24, we obtain the following result.

Theorem A.26. *For each number $n \geq 0$, there is exactly one free monoid (up to isomorphism) with a basis of n elements.*

Take the word monoid Σ^* over an alphabet Σ and a finite set of pairs $R = \{(u_i, v_i) \mid 1 \leq i \leq m\}$ in $\Sigma^* \times \Sigma^*$. Two strings $x, y \in \Sigma^*$ are *equivalent*, written $x \sim y$, if $x = y$ or there is a finite sequence $x = s_0, s_1, \ldots, s_k = y$ of strings in Σ^* such that for each $0 \leq i \leq k-1$, $s_i = a_i u_j b_i$ and $s_{i+1} = a_i v_j b_i$, or $s_i = a_i v_j b_i$ and $s_{i+1} = a_i u_j b_i$, where $a_i, b_i \in \Sigma^*$ and $1 \leq j \leq m$. This relation is an equivalence relation on Σ^*. The equivalence class of $x \in \Sigma^*$ is

$$[x] = \{y \in \Sigma^* \mid x \sim y\} \tag{A.28}$$

and the quotient set consisting of all equivalence classes is

$$M = \{[x] \mid x \in \Sigma^*\}. \tag{A.29}$$

Proposition A.27. *The quotient set M forms a monoid under the operation*

$$[x] \cdot [y] = [xy], \quad x, y \in \Sigma^*. \tag{A.30}$$

Proof. Claim that the operation is well-defined. Indeed, let $[x] = [y]$ and $[x'] = [y']$, where $x, x', y, y' \in \Sigma^*$. Let $x = s_0, s_1, \ldots, s_k = y$ and $x' = s'_0, s'_1, \ldots, s'_l = y'$ be the corresponding sequences of strings. Then $xx' \sim yy'$ by the sequence of strings $xx' = s_0 x', s_1 x', \ldots, s_k x' = yx' = ys'_0, ys'_1, \ldots, ys'_l = yy'$. Hence, $[xy] = [x'y']$ as claimed. Moreover, let $x, y, z \in \Sigma^*$. Then by (A.30) and the associativity of the concatenation of strings in Σ^*,

$$[x] \cdot ([y] \cdot [z]) = [x] \cdot [yz] = [x(yz)] = [(xy)z] = [xy] \cdot [z] = ([x] \cdot [y]) \cdot [z].$$

Finally, the equivalence class of the empty word is the identity element of M, since for all $x \in \Sigma^*$, $[x] \cdot [\epsilon] = [x\epsilon] = [x] = [\epsilon x] = [\epsilon] \cdot [x]$. □

By (A.30), the above equivalence relation is compatible with the monoid operation and is therefore called a *congruence relation.*

Let $\Sigma = \{a_1, \ldots, a_n\}$. Then the above monoid M is said to have the *presentation*

$$M = \langle a_1, \ldots, a_n \mid u_1 = v_1, \ldots, u_m = v_m \rangle$$

given by the *generators* $a_1, \ldots, a_n$ and the *relations* $u_1 = v_1, \ldots, u_m = v_m$.

Example A.28. Let $\Sigma = \{a, b\}$ and $R = \{(aaa, a), (bbb, b), (ab, ba)\}$. The corresponding monoid M has the presentation

$$M = \langle a, b \mid aaa = a, bbb = b, ab = ba \rangle.$$

Note that each element of Σ^* has the form $a^{i_1} b^{j_1} \ldots a^{i_l} b^{j_l}$, where $i_1, \ldots, i_l, j_1, \ldots, j_l \geq 0$ and $l \geq 0$. By the relation (ab, ba), each element $a^{i_1} b^{j_1} \ldots a^{i_l} b^{j_l}$ is equivalent to $a^i b^j$, where $i = i_1 + \ldots + i_l$ and $j = j_1 + \ldots + j_l$. Moreover, by the relations (aaa, a) and (bbb, b), each element $a^i b^j$ with $i, j \geq 0$ is equivalent to one the words $\epsilon, a, b, aa, ab, bb, aab, abb, aabb$. Thus the monoid M consists of nine elements: $[\epsilon]$, $[a]$, $[b]$, $[aa]$, $[ab]$, $[bb]$, $[aab]$, $[abb]$, and $[aabb]$. The multiplication table of the monoid M is given in Table A.1. ♢

Further examples of monoid presentations can be found in Section 8.3.

$\cdot$	ϵ	a	b	aa	ab	bb	aab	abb	$aabb$
ϵ	ϵ	a	b	aa	ab	bb	aab	abb	$aabb$
a	a	aa	ab	a	aab	abb	ab	$aabb$	abb
b	b	ab	bb	aab	abb	b	$aabb$	ab	aab
aa	aa	a	aab	aa	ab	$aabb$	aab	abb	$aabb$
ab	ab	aab	abb	ab	$aabb$	ab	abb	aab	ab
bb	bb	abb	b	$aabb$	ab	bb	aab	abb	$aabb$
aab	aab	ab	$aabb$	aab	abb	aab	$aabb$	ab	aab
abb	abb	$aabb$	ab	abb	aab	abb	ab	$aabb$	abb
$aabb$	$aabb$	abb	aab	$aabb$	ab	$aabb$	aab	abb	$aabb$

Table A.1. Multiplication table of monoid M. The elements of M are written without brackets.

Index