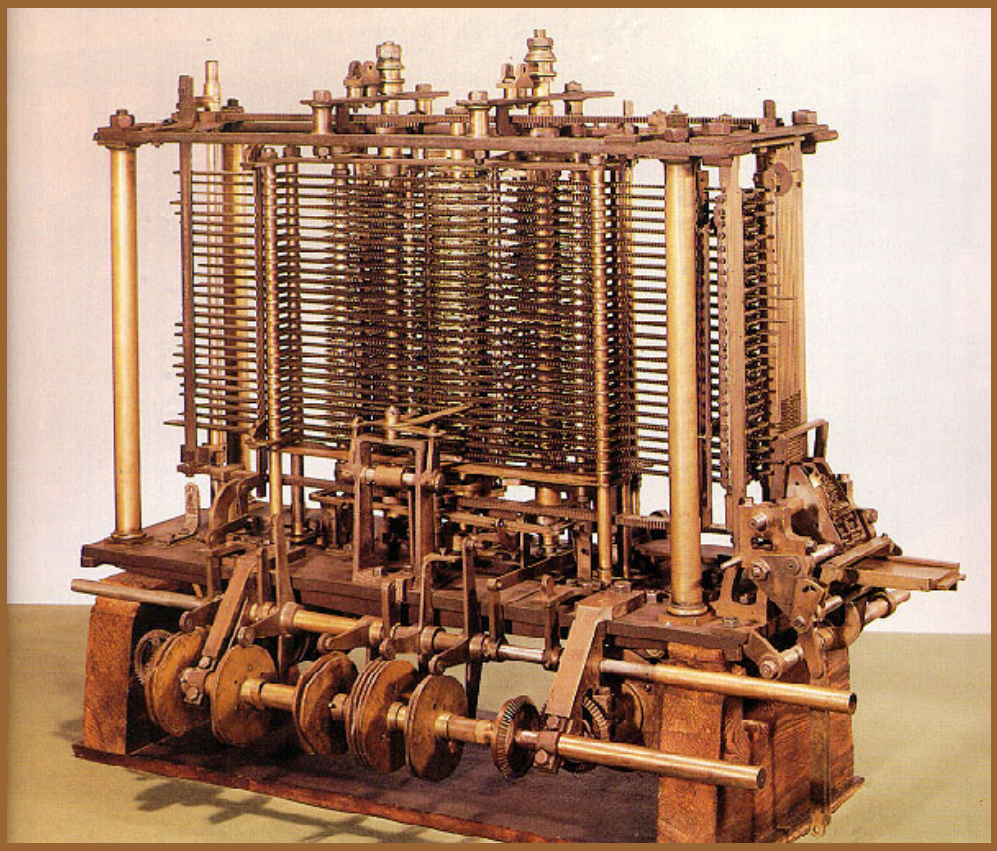


Computability Theory

Karl-Heinz Zimmermann

Computability Theory

Karl-Heinz Zimmermann

Computability Theory

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-tubdok-11072

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 solvable by an algorithm, a rigorous formalism is necessary that allows mathematical proofs. The need for such a formalism became apparent in the works of David Hilbert (1900) on the foundations of mathematics and Kurt Gödel (1931) on the incompleteness of elementary arithmetic.

The first investigations in the field were conducted by the logicians Alonzo Church, Stephen Kleene, Emil Post, and Alan Turing in the early 1930s. They 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 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 logicians and computer scientists. Many mathematical problems are known to be undecidable such as the word problem for groups, the halting problem, and Hilbert's tenth problem.

This book is a development of class notes for a two-hour lecture including a one-hour lab held for Bachelor students of Computer Science at the Hamburg University of Technology in the summer term 2011. The aim of the course was to present the basic results of computability theory, including mathematical models of computability (Turing machine, unlimited register machine, and LOOP and GOTO programs), primitive recursive and partial recursive functions, Ackermann's function, Gödel numbering, universal functions, smn theorem, Kleene's normal form, undecidable sets, theorems of Rice, and word problems. The manuscript partly follows the notes taken by the author during his studies at the University of Erlangen-Nuremberg. I would like thank again my teachers Martin Becker[†] and Volker Strehl for giving inspiring lectures in this field.

First of all, I would like to express my thanks to Ralf Möller for valuable comments. I am also grateful to Mahwish Saleemi for conducting the lab and to Wolfgang Brandt for valuable technical support. Finally, I thank my students for their attention, their stimulating questions, and their dedicated work.

Hamburg, Juli 2011 *Karl-Heinz Zimmermann*

Contents

1

Register Machine

The register machine is an abstract computing machine that allows to make precise the notion of computability. The unlimited register machine (URM) introduced by Sheperdson and Sturgis (1963) 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

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

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 denumerable.*

Proof. Let $(p_0, p_1, p_2, \ldots)$ denote the sequence of prime numbers. Due to the unique factorization of integers into primes, the mapping

$$\Omega \to \mathbb{N}_0 : \omega \mapsto \prod_i p_i^{\omega_i}$$

is a bijection. □

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

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

and a *range*

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

Two partial functions $f, g \in E(\Omega)$ are *equal*, written $f = g$, iff 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.

For instance, the *increment function* $a_k \in E(\Omega)$ with respect to the kth register is given as $a_k : \omega \mapsto \omega'$ where

$$\omega'_n = \begin{cases} \omega_n & \text{if } n \neq k, \\ \omega_k + 1 & \text{otherwise,} \end{cases} \tag{1.6}$$

and the *decrement function* $s_k \in E(\Omega)$ with respect to the kth register is defined as $s_k : \omega \mapsto \omega'$ where

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

The binary operation $\dot{-}$ on $\mathbb{N}_0$ denotes the *asymmetric difference*

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y, \\ 0 & \text{otherwise.} \end{cases} \tag{1.8}$$

Both functions a_k and s_k are total.

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

$$R_f = \{(\omega, f(\omega)) \mid \omega \in \mathrm{dom}(f)\}. \tag{1.9}$$

It follows that two partial functions $f, g \in E(\Omega)$ are equal iff 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 by

$$(g \circ f)(\omega) = g(f(\omega), \tag{1.10}$$

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

$$\mathrm{dom}(g \circ f) = \{\omega \in \Omega \mid \omega \in \mathrm{dom}(f) \wedge f(\omega) \in \mathrm{dom}(g)\}. \tag{1.11}$$

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

Proposition 1.2. *The set $E(\Omega)$ together with the binary operation of composition is a semigroup.*

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

$$f^0 = \mathrm{id}_\Omega, \quad \text{and} \quad f^{n+1} = f \circ f^n,\ n \in \mathbb{N}_0. \tag{1.12}$$

In particular, $f^1 = f \circ \mathrm{id}_\Omega = f$.

Consider for each $f \in E(\Omega)$ and $\omega \in \Omega$ the sequence

$$\omega = f^0(\omega), f^1(\omega), f^2(\omega), \ldots \tag{1.13}$$

This sequence is finite if $\omega \notin \mathrm{dom}(f^j)$. For this, put

$$\lambda(f,\omega) = \begin{cases} \min\{j \in \mathbb{N}_0 \mid \omega \notin \mathrm{dom}(f^j)\} & \text{if } \{\ldots\} \neq \emptyset, \\ \infty & \text{otherwise.} \end{cases} \tag{1.14}$$

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

$$f^{*n}(\omega) = f^k(\omega), \tag{1.15}$$

if there is an integer $k \geq 0$ with $k < \lambda(f,\omega)$ such that the content of the nth register is zero 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 ($\uparrow$). The computation of f^{*n} can be carried out by the `while` loop 1.1.

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

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

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

$$Z = (\Sigma_{10} \setminus \{0\})\Sigma_{10}^+ \cup \Sigma_{10}. \tag{1.16}$$

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

The URM programs are words over the alphabet

$$\Sigma_{\mathrm{URM}} = \{A, S, (,), ;\} \cup Z. \tag{1.17}$$

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 $\sigma \in Z$, $(P)\sigma \in \mathcal{P}_{\mathrm{URM}}$,

4. if $P, Q \in \mathcal{P}_{\mathrm{URM}}$, $P;Q \in \mathcal{P}_{\mathrm{URM}}$.

The programs $A\sigma$ and $S\sigma$ are called *atomic*, the program $(P)\sigma$ is denoted as the *iteration* of the program P with respect to the register R_σ, and the program $P;Q$ is called 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

$$P^n = P;P;\ldots;P \quad (n \text{ times}). \tag{1.18}$$

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

Lemma 1.3. *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 since it only increases readability. In this way, we obtain the following

Proposition 1.4. *The set $\mathcal{P}_{\mathrm{URM}}$ together with the operation of concatenation is a subsemigroup of Σ^+_{URM} which is freely generated by the set of blocks $\mathcal{B}$.*

Example 1.5. 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 $P_2 = ((A1;S3)3;S2;(A0;A3;S4)4;(A4;S0)0)2$. ⧫

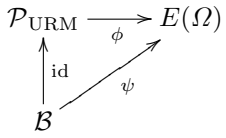
1.3 Semantics of URM Programs

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 $\sigma \in Z$, $|(P)\sigma| = |P|^{*\sigma}$,
4. if $P, Q \in \mathcal{P}_{\mathrm{URM}}$, $|P;Q| = |Q| \circ |P|$.

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 1.6. *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:*



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

A partial function $f \in E(\Omega)$ is called *URM computable* if there is an URM program P such that $|P| = f$. Note that the class $\mathcal{P}_{\mathrm{URM}}$ is denumerable, 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 URM computable functions and let $\mathcal{T}_{\mathrm{URM}}$ depict the class of all total URM computable functions. Clearly, $\mathcal{T}_{\mathrm{URM}} \subset \mathcal{F}_{\mathrm{URM}}$.

Arithmetic function can be computed by URM programs. To see this, observe that the computation of a function $f : \mathbb{N}_0^k \to \mathbb{N}_0^m$ by an URM requires to load the registers with the initial values and to read out the result. For this, define the total functions

$$\alpha_k : \mathbb{N}_0^k \to \Omega : (x_1, \ldots, x_k) \mapsto (0, x_1, \ldots, x_k, 0, 0, \ldots) \tag{1.19}$$

and

$$\beta_m : \Omega \to \mathbb{N}_0^m : (\omega_0, \omega_1, \omega_2, \ldots,) \mapsto (\omega_1, \omega_2, \ldots, \omega_m). \tag{1.20}$$

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

$$\|P\|_{k,m} = \beta_m \circ |P| \circ \alpha_k. \tag{1.21}$$

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

$$f = \|P\|_{k,m}. \tag{1.22}$$

Examples 1.7. • Addition of natural numbers is URM computable. To see this, consider the URM program

$$P_+ = (A1; S2)2. \tag{1.23}$$

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

$$\|P_+\|_{2,1}(x, y) = x + y, \quad x, y \in \mathbb{N}_0. \tag{1.24}$$

- Multiplication of natural number is URM computable. To see this, consider the URM program

$$P_\cdot = (A3; A4; S1)1; ((A1; S3)3; S2; (A0; A3; S4)4; (A3; S0)0)2. \tag{1.25}$$

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

$$\|P_\cdot\|_{2,1}(x, y) = xy, \quad x, y \in \mathbb{N}_0. \tag{1.26}$$

- Asymmetric difference is URM computable. For this, consider the URM program

$$P_{\dot{-}} = (S1; S2)2. \tag{1.27}$$

This program transforms the initial state (ω_n) into the state $(\omega_0, \omega_1 \dot{-} \omega_2, \omega_2, \omega_3, \ldots)$ and thus yields the URM computable function

$$\|P_{\dot{-}}\|_{2,1}(x, y) = x \dot{-} y, \quad x, y \in \mathbb{N}_0. \tag{1.28}$$

- Consider the sign function $\mathrm{sgn} : \mathbb{N}_0 \to \mathbb{N}_0$ given by $\mathrm{sgn}(x) = 1$ if $x > 0$ and $\mathrm{sgn}(x) = 0$ if $x = 0$. This function is URM computable since it is calculated by the URM program

$$P_{\mathrm{sgn}} = (A2; S1)1; (A1; (S2)2)2. \tag{1.29}$$

♦

Note that URM programs are *translation invariant* in the sense that if an URM program P manipulates the registers $R_{i_1}, \ldots, R_{i_k}$, 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 $P = (A1; S2)2$, $(P)[+5] = (A6; S7)7$.

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 $(x_1, \ldots, x_k) \in \mathbb{N}_0^k$,

$$(P \circ \alpha_k)(x_1, \ldots, x_k) = \begin{cases} (0, f(x_1, \ldots, x_k), 0, 0, \ldots) & \text{if } (x_1, \ldots, x_k) \in \mathrm{dom}(f), \\ \uparrow & \text{otherwise.} \end{cases} \tag{1.30}$$

A normal URM-program computes a 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 1.8. *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 σ is the largest number of a register that contains a non-zero value in the final state of computation. Then the corresponding normal URM-program is given by

$$P = Q; (S0)0; (S2)2; \ldots; (S\sigma)\sigma. \tag{1.31}$$

□

Next, we present a non-URM computable function. For this, let $P_0, P_1, P_2, \ldots$ be the sequence of all URM programs listed according to length and lexicographic order (used as tie breaker). Consider the function $f : \mathbb{N}_0 \to \mathbb{N}_0$ defined as

$$f(x) = \begin{cases} 1 \text{ if } \|P\|_{1,1}(x) = 0, \\ 0 \text{ otherwise.} \end{cases} \tag{1.32}$$

Lemma 1.9. *The function f is not URM computable.*

Proof. Suppose f is URM computable. Then there is an URM program P_n in the list such that $\|P_n\|_{1,1} = f$. But f is total and so $\|P_n\|_{1,1}(x) = f(x)$ for all $x \in \mathbb{N}_0$. Thus $\|P_n\|_{1,1}(n) = f(n)$ contradicting the definition of f. □

Finally, we introduce two programs that are useful for the transport or distribution of the contents of registers. The first function *reloads* the content of register R_i into $k \geq 0$ registers $R_{j_1}, \ldots, R_{j_k}$ 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_n) into the state (ω'_n) 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 $k \geq 0$ registers $R_{j_1}, \ldots, R_{j_k}$ 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_n) into the state (ω'_n) 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 recursion 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 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

A *semi-Peano structure* is a triple $\mathcal{A} = (A, \alpha, a)$ consisting of a non-empty set A, an unary operation $\alpha : A \to A$, and an element $a \in A$. Let $\mathcal{A} = (A, \alpha, a)$ and $\mathcal{B} = (B, \beta, b)$ be semi-Peano structures. A mapping $\phi : A \to B$ is called a *morphism*, written $\phi : \mathcal{A} \to \mathcal{B}$, if ϕ commutes with the unary operations, i.e., $\beta \circ \phi = \phi \circ \alpha$, and correspondingly assigns the distinguished elements, i.e., $\phi(a) = b$.

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

- α is injective,
- $a \notin \operatorname{ran}(\alpha)$, 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 (1958-1932).

Lemma 2.1. *If $\mathcal{A} = (A, \alpha, a)$ is a Peano structure, $A = \{\alpha^n(a) \mid n \in \mathbb{N}_0\}$.*

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

Lemma 2.2. *If $\mathcal{A} = (A, \alpha, a)$ is a Peano structure, then for all $m, n \in \mathbb{N}_0$, $m \neq n$ implies $\alpha^m(a) \neq \alpha^n(a)$.*

Proof. Define T as the set of all elements $\alpha^m(a)$ such that $\alpha^n(a) \neq \alpha^m(a)$ for all $n > m$.

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 $\alpha(x) = \alpha^{m+1}(x) \notin T$. 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 the *fundamental Lemma* for Peano structures.

Proposition 2.3. *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. First, the existence of ϕ is shown. For this, define $\phi(\alpha^n(a)) = \beta^n(b)$ for all $n \in \mathbb{N}_0$. The above Lemmas imply that ϕ is a mapping. 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 $(\phi \circ \alpha)(x) = \phi(\alpha^{m+1}(a)) = \beta^{m+1}(b) = \beta(\beta^m(b)) = \beta(\phi(\alpha^m(a))) = (\beta \circ \phi)(x)$. Thus ϕ is a morphism.

Second, the uniqueness of ϕ is proved. 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 $\alpha(x) \in T$. By the induction axiom, $T = A$ and so $\phi = \psi$. □

The fundamental Lemma immediately leads to a result of Richard Dedekind (1931-1916).

Corollary 2.4. *There is one Peano structure up to isomorphism.*

Proof. The statement to be proved is that if $\mathcal{A} = (A, \alpha, a)$ and $\mathcal{B} = (B, \beta, b)$ are Peano structures, 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 $\phi \circ \psi = \mathrm{id}_B$. Indeed, 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 map $\mathrm{id}_A : \mathcal{A} \to \mathcal{A} : x \mapsto x$ is a morphism. Finally, by the fundamental Lemma, $\psi \circ \phi = \mathrm{id}_A$ and similarly $\phi \circ \psi = \mathrm{id}_B$. □

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

Proposition 2.5. *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 $y \in \mathbb{N}_0$.*

Example 2.6. 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. $f_+(x, y + 1) = \nu(f_+(x, y)) = f_+(x, y) + 1$ for all $x, y \in \mathbb{N}_0$.

Indeed, consider the semi-Peano structure $(\mathbb{N}_0, \nu, x)$. 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. $f_x(y+1) = \nu(f_x(y)) = f_x(y) + 1$ 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 $x, y \in \mathbb{N}_0$. Thus addition can be recursively defined. ⧫

Proposition 2.7. *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. $f_{\boldsymbol{x}}(0) = (0, g(\boldsymbol{x}))$,
2. $f_{\boldsymbol{x}}(y+1) = \alpha_{\boldsymbol{x}}(y, f_{\boldsymbol{x}}(y)) = (y+1, h(\boldsymbol{x}, y, f_{\boldsymbol{x}}(y)))$ 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 $f(\boldsymbol{x}, y) = \pi_2^{(2)} \circ f_{\boldsymbol{x}}(y)$, $\boldsymbol{x} \in \mathbb{N}_0^k$ and $y \in \mathbb{N}_0$. □

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

2.2 Primitive Recursive Functions

The set 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 *unary 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 function* $\pi_k^{(n)} : \mathbb{N}_0^n \to \mathbb{N}_0 : (x_1, \ldots, x_n) \mapsto x_k$, where $n \geq 1$ and $1 \leq k \leq n$.

Using these functions, more complex primitive recursive functions are recursively defined.

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

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

Proposition 2.8. *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 with $(h_1, \ldots, h_k)$. By induction, it can be assumed that the functions $g, h_1, \ldots, h_k$ are total. Then by definition, the function f is also total.

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

Examples 2.9. The basic functions of addition and multiplication are primitive recursive.

1. The function $f_+ : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x, y) \mapsto x + y$ given in Example 2.6 has the properties
   a) $f_+(x, 0) = \mathrm{id}_{\mathbb{N}_0}(x) = x$ for all $x \in \mathbb{N}_0$ and
   b) $f_+(x, y + 1) = (\nu \circ \pi_3^{(3)})(x, y, f_+(x, y)) = f_+(x, y) + 1$ for all $x, y \in \mathbb{N}_0$.
2. Define the function $f_\cdot : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x, y) \mapsto xy$ inductively as follows:
   a) $f_\cdot(x, 0) = 0$ for all $x \in \mathbb{N}_0$ and
   b) $f_\cdot(x, y + 1) = f_\cdot(x, y) + x = f_+(x, f_\cdot(x, y))$ for all $x, y \in \mathbb{N}_0$.
   This leads to the primitive recursive scheme
   a) $f_\cdot(x, 0) = c_0^{(1)}(x)$ for all $x \in \mathbb{N}_0$ and
   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 2.10. *Each primitive recursive function is URM computable.*

Proof. First, claim that each basic function is URM computable.

1. For the 0-ary constant function define the URM program $P_0^{(0)} = A0; S0$ giving $\|P_0^{(0)}\|_{0,1} = c_0^{(0)}$.
2. For the unary constant function consider the URM program $P_0^{(1)} = (S1)1$ providing $\|P_0^{(1)}\|_{1,1} = c_0^{(1)}$.
3. For the successor function take the URM program $P_{+1} = A1$ yielding $\|P_{+1}\|_{1,1} = \nu$.
4. For the projection function $\pi_k^{(n)}$, $n \geq 1$ and $1 \leq k \leq n$, define the URM program $P_{p(n,k)} = R(k; 0); (S1)1; R(0; 1)$ showing that $\|P_{n,k}\|_{n,1} = \pi_k^{(n)}$.

Second, in view of the composition $f = g(h_1, \ldots, h_k)$, it can be assumed by induction that there are normal URM programs P_g and $P_{h_1}, \ldots, P_{h_k}$ such that $\|P_g\|_{k,1} = g$ and $\|P_{h_i}\|_{m,1} = h_i$, $1 \leq i \leq k$. 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 $j \geq 1$,
- reload the result $h_i(x_1, \ldots, x_n)$ into R_{n+i}.
- reload the values in R_{n+i} are reloaded into R_i, $1 \leq i \leq k$,
- compute the function $g(h_1(\boldsymbol{x}), \ldots, h_k(\boldsymbol{x}))$.

To this end, for $1 \leq i \leq k$, put

$$Q_i = C(1; n + k + 2); \ldots; C(n, 2n + k + 1); (P_{h_i})[+n + k + 1]; R(n + k + 2; n + i) \tag{2.6}$$

and set

$$P_f = Q_1; \ldots; Q_k; (S1)1; \ldots; (Sn)n; R(n+1;1); \ldots; R(n+k;k); P_g. \tag{2.7}$$

It follows that $\|P_f\|_{n,1} = f$.

Third, in view of the primitive recursion $f = \mathrm{pr}(g,h)$, the computation of the function value $f(\boldsymbol{x}, y)$ can be accomplished in $y+1$ steps:

- compute $f(\boldsymbol{x}, 0) = g(\boldsymbol{x})$, and
- for each $1 \leq i \leq y$, calculate $f(\boldsymbol{x}, i) = h(\boldsymbol{x}, i-1, f(\boldsymbol{x}, i-1))$.

By induction, there are normal URM programs P_g and P_h such that $\|P_g\|_{k,1} = g$ and $\|P_h\|_{k+2,1} = h$. Moreover, the registers $R_1, \ldots, R_{k+1}$ contain the input values for the computation of $f(\boldsymbol{x}, y)$. The register R_{k+2} is used as a counter and the URM programs $(P_g)[+k+3]$ and $(P_h)[+k+3]$ make use of the registers R_{k+3+j}, where $j \geq 0$. Define

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

It follows that $\|P_f\|_{k+1,1} = f$. □

The URM programs for composition and primitive recursion also make 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 with $(h_1, \ldots, h_k)$ 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 2.11. *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 2.12. *Let $g : \mathbb{N}_0^k \to \mathbb{N}_0$ and $h : \mathbb{N}_0^{k+2} \to \mathbb{N}_0$ be partial functions. There is a unique function $f : \mathbb{N}_0^{k+1} \to \mathbb{N}_0$ such that*

1. $(\boldsymbol{x}, 0) \in \mathrm{dom}(f)$ *iff* $\boldsymbol{x} \in \mathrm{dom}(g)$ *for all* $\boldsymbol{x} \in \mathbb{N}_0^k$,
2. $(\boldsymbol{x}, y+1) \in \mathrm{dom}(f)$ *iff* $(\boldsymbol{x}, y) \in \mathrm{dom}(f)$ *and* $(\boldsymbol{x}, y, f(\boldsymbol{x}, y)) \in \mathrm{dom}(h)$ *for all* $\boldsymbol{x} \in \mathbb{N}_0^k$, $y \in \mathbb{N}_0$,
3. $f(\boldsymbol{x}, 0) = g(\boldsymbol{x})$ *for all* $\boldsymbol{x} \in \mathbb{N}_0^k$, *and*
4. $f(\boldsymbol{x}, y+1) = h(\boldsymbol{x}, y, f(\boldsymbol{x}, y))$ *for all* $\boldsymbol{x} \in \mathbb{N}_0^k$, $y \in \mathbb{N}_0$.

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

Proposition 2.13. *The class of URM computable functions is closed under primitive recursion; that is, if* $g : \mathbb{N}_0^k \to \mathbb{N}_0$ *and* $h : \mathbb{N}_0^{k+2} \to \mathbb{N}_0$ *are URM computable,* $f = \mathrm{pr}(g, h)$ *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, let $\mathcal{T}_{\mathrm{URM}}$ depict the class of all URM computable total functions, and let $\mathcal{T}$ signify the class of all total functions.

Proposition 2.14. *The classes* $\mathcal{P}$, $\mathcal{T}_{\mathrm{URM}}$, *and* $\mathcal{T}$ *are primitively closed.*

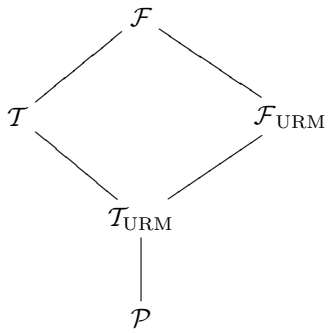
In particular, the class $\mathcal{P}$ of primitive recursive functions is the smallest class of functions which is primitively closed. Indeed, we have

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

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

Proposition 2.15. *The classes* $\mathcal{F}_{\mathrm{URM}}$ *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 $\mathcal{T}_{\mathrm{URM}} \subset \mathcal{F}_{\mathrm{URM}}$ and $\mathcal{T} \subset \mathcal{F}$ are obvious, while the strict inclusions $\mathcal{T}_{\mathrm{URM}} \subset \mathcal{T}$ and $\mathcal{F}_{\mathrm{URM}} \subset \mathcal{F}$ follow by counting arguments. However, the strict inclusion $\mathcal{P} \subset \mathcal{T}_{\mathrm{URM}}$ is not so obvious. An example of a total URM computable function that is not primitive recursive will be given later.

2.3 Closure Properties

This section provides a small repository of recursive 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 2.16. *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

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

□

Examples 2.17. Three important special cases for 2-ary functions are the permutation of variables: $f^\phi : (x, y) \mapsto (y, x)$, the adjunct of variables: $f^\phi : (x, y) \mapsto x$, and the identification of variables: $f^\phi : (x, y) \mapsto (x, x)$. ♦

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 2.18. *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 $\boldsymbol{a}$ 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 2.19. *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 set $\mathbb{N}_0^k$. Moreover, let $g_i : \mathbb{N}_0^k \to \mathbb{N}_0$, $1 \leq i \leq r$, be total functions. Define the function

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

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

Proposition 2.20. *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. In case of $r = 2$, the function f is given in prefix notation as follows:

$$f = +(\cdot(g_1, \mathrm{csg}(h_1)), \cdot(g_2, \mathrm{csg}(h_2))). \tag{2.18}$$

The general case follows by induction on r. □

Example 2.21. Let csg denote the *cosign function*, i.e., $\mathrm{csg}(x) = 0$ if $x > 0$ and $\mathrm{csg}(x) = 1$ if $x = 0$. The mappings $h_1 : x \mapsto x \bmod 2$ and $h_2 : x \mapsto \mathrm{csg}(x \bmod 2)$ define a partition of the set $\mathbb{N}_0$ into the set of even natural numbers and the set of odd natural numbers. Using this, a function defined by cases is the following:

$$f(x) = \begin{cases} x/2 & \text{if } x \text{ is even,} \\ (x+1)/2 & \text{if } x \text{ is odd.} \end{cases} \tag{2.19}$$

♦

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 2.22. *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 2.23. 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 is primitive recursive and thus the bounded product, $\Pi f(x) = x!$, $x \in \mathbb{N}_0$, is primitive recursive. ♦

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

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

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 $y + 1$.

Proposition 2.24. *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

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

Define the $k+2$-ary functions

$$\begin{array}{ll} g_1 : (\boldsymbol{x}, y, z) \mapsto z, & h_1 : (\boldsymbol{x}, y, z) \mapsto z \dot{-} y \\ g_2 : (\boldsymbol{x}, y, z) \mapsto y+1, & h_2 : (\boldsymbol{x}, y, z) \mapsto \mathrm{csg}(z \dot{-} y) * \mathrm{sgn}(f(\boldsymbol{x}, y+1)) \\ g_3 : (\boldsymbol{x}, y, z) \mapsto y+2, & h_3 : (\boldsymbol{x}, y, z) \mapsto \mathrm{csg}(h_1(\boldsymbol{x}, y, z) * h_2(\boldsymbol{x}, y, z)). \end{array} \tag{2.27}$$

These functions are primitive recursive. Moreover, the functions h_1, h_2, and h_3 provide a partition of $\mathbb{N}_0$. Thus by case distinction, the following function is primitive recursive:

$$g(\boldsymbol{x}, y, z) = \begin{cases} g_1(\boldsymbol{x}, y, z) \text{ if } h_1(\boldsymbol{x}, y, z) = 0, \\ g_2(\boldsymbol{x}, y, z) \text{ if } h_2(\boldsymbol{x}, y, z) = 0, \\ g_3(\boldsymbol{x}, y, z) \text{ if } h_3(\boldsymbol{x}, y, z) = 0. \end{cases} \tag{2.28}$$

Note that $h_1(\boldsymbol{x}, y, \overline{\mu} f(\boldsymbol{x}, y)) = 0$ means $\overline{\mu} f(\boldsymbol{x}, y)) \leq y$ and $h_2(\boldsymbol{x}, y, \overline{\mu} f(\boldsymbol{x}, y)) = 0$ is equivalent to $\overline{\mu} f(\boldsymbol{x}, y)) = y + 1$ and $f(\boldsymbol{x}, y+1)) = 0$. It follows that the bounded minimalization $\overline{\mu} f$ corresponds to the primitive recursive scheme $\overline{\mu} f = \mathrm{pr}(s, g)$, where $s : \mathbb{N}_0^k \to \mathbb{N}_0$ is defined by $s(\boldsymbol{x}) = \mathrm{sgn}(f(\boldsymbol{x}, 0))$. □

Example 2.25. 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. Thus the value z can be provided by bounded minimalization. To this end, define the function $f : \mathbb{N}_0^3 \to \mathbb{N}_0 : (x, y, z) \mapsto \operatorname{csg}(y \cdot z \dot{-} x)$; that is,

$$f(x, y, z) = \begin{cases} 0 \text{ if } y \cdot z > x, \\ 1 \text{ otherwise.} \end{cases} \tag{2.30}$$

Applying bounded minimalization to f yields the primitive recursive function

$$\overline{\mu} f(x, y, z) = \begin{cases} \text{smallest } j \leq z \text{ with } y \cdot j > x \text{ if } j \text{ exists}, \\ z + 1 \qquad\qquad\qquad\qquad\quad \text{otherwise.} \end{cases} \tag{2.31}$$

Identification of variables provides the primitive recursive function

$$(\overline{\mu} f)' : (x, y) \mapsto \overline{\mu} f(x, y, x), \tag{2.32}$$

which is given as

$$(\overline{\mu} f)'(x, y) = \begin{cases} \text{smallest } j \leq x \text{ with } y \cdot j > x \text{ if } y \geq 1, \\ x + 1 \qquad\qquad\qquad\qquad\quad \text{if } y = 0. \end{cases} \tag{2.33}$$

It follows that $\div(x, y) = (\overline{\mu} f)'(x, y) \dot{-} 1$. Finally, the remainder of x modulo y is given by $\operatorname{rem}(x, y) = y \dot{-} x \cdot \div(x, y)$ and thus is also primitive recursive. ♦

Pairing Functions

A pairing function uniquely encodes pairs of natural numbers into single natural numbers. Each 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

$$J_2(m, n) = \frac{1}{2}(m + n)(m + n + 1) + m. \tag{2.34}$$

Proposition 2.26. *The Cantor function J_2 is a pairing function.*

Proof. Write the elements of $\mathbb{N}_0^2$ into a table:

$$\begin{array}{llll} (0,0) & (0,1) & (0,2) & (0,3) \ldots \\ (1,0) & (1,1) & (1,2) & (1,3) \ldots \\ (2,0) & (2,1) & (2,2) & \ldots \\ (3,0) & (3,1) & \ldots & \\ \ldots & & & \end{array}$$

Write down the elements of this table by moving along the diagonals which go from north-east to south-west; that is, write them in sequence (0,0), (0,1), (1,0), (0,2), (1,1), (2,0), (0,3), (1,2), (2,1), (3,0), (4,0), and so on. There are $k+1$ pairs (m,n) with $m+n=k$ and thus the kth diagonal contains $k+1$ elements, $k \geq 1$. Thus the pair (m,n) occurs at the position $1+2+\ldots+(m+n)+m$, which equals $J_2(m,n)$. Finally, the integral division function gives $f(m,n) = \div((m,n)(m+n+1),2)+m$, and so J_2 is primitive recursive. □

The inverse of the Cantor function J_2 is given by coordinate functions $K_2, L_2 : \mathbb{N}_0 \to \mathbb{N}_0$ such that

$$J_2^{-1}(n) = (K_2(n), L_2(n)), \quad n \in \mathbb{N}_0. \tag{2.35}$$

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

$$\frac{1}{2}s(s+1) \leq n < \frac{1}{2}(s+1)(s+2) \tag{2.36}$$

and put

$$m = n - \frac{1}{2}s(s+1). \tag{2.37}$$

Then $m \leq s$, since $m \leq \frac{1}{2}(s+1)(s+2) - 1 - \frac{1}{2}s(s+1) = s$. Finally, put

$$K_2(n) = m \quad \text{and} \quad L_2(n) = s - m. \tag{2.38}$$

Proposition 2.27. *The coordinate functions K_2 and L_2 are primitive recursive.*

Proof. Let $n \in \mathbb{N}_0$. The corresponding number s in (2.36) can be determined by bounded minimalization. Then by (2.37) and (2.38), $J_2(K_2(n), L_2(n)) = J_2(m, s-m) = \frac{1}{2}s(s+1) - m = n$ and so (2.35) follows. □

The Cantor function J_2 can be used to obtain bijections $J_n : \mathbb{N}_0^n \to \mathbb{N}_0$ for all n. For this, define J_1 to be the identity mapping and let J_2 be the Cantor pairing function. If J_n is already given, define J_{n+1} as follows:

$$J_{n+1}(x_1, \ldots, x_n, x_{n+1}) = J_2(J_n(x_1, \ldots, x_n), x_{n+1}). \tag{2.39}$$

Proposition 2.28. *The functions J_n are primitive recursive bijections.*

Proof. The functions J_1 and J_2 are primitive recursive bijections. Suppose that J_n is a primitive recursive bijection. The function J_{n+1} is given by the composition

$$J_{n+1} = J_2(J_n(\pi_1^{(n+1)}, \ldots, \pi_n^{(n+1)}), \pi_{n+1}^{(n+1)}) \tag{2.40}$$

and thus is primitive recursive.

Next, let $J_{n+1}(\boldsymbol{x}, y) = J_{n+1}(\boldsymbol{x}', y')$, where $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{N}_0^n$ and $y, y' \in \mathbb{N}_0$. Then $J_2(J_n(\boldsymbol{x}), y) = J_2(J_n(\boldsymbol{x}'), y')$ and so $J_n(\boldsymbol{x}) = J_n(\boldsymbol{x}')$ and $y = y'$), since J_2 is one-to-one. Moreover, by induction, J_n is one-to-one and thus $\boldsymbol{x} = \boldsymbol{x}'$. It follows that J_{n+1} is one-to-one.

Finally, let $k \in \mathbb{N}_0$. Then $J_2(k) = (k', y)$ for some $k', y \in \mathbb{N}_0$, since J_2 is onto. Moreover, by induction, J_n is onto and so $J_n(\boldsymbol{x}) = k'$ for some $\boldsymbol{x} \in \mathbb{N}_0^n$. Thus $J_{n+1}(\boldsymbol{x}, y) = J_2(J_n(\boldsymbol{x}), y) = J_2(k', y) = k$ and hence J_{n+1} is onto. □

Proposition 2.29. *The inverse of the function J_n is composed of the coordinate functions K_2 and L_2 as follows:*

$$J_n^{-1}(k) = (K_2^{n-1}(k), L_2 \circ K_2^{n-2}(k), \ldots, L_2 \circ K_2(k), L_2(k)), \quad k \in \mathbb{N}_0. \tag{2.41}$$

The function J_n^{-1} is primitive recursive.

Proof. The assertion (2.41) is clear for $n = 2$. Let $n \geq 2$ and let $J_{n+1}(\boldsymbol{x}, y) = k$ for some $\boldsymbol{x} \in \mathbb{N}_0^n$ and $y \in \mathbb{N}_0$. Then $k = J_2(J_n(\boldsymbol{x}), y) = J_2(K_2(k), L_2(k))$. Thus $J_n(\boldsymbol{x}) = K_2(k)$. By hypothesis, $\boldsymbol{x} = J_n^{-1}(K_2(k)) = (K_2^n(k), L_2 \circ K_2^{n-1}(k), \ldots, L_2 \circ K_2(k))$ and so the result follows. Finally, the function J_n^{-1} is primitive recursive, since it is a composition of primitive recursive functions. □

Iteration

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

$$f^0 = \mathrm{id}_{\mathbb{N}_0} \quad \text{and} \quad f^{n+1} = f \circ f^n, \ n \geq 0. \tag{2.42}$$

The *iteration* of a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ is given by the function

$$g : \mathbb{N}_0^2 \to \mathbb{N}_0 : (x, y) \mapsto f^y(x). \tag{2.43}$$

Example 2.30. Consider the function $f : \mathbb{N}_0 \to \mathbb{N}_0 : x \mapsto 2x$. The iteration of f is the function $g : \mathbb{N}_0^2 \to \mathbb{N}_0$ given by $g(x, y) = 2^y \cdot x$. ♦

Proposition 2.31. *If f is an unary primitive recursive function, the iteration of f is also primitive recursive.*

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

$$g(x, 0) = x \tag{2.44}$$

and

$$g(x, y + 1) = f(g(x, y)) = f \circ \pi_3^{(3)}(x, y, g(x, y)), \ x, y \in \mathbb{N}_0. \tag{2.45}$$

□

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:

$$f(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathbb{N}_0^k. \tag{2.46}$$

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

$$f^0(\boldsymbol{x}) = \boldsymbol{x} \quad \text{and} \quad f^{n+1}(\boldsymbol{x}) = (f_1(f^n(\boldsymbol{x})), \ldots, f_k(f^n(\boldsymbol{x}))), \quad \boldsymbol{x} \in \mathbb{N}_0^k. \tag{2.47}$$

The definitions immediately give rise to the following result.

Proposition 2.32. *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

$$g_i : \mathbb{N}_0^{k+1} \to \mathbb{N}_0 : (\boldsymbol{x}, y) \mapsto (\pi_i^{(k)} \circ f^y)(\boldsymbol{x}), \quad 1 \leq i \leq k. \tag{2.48}$$

Proposition 2.33. *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

$$g_i(\boldsymbol{x}, 0) = x_i \tag{2.49}$$

and

$$g_i(\boldsymbol{x}, y+1) = f_i(f^y(\boldsymbol{x})) = f_i \circ \pi_{k+2}^{(k+2)}(\boldsymbol{x}, y, g_i(\boldsymbol{x}, y)), \quad \boldsymbol{x} \in \mathbb{N}_0^k, y \in \mathbb{N}_0,\ 1 \leq i \leq k. \tag{2.50}$$

□

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 2.34. Basic relations are primitive such as the following:

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

⧫

Proposition 2.35. *If S and T are primitive subsets of $\mathbb{N}_0^k$, $S \cup T$, $S \cap T$, and $\mathbb{N}_0^k \setminus S$ are also primitive.*

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

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

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

Proposition 2.36. *If S is a primitive subset of $\mathbb{N}_0^{k+1}$, the sets $\exists S$ and $\forall S$ are also primitive.*

Proof. Clearly, $\chi_{\exists S}(\boldsymbol{x}, y) = \operatorname{sgn}((\Sigma\chi_S)(\boldsymbol{x}, y))$ and $\chi_{\forall S}(\boldsymbol{x}, y) = (\Pi\chi_S)(\boldsymbol{x}, y)$. □

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 arithmetics, 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 2.37. *1. The divisibility relation $D = \{(x, y) \in \mathbb{N}_0^2 \mid x \text{ divides } y\}$ is primitive.*
2. The set P 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 is given by

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

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

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

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

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

Both functions g and h are 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. Loops of this type allow an explicit control over the loop variable.

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 the URM program $(S\sigma)\sigma$ by $Z\sigma$.

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

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

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

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

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

$$\mathcal{P}_{\mathrm{LOOP}} = \bigcup_{n \geq 0} \mathcal{P}_{\mathrm{LOOP}(n)}. \tag{2.62}$$

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

Proposition 2.38. *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 $\mathcal{F}_{\mathrm{LOOP}(n)}$ denote the class of all LOOP-n computable functions and define the class of all LOOP computable functions $\mathcal{F}_{\mathrm{LOOP}}$ as the union of LOOP-n computable functions for all $n \geq 0$:

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

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

$$P_{\dot{-}1} = \bar{K}(1;3); [\bar{K}(2;1); A2]3. \tag{2.64}$$

Then we have for input $x = 0$,

0	1	2	3	4 ...	registers
0	0	0	0	0 ...	initially
0	0	0	0	0 ...	$\bar{K}(1;3)$
0	0	0	0	0 ...	finally

and for input $x \geq 1$,

0	1	2	3	4 ...	registers
0	x	0	0	0 ...	initially
0	x	0	x	0 ...	$\bar{K}(1;3)$
0	0	0	x	0 ...	$\bar{K}(2;1)$
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 ...	finally

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

Theorem 2.40. *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, each basic primitive recursive function is LOOP computable:

1. 0-ary constant function : $\|Z0\|_{0,1} = c_0^{(0)}$,

2. unary 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 $\|\bar{K}(\sigma;1)\|_{k,1} = \pi_\sigma^{(k)}$, $\sigma \neq 1$.

Moreover, the class of LOOP computable functions is closed under composition and primitive recursion. This can be shown as in the proof of Theorem 2.10, where subtraction is replaced by the program in 2.39. 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.

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 each integer $m \geq n(P)$ and $0 \leq j \leq m$, consider the functions

$$k_j^{(m+1)}(P) : \mathbb{N}_0^{m+1} \to \mathbb{N}_0 : (x_0, x_1, \ldots, x_m) \mapsto (\pi_j \circ |P|)(x_0, x_1, \ldots, x_m, , 0, 0, \ldots), \tag{2.65}$$

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

$$\pi_j : \Omega \to \mathbb{N}_0 : (\omega_0, \omega_1, \omega_2, \ldots) \mapsto \omega_j. \tag{2.66}$$

The assertion to be shown is thus a special case of the following assertion: For all LOOP programs P, for all integers $m \geq n(P)$ and all $0 \leq j \leq m$, 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

$$k_j^{(m+1)}(P) : \boldsymbol{x} \mapsto \begin{cases} (\nu \circ \pi_j^{(m+1)})(\boldsymbol{x}) & \text{if } j = \sigma, \\ \pi_j^{(m+1)}(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{2.67}$$

This function is primitive recursive.

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

$$k_j^{(m+1)}(P) : \boldsymbol{x} \mapsto \begin{cases} (c_0^{(1)} \circ \pi_j^{(m+1)})(\boldsymbol{x}) & \text{if } j = \sigma, \\ \pi_j^{(m+1)}(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{2.68}$$

This function is primitive recursive.

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

$$k_j^{(m+1)}(P) : \boldsymbol{x} \mapsto \begin{cases} (c_0^{(1)} \circ \pi_j^{(m+1)})(\boldsymbol{x}) & \text{if } j = 0, \\ \pi_\sigma^{(m+1)}(\boldsymbol{x}) & \text{if } j = \tau, \\ \pi_j^{(m+1)}(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{2.69}$$

This function is 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

$$\begin{aligned} k_j^{(m+1)}(P)(\boldsymbol{x}) &= (\pi_j \circ P)(\boldsymbol{x}, 0, 0, \ldots) \\ &= (\pi_j \circ |R| \circ |Q|)(\boldsymbol{x}, 0, 0, \ldots) \\ &= k_j^{(m+1)}(R)(k_0^{(m+1)}(Q)(\boldsymbol{x}), \ldots, k_m^{(m+1)}(Q)(\boldsymbol{x})), \\ &= k_j^{(m+1)}(R)(k_0^{(m+1)}(Q), \ldots, k_m^{(m+1)}(Q))(\boldsymbol{x}). \end{aligned} \tag{2.70}$$

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

Finally, let $P = [Q]\sigma \in \mathcal{P}_{\mathrm{LOOP}}$, 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

$$k_j^{(m+1)}(Q; S\sigma) : \boldsymbol{x} \mapsto \begin{cases} k_j^{(m+1)}(Q)(\boldsymbol{x}) & \text{if } j \neq \sigma, \\ (f \dot{-}_1 \circ \pi_j^{(m+1)})(\boldsymbol{x}) & \text{otherwise.} \end{cases} \tag{2.71}$$

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 $k_j^{(m+1)}(Q; S\sigma)$, where $0 \leq j \leq m$. That is,

$$k^{(m+1)}(Q; S\sigma)(\boldsymbol{x}) = (k_0^{(m+1)}(Q; S\sigma)(\boldsymbol{x}), \ldots, k_m^{(m+1)}(Q; S\sigma)(\boldsymbol{x})). \tag{2.72}$$

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,

$$g(\boldsymbol{x}, 0) = \boldsymbol{x} \quad \text{and} \quad g(\boldsymbol{x}, y+1) = k^{(m+1)}(Q; S\sigma)(g(\boldsymbol{x}, y)). \tag{2.73}$$

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

$$(\pi_j^{(m+1)} \circ g)(\boldsymbol{x}, y) = k_j^{(m+1)}((Q; S\sigma)^y)(\boldsymbol{x}), \tag{2.74}$$

where $(Q; S\sigma)^y$ denotes the y-fold composition of the program $Q; S\sigma$.

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

$$|P|(\omega) = |(Q; S\sigma)\sigma|(\omega) = |(Q; S\sigma)^{\omega_\sigma}|(\omega),\ \omega \in \Omega. \tag{2.75}$$

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.,

$$k_j^{(m+1)}(P)(\boldsymbol{x}) = (\pi_j^{(m+1)} \circ g)(\boldsymbol{x}, \pi_\sigma^{(m+1)}(\boldsymbol{x})),\ \boldsymbol{x} \in \mathbb{N}_0^{m+1}. \tag{2.76}$$

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

3

Partial Recursive Functions

The partial recursive functions form a class of partial functions which are computable in an intuitive sense. In fact, the partial recursive functions are precisely the functions that can be computed by Turing machines or unlimited register machines. By Church's thesis, the partial recursive functions provide a formalization of the notion of computability. The partial recursive functions are closely related to the primitive recursive functions and their inductive definition builds upon them.

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 formalized analogue of the intuitive notion of computability. To this end, a formal analogon 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

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

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; e.g., the function $f(x, y) = (x + y) \dot{-} 3$ is total, while its minimalization μf is partial with $\mathrm{dom}(\mu f) = \{0, 1, 2, 3\}$.

- The minimalization function μf may be total even if f is partial, e.g., take the partial function $f(x, y) = x - y$ if $y \leq x$ and $f(x, y) = \uparrow$ if $y > x$, the minimalization $\mu f(x) = x$ is total with $\mathrm{dom}(\mu f) = \mathbb{N}_0$. ⧫

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 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

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

The first block P'_f provides the computation in (3.2) for $y = 0$. The second block $(Ak+1; (Sk+2)k+2; P'_f)k+2$; iteratively calculates (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 ad infinitum. It follows that this program computes the minimalization of f, i.e., $\|P_{\mu f}\|_{k,1} = \mu f$. □

Note that $P_{\mu f}$ is generally not a LOOP program even if P'_f is a LOOP program.

Algorithm 3.1 Minimalization of f.

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

3.2 GOTO Programs

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

First, define the *syntax* of GOTO programs. For this, let $V = \{x_n \mid n \in \mathbb{N}\}$ 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

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

- conditionals

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

The first component of an instruction (denoted by l) is called a *label*, the third component of the instructions $(l, x_\sigma+, m)$ and $(l, x_\sigma-, m)$ (denoted by m) is termed *next label*, and the last two components of the 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 $L(P) = \{\lambda(s_i) \mid 0 \leq i \leq q\}$ describe the set of labels in P.

A GOTO program $P = s_0; s_1; \ldots; s_q$ is called *standard* if the ith 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:

$$\begin{array}{llll} 0 & \texttt{if}\ x_2 = 0 & 3 & 1 \\ 1 & x_1 \leftarrow x_1 + 1 & 2 & \\ 2 & x_2 \leftarrow x_2 - 1 & 0 & \end{array}$$

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 program. 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. In particular, 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 register R_σ, the instruction $(l, x_\sigma \leftarrow x_\sigma - 1, m)$ decrements the register R_σ if 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 to the statement with label m is performed. The execution of a GOTO program *terminates* if the label of the instruction counter does not correspond to an instruction.

More specifically, let P be a GOTO program and 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 is chosen large enough to provide workspace for the computation, i.e., $n \geq \max\{k, \max\{\sigma \mid x_\sigma \in V(P)\}\}$.

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 the GOTO program be given as $P = s_0; s_1; \ldots; s_q$. 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 $\boldsymbol{z} = (z_0, z_1, \ldots, z_n)$, briefly $\boldsymbol{z} \vdash_P \boldsymbol{z}'$, if z_0 is a label in P, say $z_0 = \lambda(s_i)$ for some $0 \leq i \leq q$, and

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

$$\boldsymbol{z}' = (m, z_1, \ldots, z_{\sigma-1}, z_\sigma + 1, z_{\sigma+1}, \ldots, z_n), \tag{3.11}$$

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

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

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

$$\boldsymbol{z}' = \begin{cases} (k, z_1, \ldots, z_n) & \text{if } z_\sigma = 0, \\ (m, z_1, \ldots, z_n) & \text{otherwise.} \end{cases} \tag{3.13}$$

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

$$E_P : \boldsymbol{z} \mapsto \begin{cases} \boldsymbol{z}' \text{ if } \boldsymbol{z} \vdash_P \boldsymbol{z}', \\ \boldsymbol{z} \quad \text{otherwise.} \end{cases} \tag{3.14}$$

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

Proposition 3.4. *For each GOTO program P and each number $n \geq \{\sigma \mid x_\sigma \in V(P)\}$, 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 $0 \leq j \leq n$, $\pi_j^{(n+1)} \circ E_P$ is primitive recursive.*

The execution of a GOTO 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 $i \in \mathbb{N}_0$. 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. Such an event may eventually not happen. In this way, the *runtime function* of P is a partial function $Z_P : \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ given by

$$Z_P : \boldsymbol{z} \mapsto \begin{cases} \min\{t \in \mathbb{N}_0 \mid (\pi_0^{(n+1)} \circ E_P^t)(\boldsymbol{z}) \notin L(P)\} & \text{if } \{\ldots\} \neq \emptyset, \\ \uparrow & \text{otherwise.} \end{cases} \tag{3.15}$$

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; that is, the function

$$E'_P : \mathbb{N}_0^{n+2} \to \mathbb{N}_0^{n+1} : (\boldsymbol{z}, t) \mapsto E_P^t(\boldsymbol{z}). \tag{3.16}$$

But 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:

$$E''_P : \mathbb{N}_0^{n+2} \to \mathbb{N}_0^{n} : (\boldsymbol{z}, t) \mapsto (\chi_{L(P)} \circ \pi_0^{(n+1)} \circ E'_P)(\boldsymbol{z}, t). \tag{3.17}$$

This function has the property that

$$E''_P(\boldsymbol{z}, t) = \begin{cases} 1 \text{ if } E_P^t(\boldsymbol{z}) = (z'_0, z'_1, \ldots, z'_n) \text{ and } z'_0 \in L(P), \\ 0 \text{ otherwise.} \end{cases} \tag{3.18}$$

But by definition $Z_P = \mu E''_P$ and so the result follows. □

The *residual step function* of a 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:

$$R_P(\boldsymbol{z}) = \begin{cases} E'_P(\boldsymbol{z}, Z_P(\boldsymbol{z})) & \text{if } \boldsymbol{z} \in \mathrm{dom}(Z_P), \\ \uparrow & \text{otherwise.} \end{cases} \tag{3.19}$$

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 function and hence 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

$$f = \|P\|_{k,1}. \tag{3.20}$$

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 must be 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 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
...				

The program provides the total function $\|P_+\|_{2,1}(x_1, x_2) = x_1 + x_2$ and the (total) runtime function $Z_{P_+}(0, x_1, x_2) = 3 \cdot x_2 + 1$. ♦

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 translated 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 that does not make use of the register R_0. Write P as a string $P = \tau_0\tau_1 \dots \tau_q$, where each τ_i is a substring 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 $\phi(P) = s_0; s_1; \dots; s_q$ is established that has the required property $\|P\|_{k,1} = \|\phi(P)\|_{k,1}$, as claimed. □

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

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

These substrings give rise to the following GOTO program:

$$\begin{array}{llll} 0 & \texttt{if } x_2 = 0 & 4 & 1 \\ 1 & x_1 \leftarrow x_1 + 1 & 2 & \\ 2 & x_2 \leftarrow x_2 - 1 & 3 & \\ 3 & \texttt{if } x_2 = 0 & 4 & 1 \end{array}$$

♦

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 number

$$G(\omega) = p_0^{\omega_0} p_1^{\omega_1} p_2^{\omega_2} \cdots. \tag{3.21}$$

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

$$G_i(x) = (x)_i, \tag{3.22}$$

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 2.37.

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

$$|P|(\omega) = \omega' \quad \Longleftrightarrow \quad |\overline{P}|(0, G(\omega)) = (0, G(\omega')). \tag{3.23}$$

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

$$|M(k)|(0, x) = (0, k \cdot x), \tag{3.24}$$

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

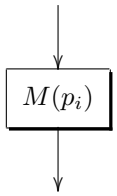
$$|T(k)|(0, x) = \begin{cases} (1, x) \text{ if } k \text{ divides } x, \\ (0, x) \text{ otherwise.} \end{cases} \tag{3.26}$$

For instance, the GOTO-2 program $M(k)$ can be implemented by two consecutive loops: Initially, $x_1 = 0$ 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. After the loop, $x_1 = 0$ and $x_2 = k \cdot x$. This leads to the following standard program:

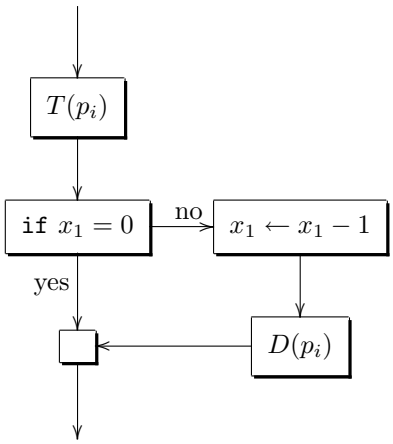
0	**if** $x_2 = 0$	$k+2$	1
1	$x_2 \leftarrow x_2 - 1$	2	
2	$x_1 \leftarrow x_1 + 1$	3	
	$\dots$		
$k+1$	$x_1 \leftarrow x_1 + 1$	0	
$k+2$	**if** $x_1 = 0$	$k+5$	$k+3$
$k+3$	$x_2 \leftarrow x_2 + 1$	$k+4$	
$k+4$	$x_1 \leftarrow x_1 - 1$	$k+2$	

Note that these GOTO-2 programs can be realized as standard programs.

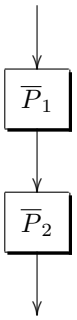
The assignment $P \mapsto \overline{P}$ will be established by making use of the inductive definition of URM programs. For this, flow diagrams will be employed to simplify the notation. First, the program $\overline{Ai}$ can be realized by the flow chart



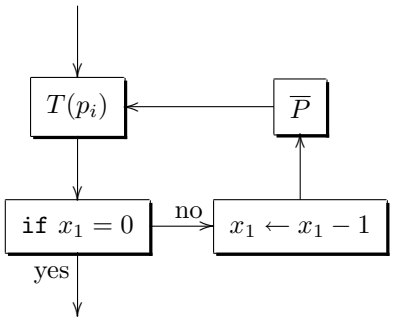
Second, the program $\overline{Si}$ is given by the flow diagram



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 program in pseudo code:

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

♦

By using the inductive definition of URM programs, the assignment $P \mapsto \overline{P}$ is well-defined. However, 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 initial values and to identify the results. For this, define the primitive recursive functions

$$\hat{\alpha}_k : \mathbb{N}_0^k \to \mathbb{N}_0^2 : (x_1, \ldots, x_k) \mapsto (0, G(0, x_1, \ldots, x_k, 0, \ldots)) \tag{3.27}$$

and

$$\hat{\beta}_m : \mathbb{N}_0^2 \to \mathbb{N}_0^m : (a, b) \mapsto (G_1(b), \ldots, G_m(b)). \tag{3.28}$$

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

$$\|P\|_{k,m} = \hat{\beta}_m \circ |\overline{P}| \circ \hat{\alpha}_k. \tag{3.29}$$

3.5 Church's Thesis

The 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 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. This thesis characterizes the nature of computation and cannot be formally proved. Nevertheless, it has 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 5.10).

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 earliest-discovered example of a total computable function that is not primitive recursive.

4.1 The 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 programs is strict.

Define a sequence (B_n) of unary 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 power of B_n. 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 $x, n, p \in \mathbb{N}_0$:*

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

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

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

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

$$B_n(x) < B_n(x+1), \tag{4.7}$$

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

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:

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

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. □

It is an important fact that the $n+1$th small Ackermann function grows faster than any power of the nth Ackermann function.

Proposition 4.3. *For all number n and p, there is a number $x_0 \geq 0$ 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 $x \geq 0$, $B_n^0(x) = x < x + 1 \leq B_{n+1}(x)$ by (4.6) and (4.7). 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

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

$$
\begin{aligned}
&< B_n^2(B_{n+1}(x \dot{-} 2)), \quad \text{by induction, } x \geq x_0' + 2,\\
&= B_n^2(B_n^{x \dot{-} 2}(1))\\
&= B_n^x(1)\\
&= B_{n+1}(x), \quad x \geq 2.
\end{aligned}
$$

□

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

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

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

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

This program computes $B_n^x(1)$. 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

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 to show that Ackermann's function is not primitive recursive.

For this, assume that the LOOP programs do not use the register R_0 in order to perform their task. This is not an essential restriction since it can always be the achieved by transformation of variables. The register R_0 will be solely used to determine the runtime of the 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 computes the runtime of P given by the number of elementary operations (addition, set to zero, copy):

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

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

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

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

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

The definitions immediately lead 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

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

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

Proposition 4.6. *If P is a LOOP-n program, P' is also a LOOP-n program and has 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

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

The corresponding runtime program is

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

⧫

Next, introduce a function $\lambda : \mathcal{P}_{\text{LOOP}} \to \mathbb{N}_0$ that assigns to each LOOP program a measure of *complexity*, which is inductively defined as follows:

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

For each LOOP program P, $\lambda(P)$ is the number of LOOP-0 subprograms of P plus twice the number of iterations of P.

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

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

⧫

A k-ary total function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is said to be*bounded* by an unary total function $g : \mathbb{N}_0 \to \mathbb{N}_0$ if

$$f(\boldsymbol{x}) \leq g(\max(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathbb{N}_0^k, \tag{4.37}$$

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 $k \geq \max\{n(Q), n(R)\}$,

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

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

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

The last inequality follows from the assertion $B_n^{x \cdot a}(y) \leq B_{n+1}^{a+2}(y)$ that holds for all $x \leq y$. Indeed, by the properties of the small Ackermann functions, $B_n^{x \cdot a}(y) \leq B_n^{y \cdot a}(y) \leq B_n^{y \cdot a}(B_{n+1}(y)) = B_n^{y \cdot a}(B_n^y(1)) = B_n^{y(a+1)}(1) = B_{n+1}(y(a+1)) \leq B_{n+1}(y \cdot 2^{a+1}) = B_{n+1}(B_1^{a+1}(y)) \leq B_{n+1}(B_{n+1}^{a+1}(y)) = B_{n+1}^{a+2}(y))$. □

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 $k \geq n(P)$,

$$\max(\|\gamma(P)\|_{k,k}(\boldsymbol{x})) \leq B_n^m(\max(\boldsymbol{x})). \tag{4.42}$$

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

For sake of completeness, the converse of the above proposition also holds. It is known as the theorem of Meyer and Ritchie (1967).

Theorem 4.11. *(Meyer-Ritchie) Let $n \geq 2$. A 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 allows to show 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 the fact that by Proposition 4.3, B_{n+1} grows faster than any power of B_n. □

4.3 Ackermann's Function

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

$$A(0, n) = n + 1, \tag{4.43}$$

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

$$A(m + 1, n + 1) = A(m, A(m + 1, n)). \tag{4.45}$$

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

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

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

- $A_0(y) = y + 1 = \nu(y + 3) - 3$,
- $A_1(y) = y + 2 = f_+(2, y + 3) - 3$,
- $A_2(y) = 2y + 3 = f_\cdot(2, y + 3) - 3$,
- $A_3(y) = 2^{y+3} - 3 = f_{\exp}(2, y + 3) - 3$,
- $A_4(y) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} - 3 = f_{\mathrm{itexp}}(2, y + 3) - 3$, *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

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

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

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

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 2-ary function $A : \mathbb{N}_0^2 \to \mathbb{N}_0$ given by

$$A(x,y) = B_x(y), \quad x, y \in \mathbb{N}_0. \tag{4.46}$$

This function is a variant of the original Ackermann function.

Proposition 4.15. *The function A in (4.46) is given by the equations*

$$A(0,y) = B_0(y), \tag{4.47}$$
$$A(x+1,0) = 1, \tag{4.48}$$
$$A(x+1,y+1) = A(x, A(x+1,y)). \tag{4.49}$$

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

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

Example 4.16.

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

$$
\begin{aligned}
&= A(0, A(1, 3)) \\
&= A(0, A(0, A(1, 2))) \\
&= A(0, A(0, A(0, A(1, 1)))) \\
&= A(0, A(0, A(0, A(0, A(1, 0))))) \\
&= A(0, A(0, A(0, A(0, 1)))) \\
&= A(0, A(0, A(0, 2))) \\
&= A(0, A(0, 4)) \\
&= A(0, 6) \\
&= 8.
\end{aligned}
$$

⧫

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

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

$$A(x, y) \leq B_n^p(\max\{x, y\}). \tag{4.50}$$

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

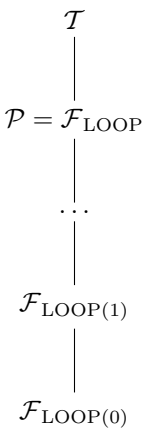
$$B_n^p(y) < B_{n+1}(y). \tag{4.51}$$

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

$$A(n + 1, y) \leq B_n^p(\max\{n + 1, y\}) = B_n^p(y) < B_{n+1}(y) = A(n + 1, y). \tag{4.52}$$

□

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



All inclusions are strict.

5

Acceptable Programming Systems

Acceptable programming systems form the basis for the abstract development of computability theory. This chapter provides some 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. The key to these results lies in the Gödel numbering of GOTO programs.

5.1 Gödel Numbering of GOTO Programs

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

For this, the Cantor pairing function $J_2 : \mathbb{N}_0^2 \to \mathbb{N}_0$ will be extended to an encoding $J : \mathbb{N}_0^* \to \mathbb{N}_0$ of the set of strings over natural numbers:

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

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

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

Example 5.1. We have $J(1,3) = J_2(J(1),3) + 1 = J_2(J_2(0,1),3) + 1 = J_2(1,3) + 1 = 11 + 1 = 12$. ♦

Proposition 5.2. *The 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 composition of primitive recursive functions:

$$J = \nu \circ J_2(J(\pi_1^{(k+1)}, \ldots, \pi_k^{(k+1)}), \pi_{k+1}^{(k+1)}). \tag{5.4}$$

By induction, J is primitive recursive for strings of length $\leq k+1$ and thus the claim follows.

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

$$u = K_2(n-1) \quad \text{and} \quad v = L_2(n-1). \tag{5.5}$$

Then $J_2(u, v) = J_2(K_2(n-1), L_2(n-1)) = n - 1$. Clearly, $K_2(z) \leq z$ 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, $J_2(J(\boldsymbol{x}), v) + 1 = J_2(J(\boldsymbol{y}), w) + 1$ and so $J_2(J(\boldsymbol{x}), v) = J_2(J(\boldsymbol{y}), w)$. 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, $A = \mathbb{N}_0$; that is, J is bijective. □

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

$$K(n) = K_2(n \dot{-} 1) \quad \text{and} \quad L(n) = L_2(n \dot{-} 1), \quad n \in \mathbb{N}_0. \tag{5.6}$$

Proposition 5.3. *The functions K and J are primitive recursive and have the property*

$$J(K(n), L(n)) = n, \quad n \in \mathbb{N}_0. \tag{5.7}$$

Proof. Since K_2 and L_2 are primitive recursive, it follows that K and L are primitive recursive, too. Let $n \geq 1$. By Proposition 5.2, there are strings $\boldsymbol{x} \in \mathbb{N}_0^*$ 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 by definition of K and L the result follows. □

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

Proposition 5.4. *A string $\boldsymbol{x}$ in $\mathbb{N}_0^*$ is of length k iff k is the smallest number such that $K^k(J(\boldsymbol{x})) = 0$.*

Proof. The empty string satisfies $K^0(J()) = 0$. Let $(\boldsymbol{x}, y)$ be a non-empty string. Put $n = J(\boldsymbol{x}, y)$. Then $n > 0$ and as in the proof of Proposition 5.3, we obtain $K(n) = J(\boldsymbol{x})$ and $L(n) = y$. But by (5.7), we have $K \circ J(K(n), L(n)) = K(n)$. Thus $K \circ J(\boldsymbol{x}, y) = J(\boldsymbol{x})$ 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 it is the empty string. The result follows. □

Let $f : \mathbb{N}_0^* \times \mathbb{N}_0 \to \mathbb{N}_0$ be the function given by

$$f : (\boldsymbol{x}, k) \mapsto K^k(J(\boldsymbol{x})). \tag{5.8}$$

Define the *length function* $\lg : \mathbb{N}_0^* \to \mathbb{N}_0$ by minimalization of the function f, i.e., for all strings $\boldsymbol{x} \in \mathbb{N}_0^*$,

$$\lg(\boldsymbol{x}) = \mu f(\boldsymbol{x}) = \begin{cases} k \text{ if } k \text{ smallest with } f(\boldsymbol{x}, k) = 0, \\ \uparrow \text{ otherwise.} \end{cases} \tag{5.9}$$

Proposition 5.5. *The length function* lg *is recursive.*

Proof. The length function is partial recursive, since it is obtained by minimalization of the primitive recursive function f. Moreover, the proof of Proposition 5.4 shows that for each non-empty string $(\boldsymbol{x}, y)$, the application of the function K to the number $J(\boldsymbol{x}, y)$ yields $J(\boldsymbol{x})$. But each string has finite length and so the repeated application of K to $J(\boldsymbol{x}, y)$ yields 0 after a finite number of steps. Hence, the length function is total. □

Proposition 5.6. *The inverse of the function J is composed of the coordinate functions K and L as follows:*

$$J^{-1}(n) = (K^{k-1}(n), L \circ K^{k-2}(n), \ldots, L \circ K(n), L(n)), \quad n \in \mathbb{N}_0, \tag{5.10}$$

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

Proof. The case $n = 0$ is clear. Let $\boldsymbol{x} \in \mathbb{N}_0^k$ and $y \in \mathbb{N}_0$ such that $J(\boldsymbol{x}, y) = n$. Then $K(n) = J(\boldsymbol{x})$ and $L(n) = y$. Since $K(n) < n$, the induction hypothesis gives $\boldsymbol{x} = J^{-1}(K(n)) = (K^k(n), L \circ K^{k-1}(n), \ldots, L \circ K(n))$. Thus $(\boldsymbol{x}, y) = (K^k(n), L \circ K^{k-1}(n), \ldots, L \circ K(n), L(n))$, 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. In the following, let $\mathcal{P}_{\mathrm{SGOTO}}$ denote the class of SGOTO programs. SGOTO programs will allow a slightly simpler Gödel numbering than GOTO program.

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

$$I(s_l) = \begin{cases} 3 \cdot J(i-1, k) & \text{if } s_l = (l, x_i \leftarrow x_i + 1, k), \\ 3 \cdot J(i-1, k) + 1 & \text{if } s_l = (l, x_i \leftarrow x_i - 1, k), \\ 3 \cdot J(i-1, k, m) + 2 & \text{if } s_l = (l, \texttt{if } x_i = 0, k, m). \end{cases} \tag{5.11}$$

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 also primitive recursive.

Note that the function I allows to identify the lth instruction of an SGOTO program given its Gödel number n. Indeed, the residue of n modulo 3 provides the type of instruction. In turn, the instruction can be decoded by using Proposition 5.6 as follows:

$$s_l = \begin{cases} (l, x_{K(n)+1} \leftarrow x_{K(n)+1} + 1, L(n)) & \text{if } n \equiv 0 \bmod 3, \\ (l, x_{K(n)+1} \leftarrow x_{K(n)+1} - 1, L(n)) & \text{if } n \equiv 1 \bmod 3, \\ (i, \texttt{if } x_{K^2(n)+1} = 0, L(K(n)), L(n)) & \text{if } n \equiv 2 \bmod 3. \end{cases} \tag{5.12}$$

Define a *Gödel numbering* of an SGOTO program $P = s_0; s_1; \ldots; s_q$ as follows:

$$\Gamma(P) = J(I(s_0), I(s_1), \ldots, I(s_q)). \tag{5.13}$$

Proposition 5.7. *The function $\Gamma : \mathcal{P}_{\mathrm{SGOTO}} \to \mathbb{N}_0$ 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 itself primitive recursive. □

For each SGOTO program P, the number $\Gamma(P)$ is called the *Gödel number* of P. The SGOTO program P with Gödel number e is denoted by P_e. The Gödel numbering provides a list of all SGOTO programs

$$P_0, P_1, P_2, \ldots. \tag{5.14}$$

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 (5.7), $K(n) = J(\boldsymbol{x})$ 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 number is established as described in (5.12).

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

$$\phi_e^{(n)} = \|P_e\|_{n,1}. \tag{5.15}$$

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 (5.14) provides a list of all n-ary partial recursive functions:

$$\phi_0^{(n)}, \phi_1^{(n)}, \phi_2^{(n)}, \ldots. \tag{5.16}$$

Note that the list contain 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 kept fixed and then are considered as parameters. The smn theorem does not only tell that the resulting functions are computable but also how to compute an index for them. The basic form applies to functions with two arguments.

Proposition 5.8. *For each 2-ary partial recursive function f, there is an unary primitive recursive function g such that*

$$f(x, \cdot) = \phi_{g(x)}^{(1)}, \quad x \in \mathbb{N}_0. \tag{5.17}$$

Proof. Let $P_e = s_0; s_1; \ldots; s_q$ be an SGOTO program that computes the 2-ary function f. For each number $x \in \mathbb{N}_0$, consider the following SGOTO program Q_x:

$$
\begin{array}{lll}
0 & \texttt{if } x_1 = 0 & 3 \qquad 1 \\
1 & x_2 \leftarrow x_2 + 1 & 2 \\
2 & x_1 \leftarrow x_1 - 1 & 0 \\
3 & x_1 \leftarrow x_1 + 1 & 4 \\
4 & x_1 \leftarrow x_1 + 1 & 5 \\
 & \vdots & \\
2+x & x_1 \leftarrow x_1 + 1 & 3+x \\
 & s'_0 & \\
 & s'_1 & \\
 & \vdots & \\
 & s'_q &
\end{array}
$$

where $P'_e = s_0; s_1; \ldots; s_q$ is the SGOTO program that is derived from P by replacing each 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:

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

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 $g(x) = \Gamma(Q_x)$. This function is primitive recursive by Proposition 5.7. But by definition, $\phi^{(1)}_{g(x)} = \|Q_x\|_{1,1}$ and thus the result follows. □

This assertion can be extended to the socalled smn theorem. The unimaginative name originates from Kleene's notation $s_{m,n}$ for the primitive recursive function playing the key role.

Theorem 5.9. *(***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*

$$\phi^{(m+n)}_e(\boldsymbol{x}, \cdot) = \phi^{(n)}_{s_{m,n}(e,\boldsymbol{x})}, \quad \boldsymbol{x} \in \mathbb{N}_0^m,\ e \in \mathbb{N}_0. \tag{5.18}$$

Proof. The idea is quite similar to that in the proof of the previous proposition. Take an SGOTO program $P_e = s_0; s_1; \ldots; s_q$ that computes $\phi^{(m+n)}_e$. For each input $\boldsymbol{x} \in \mathbb{N}_0^m$, extend the program P_e to an SGOTO program $Q_{e,\boldsymbol{x}}$ providing the following intermediate configurations:

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

It follows that $\|Q_{e,\boldsymbol{x}}\|_{n,1}(\boldsymbol{y}) = \phi^{(m+n)}_e(\boldsymbol{x}, \boldsymbol{y})$ for all $\boldsymbol{x} \in \mathbb{N}_0^m$ and $\boldsymbol{y} \in \mathbb{N}_0^n$.

Consider the function $s_{m,n} : \mathbb{N}_0^{m+1} \to \mathbb{N}_0$ defined by $s_{m,n}(e, \boldsymbol{x}) = \Gamma(Q_{e,\boldsymbol{x}})$. This function is primitive recursive by Proposition 5.7. 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 $\psi_{\text{univ}}^{(n)} : \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ such that

$$\psi_{\text{univ}}^{(n)}(e, \cdot) = \phi_e^{(n)}, \quad e \in \mathbb{N}_0. \tag{5.19}$$

Theorem 5.10. *For each number $n \geq 1$, the universal function $\psi_{\text{univ}}^{(n)}$ exists and is partial recursive.*

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

$$\begin{array}{ccc} e, \xi & \xrightarrow[E]{} & e, \xi' \\ {\scriptstyle G^{-1}}\downarrow & & \uparrow{\scriptstyle G} \\ P_e, \omega & \xrightarrow[s_l]{} & P_e, \omega' \end{array}$$

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

$$E(e, \xi) = (e, \xi'). \tag{5.20}$$

The function E can be described in a higher programming language as given in 5.1. This description shows that the function E is recursive. It follows that the function $\pi_2^{(2)} \circ E : \mathbb{N}_0^2 \to \mathbb{N}_0$ given by

$$(\pi_2^{(2)} \circ E)(e, \xi) = \xi' \tag{5.21}$$

is primitive recursive, too.

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 by

$$E'(e, \xi, t) = (\pi_2^{(2)} \circ E^t)(e, \xi). \tag{5.22}$$

This function is recursive, since it is given by the iteration of a recursive function.

Third, the computation of the SGOTO program P_e terminates if it reaches a non-existent label. That is, if $\lg(e)$ denotes the length of the SGOTO program P_e, termination is reached if and only if

$$G_0(\xi) > \lg(e). \tag{5.23}$$

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

Algorithm 5.1 One-step function.

Require: (e, ξ)
Ensure: (e, ξ')
  if $\xi = 0$ **then**
    $\xi' \leftarrow 0$
  else
    $l \leftarrow G_0(\xi)$ {label, $\xi = 2^l \ldots$}
    $q \leftarrow lg(e)$
    if $l > q$ **then**
      $\xi' \leftarrow \xi$
    else
      $(\sigma_0, \sigma_1, \ldots, \sigma_q) \leftarrow J^{-1}(e)$
      $\xi_0 \leftarrow \xi \div 2^l$ {$\xi = 2^l \xi_0$}
      $j \leftarrow \sigma_l \div 3$ {reveals instruction}
      if $\sigma_l \bmod 3 = 0$ **then**
        $i \leftarrow K_2(j) + 1$ {$s_l = (l, x_i \leftarrow x_i + 1, k)$}
        $k \leftarrow L_2(j)$
        $\xi' \leftarrow \xi_0 \cdot 2^k \cdot p_i$
      end if
      if $\sigma_l \bmod 3 = 1$ **then**
        $i \leftarrow K_2(j) + 1$ {$s_l = (l, x_i \leftarrow x_i - 1, k)$}
        $k \leftarrow L_2(j)$
        if $G_i(\xi) = 0$ **then**
          $\xi' \leftarrow \xi_0 \cdot 2^k$
        else
          $\xi' \leftarrow (\xi_0 \cdot 2^k) \div p_i$
        end if
        if $\sigma_l \bmod 3 = 2$ **then**
          $i \leftarrow K_2(j) + 1$ {$s_l = (l, \texttt{if}\ x_i = 0, k, m)$}
          $k \leftarrow K_2 \circ L_2(j)$
          $m \leftarrow L_2^2(j)$
          if $G_i(\xi) = 0$ **then**
            $\xi' \leftarrow \xi_0 \cdot 2^k$
          else
            $\xi' \leftarrow \xi_0 \cdot 2^m$
          end if
        end if
      end if
    end if
  end if
  return (e, ξ')

$$Z(e,\xi) = \mu_t(\mathrm{csg}(G_0(E'(e,\xi,t)) \dot{-} \lg(e))), \tag{5.24}$$

where the minimalization $\mu = \mu_t$ is subject to the variable t counting the number of steps. This function is partial recursive since the program P_e may not terminate.

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:

$$R(e,\xi) = G_1(E'(e,\xi,Z(e,\xi))). \tag{5.25}$$

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

Summing up, the desired partial recursive function is given as

$$\psi_{\mathrm{univ}}^{(n)}(e,\boldsymbol{x}) = \phi_e^{(n)}(\boldsymbol{x}) = \|P_e\|_{n,1}(\boldsymbol{x}) = R(e, G(0,x_1,\ldots,x_n,0,0,\ldots)), \tag{5.26}$$

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 for universal function 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 a SGOTO program will halt when run with a particular input and if so, a corresponding function is used to obtain 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 for each number $n \geq 1$ the relation $S_n \subseteq \mathbb{N}_0^{n+3}$ defined as

$$(e,\boldsymbol{x},z,t) \in S_n \quad :\Longleftrightarrow \quad \mathrm{csg}[G_0(E'(e,\xi_{\boldsymbol{x}},t)) \dot{-} \lg(e)] = 0 \quad \wedge \quad G_1(E'(e,\xi_{\boldsymbol{x}},t)) = z, \tag{5.27}$$

where $\xi_{\boldsymbol{x}} = (0,x_1,\ldots,x_n,0,0,\ldots)$ is the initial state comprising the input $\boldsymbol{x} = (x_1,\ldots,x_n) \in \mathbb{N}_0^n$. Clearly, $(e,\boldsymbol{x},z,t) \in S_n$ iff the program ϕ_e with input $\boldsymbol{x}$ terminates after t steps with the result z. The set S_n is recursive since the number of steps is explicitly given.

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

$$\mu A(\boldsymbol{x}) = \mu(\mathrm{csg} \circ \chi_A)(\boldsymbol{x}) = \begin{cases} y \text{ if } (\boldsymbol{x},y) \in A \text{ and } (\boldsymbol{x},i) \notin A \text{ for all } 0 \leq i < y, \\ \uparrow \text{ otherwise.} \end{cases} \tag{5.28}$$

We write

$$\mu A(\boldsymbol{x}) = \mu y[(\boldsymbol{x},y) \in A], \quad \boldsymbol{x} \in \mathbb{N}_0^n. \tag{5.29}$$

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

$$E_A = \{\boldsymbol{x} \mid \exists y \in \mathbb{N}_0 : (\boldsymbol{x}, y) \in A\}. \tag{5.30}$$

We write

$$\boldsymbol{x} \in E_A \Longleftrightarrow \exists y[(\boldsymbol{x}, y) \in A], \quad \boldsymbol{x} \in \mathbb{N}_0^n. \tag{5.31}$$

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

$$U_A = \{\boldsymbol{x} \mid \forall y \in \mathbb{N}_0 : (\boldsymbol{x}, y) \in A\}. \tag{5.32}$$

We write

$$\boldsymbol{x} \in U_A \Longleftrightarrow \forall y[(\boldsymbol{x}, y) \in A], \quad \boldsymbol{x} \in \mathbb{N}_0^n. \tag{5.33}$$

Theorem 5.11. *(Kleene) For each number $n \geq 1$ there is a recursive relation $T_n \subseteq \mathbb{N}_0^{n+2}$ such that for all $\boldsymbol{x} \in \mathbb{N}_0^n$:*

$$\boldsymbol{x} \in \operatorname{dom} \phi_e^{(n)} \Longleftrightarrow (e, \boldsymbol{x}) \in \exists y[(e, \boldsymbol{x}, y) \in T_n]. \tag{5.34}$$

If $\boldsymbol{x} \in \operatorname{dom} \phi_e^{(n)}$, there is a recursive function $U : \mathbb{N}_0 \to \mathbb{N}_0$ such that

$$\phi_e^{(n)}(\boldsymbol{x}) = U(\mu y[(e, \boldsymbol{x}, y) \in T_n]). \tag{5.35}$$

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

$$(e, \boldsymbol{x}, y) \in T_n \quad :\Longleftrightarrow \quad (e, \boldsymbol{x}, K_2(y), L_2(y)) \in S_n. \tag{5.36}$$

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 relation S_n is recursive and the functions K_2 and L_2 are primitive recursive, it follows that the relation T_n is recursive, too.

Let $\boldsymbol{x} \in \mathbb{N}_0^n$ lie in the domain of $\phi_e^{(n)}$ and let $\phi_e^{(n)}(\boldsymbol{x}) = z$. Then 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 $(e, \boldsymbol{x}) \in \exists y[(e, \boldsymbol{x}, y) \in T_n]$ and let $y \geq 0$ be a number 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 $(e, \boldsymbol{x}) \in\in \exists y[(e, \boldsymbol{x}, y) \in T_n]$ and so 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 provides the result of computation. □

Kleene's normal form implies that any partial recursive function can be defined 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 with a single `while` loop.

6

Turing Machine

The Turing machine is one of the earliest mathematical models of computation. It was described by the British mathematician Alan Turing (1912-1954) in 1937 as a thought experiment representing a computing machine. Despite its simplicity, a Turing machine is a universal device of computation.

6.1 The Machinery

A Turing machine consists of an infinite tape and a read/write head connected to a control mechanism. The tape is divided into infinitely many cells, each of which containing a symbol from an alphabet named 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 (left and right). At any given time instant, the machine will be in one of a finite number of possible 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 the 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 cell against which it rests, and writes in this cell a symbol from the tape alphabet; it then moves one cell to the left, or one cell to the right, or not at all; after that, the machine enters a new state.

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

- a finite alphabet Σ, called *tape alphabet*, containing a distinguished *blank* symbol b, the subset $\Sigma_I = \Sigma \setminus \{b\}$ is called the *input alphabet*.
- a finite set Q of *states*,
- a partial function $T : Q \times \Sigma \to Q \times \Sigma \times \{L, R, \Lambda\}$, the *state transition function*,
- a *start state* $q_0 \in Q$, and
- a *halt state* $q_F \in Q$,

where $T(q_F, \sigma)$ 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 contents can be considered as a mapping $\tau : \mathbb{Z} \to \Sigma$ which has the value blank almost everywhere; that is, only if finite portion of the tape contains symbols from the input alphabet.

A *configuration* of the Turing machine M consists of the contents 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:

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

where all cells to the left of a_{i_1} and to the right of a_{i_l} contain the blank symbol. This configuration is written as a triple $(a_{i_1} \ldots a_{i_{j-1}}, q, a_{i_j} \ldots a_{i_l})$.

The equation $T(q, a) = (q', a', D)$ 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:

$$(u', q', v') = \begin{cases} (uca', q', v) \text{ if } T(q, a) = (q', a', R), \\ (u, q', ca'v) \text{ if } T(q, a) = (q', a', L), \\ (uc, q', a'v) \text{ if } T(q, a) = (q', a', \Lambda). \end{cases} \tag{6.1}$$

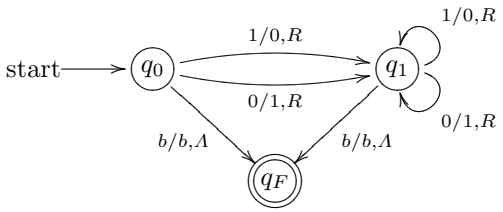
The notation $(u, q, v) \vdash (u', q', v')$ signifies that (u', q', v') is reached from (u, q, v) in one step.

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 must be in the state q_0, with the read/right head against the leftmost cell of the input; that is, the *initial configuration* is given as

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

The machine eventually performs a sequence of transitions as given by the state transition function. If the machine reaches the state q_F, its activity stops and the final output of its computation is read from the symbols remaining on the tape.

Example 6.1. Consider the Turing machine $M = (\Sigma, Q, T, q_0, q_F)$, where $\Sigma = \{0, 1, b\}$, $Q = \{q_0, q_1, q_F\}$, and T is given by the following state diagram:



In such a diagram, the encircled nodes represent the states and an arrow joining state q to state q' and bearing the label $a/a', D$ indicates the transition $T(q, a) = (q', a', D)$. The computation of the machine on the input 0011 is the following:

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

Thus the machine calculates the bitwise complement of the given binary number. ⧫

6.2 Post-Turing Machine

Turing machines specified by state diagrams are hard to follow. Therefore, a program formulation of the Turing machine known as Post-Turing machine invented by Martin Davis (born 1928) will 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 their contents 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 set of instructions which are sequentially executed starting with the first instruction. Each instruction may have a label that must be unique to 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 are introduced that will be used as macros later on. The first macro is `move left to next blank:`

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

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

$$\begin{array}{c} \text{— } b\ 1\ 1\ 1 \ldots 1 \text{ —} \\ \uparrow \end{array}$$

it ends in the configuration

$$\begin{array}{c} \text{— } b\ 1\ 1\ 1 \ldots 1 \text{ —} \\ \uparrow \end{array}$$

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

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

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

$$\begin{array}{c} \text{— } 1\ 1\ 1 \ldots 1\ b \text{ —} \\ \uparrow \end{array}$$

it stops in the configuration

$$\begin{array}{c} \text{— } 1\ 1\ 1 \ldots 1\ b \text{ —} \\ \uparrow \end{array}$$

The macro `write` $b1$ is defined as

```
write b
move right
write 1
move right
```
(6.4)

If this macro begins in the configuration

$$\begin{array}{c} \text{—}\ a_{i_0}\ a_{i_1}\ a_{i_2}\ a_{i_3}\ \text{—} \\ \uparrow \end{array}$$

it halts in the configuration

$$\begin{array}{c} \text{—}\ a_{i_0}\ b\ 1\ a_{i_3}\ \text{—} \\ \uparrow \end{array}$$

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

$$\begin{array}{c} \text{—}\ b\ 1\ 1\ 1\ \ldots\ 1\ b\ 1\ 1\ 1\ \ldots\ 1\ b\ \text{—} \\ \uparrow \end{array}$$

it terminates in the configuration

$$\begin{array}{c} \text{—}\ b\ b\ 1\ 1\ \ldots\ 1\ 1\ 1\ 1\ 1\ \ldots\ 1\ b\ \text{—} \\ \uparrow \end{array}$$

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

$$\begin{array}{c} \text{—}\ a\ b\ 1\ 1\ 1\ \ldots\ 1\ b\ \text{—} \\ \uparrow \end{array}$$

where a is an arbitrary symbol, it halts in the configuration

$$\begin{array}{l} \text{—}\ a\ b\ b\ b\ b\ \ldots\ b\ b\ \text{—} \\ \quad\uparrow \end{array}$$

Finally, the repetition of a statement such as

```
move left
move left
move left
```
(6.8)

will be abbreviated by denoting the statement and the number of repetitions in parenthesis such as

```
move left (3)
```
(6.9)

For instance, the routine `move block right (2)` shifts two consecutive blocks by one cell to the right. It turns the configuration

$$\begin{array}{l} \text{—}\ b\ 1\ 1\ \ldots\ 1\ b\ 1\ 1\ \ldots\ 1\ b\ 1\ 1\ \ldots\ 1\ b\ \text{—} \\ \quad\uparrow \end{array}$$

into the configuration

$$\begin{array}{l} \text{—}\ b\ b\ 1\ \ldots\ 1\ 1\ b\ 1\ \ldots\ 1\ 1\ 1\ 1\ \ldots\ 1\ b\ \text{—} \\ \qquad\qquad\qquad\qquad\qquad\uparrow \end{array}$$

6.3 Turing Computable Functions

It will be shown that that the Turing machine has the same capabilities as the unrestricted register machine. For this, it will be proved that URM computable functions are computable by Post-Turing programs. To this end, 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:

```
start: initiate
       simulate
       clean_up
```
(6.10)

The routine `initiate` presets the program for the computation of the function f. An input $\boldsymbol{c} = (c_1, \ldots, c_k) \in \mathbb{N}_0^k$ of the function f is encoded on the tape in unary format; that is, a number c is represented by a block of $c + 1$ 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:

$$\begin{array}{l} \text{—}\ \overbrace{1\,1\,\ldots\,1}^{c_1+1}\ b\ \overbrace{1\,1\,\ldots\,1}^{c_2+1}\ b\ \ldots\ b\ \overbrace{1\,1\,\ldots\,1}^{c_k+1}\ \text{—} \\ \quad\uparrow \end{array} \tag{6.11}$$

The blocks 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`:

$$
\begin{array}{ll}
\textit{init:} & \texttt{move right to next blank}\ (k) \\
& \texttt{write}\ b1\ (n-k) \\
& \texttt{move left to next blank}\ (n) \\
& \texttt{move right}
\end{array}
\tag{6.12}
$$

If the procedure `initiate` is started in the configuration (6.11), it ends in the configuration

$$
\begin{array}{l}
\overset{c_1+1}{\overbrace{11\ldots1}}\ b\ \overset{c_2+1}{\overbrace{11\ldots1}}\ b\ldots b\ \overset{c_k+1}{\overbrace{11\ldots1}}\ b\ 1\ b\ 1\ b\ldots b\ 1\ b\ 1\ b \\
\uparrow
\end{array}
\tag{6.13}
$$

The tape can be interpreted in a way that it contains the numbers $c_1, \ldots, c_k$ 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 such that consecutive blocks are separated by a single blank. These blocks will then correspond in turn to the used registers.

The GOTO instruction $(l, x_i \leftarrow x_i + 1, l')$ is replaced by the Post-Turing program

$$
\begin{array}{ll}
Al: & \texttt{move right to next blank}\ (i) \\
Bl: & \texttt{write}\ 1 \\
& \texttt{move right} \\
& \texttt{if read}\ b\ \texttt{then goto}\ Cl \\
& \texttt{write}\ b \\
& \texttt{move right to next blank} \\
& \texttt{goto}\ Bl \\
Cl: & \texttt{move left to next blank}\ (n) \\
& \texttt{move right} \\
& \texttt{goto}\ Al'
\end{array}
\tag{6.14}
$$

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 substituted by the program

$$
\begin{array}{ll}
Al: & \texttt{move right to next blank}\ (i) \\
& \texttt{move left}\ (2) \\
& \texttt{if read}\ 1\ \texttt{then goto}\ Bl \\
& \texttt{goto}\ Cl \\
Bl: & \texttt{move left to next blank}\ (i) \\
& \texttt{move right} \\
& \texttt{move block right}\ \ (i-1) \\
& \texttt{write}\ b \\
& \texttt{move left to next blank}\ (i-1) \\
& \texttt{move right} \\
& \texttt{goto}\ Al' \\
Cl: & \texttt{move left to next blank}\ (i-1) \\
& \texttt{move right} \\
& \texttt{goto}\ Al'
\end{array}
\tag{6.15}
$$

This subroutine shortens the ith block by a single 1 if it contains at least two 1's; otherwise, the block is left invariant. Then all blocks to the left are shifted by one cell to the right.

The GOTO instruction $(l, \texttt{if}\ x_i = 0, l', l'')$ is superseded by the program

$$
\begin{array}{ll}
Al: & \texttt{move right to next blank}\ (i) \\
& \texttt{move left}\ (2) \\
& \texttt{if read}\ 1\ \texttt{then goto}\ Bl \\
& \texttt{goto}\ Cl \\
Bl: & \texttt{move left to next blank}\ (i) \\
& \texttt{move right} \\
& \texttt{goto}\ Al'' \\
Cl: & \texttt{move left to next blank}\ (i-1) \\
& \texttt{move right} \\
& \texttt{goto}\ Al'
\end{array}
\tag{6.16}
$$

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 a subroutine 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 be the case if the input $\boldsymbol{c} = (c_1, \ldots, c_k)$ lies in the domain of the function f. In this case, the tape will contain in the first block the unary encoding of the result $f(\boldsymbol{c})$:

$$
\begin{array}{l}
\text{— } b \overbrace{11\ldots1}^{f(c)+1} b \overbrace{11\ldots1}^{y_2+1} b \ldots b \overbrace{11\ldots1}^{y_n+1} b \text{ —} \\
\quad\ \uparrow
\end{array}
\tag{6.17}
$$

Finally, the subroutine `clean_up` will rectify the tape such that upon termination the tape will only contain $f(\boldsymbol{c})$ ones. This is achieved by the code

$$
\begin{array}{ll}
\textit{clean:} & \texttt{move right to next blank}\ (n) \\
& \texttt{move left} \\
& \texttt{erase}\ (n-1) \\
& \texttt{write}\ b \\
& \texttt{move left to next blank} \\
& \texttt{move right}
\end{array}
\tag{6.18}
$$

This subroutine produces the tape:

$$
\begin{array}{c}
\overset{f(c)}{\overbrace{}} \\
\text{—}\ b\ 11\ldots1\ b\ \text{—} \\
\uparrow
\end{array}
\tag{6.19}
$$

This 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 all arguments $(c_1, \ldots, c_k)$ in the domain of f, the program P started with the initial tape (6.11) ends with the final tape (6.19); otherwise, the program P does 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

A Gödel numbering of Post-Turing programs will be provided similar to the Gödel numbering 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, $0 \le j \le q$. A *configuration* of the Post-Turing program P consists of the contents 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:

$$
\begin{array}{c}
\text{—}\ a_{i_1}\ a_{i_2}\ \ldots\ a_{i_v}\ \ldots\ a_{i_t}\ \text{—} \\
\uparrow \\
\sigma_j
\end{array}
\tag{6.20}
$$

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 contents of the tape,
- the position v of the read/write head, with $1 \le v \le t = \mu m(K^m(u) = 0)$, and
- the number j of the next instruction σ_j.

First, define a *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). A 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`,

$$(u', v', j') = \begin{cases} (u, v+1, j+1) & \text{if } v < \mu m(K^m(u) = 0), \\ (J(i_1, \ldots, i_t, 0), v+1, j+1) & \text{otherwise.} \end{cases} \tag{6.23}$$

- 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 GOTO 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; that is, the function

$$E'_P : \mathbb{N}_0^4 \to \mathbb{N}_0^3 : (\boldsymbol{z}, t) \mapsto E_P^t(\boldsymbol{z}). \tag{6.26}$$

But 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:

$$R_P(\boldsymbol{z}) = \begin{cases} E'_P(\boldsymbol{z}, Z_P(\boldsymbol{z})) & \text{if } \boldsymbol{z} \in \mathrm{dom}(Z_P), \\ \uparrow & \text{otherwise.} \end{cases} \tag{6.29}$$

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

$$\alpha_k : \mathbb{N}_0^k \to \mathbb{N}_0^3 : (x_1, \ldots, x_k) \mapsto (J(x_1, \ldots, x_k), 0, 0) \tag{6.30}$$

and

$$\omega_1 : \mathbb{N}_0^3 \to \mathbb{N}_0 : (u, v, j) \mapsto J^{-1}(u'), \tag{6.31}$$

where $u' = J(i_1, \ldots, i_{m-1})$ if $u = J(i_1, \ldots, i_t)$ and m is the smallest index with $i_m = 0$. Both functions are primitive recursive; α_k delivers the initial configuration of the program for a given input and ω_1 reads out the result of the computation. 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. Thus, Theorems 3.13 and 6.2 yield the following result.

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. Prominent examples are the halting problem, the word problem in group theory, and Hilbert's tenth problem.

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 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

$$\chi_A(x) = \operatorname{sgn} \circ \sum_{a \in A} \chi_=(a, x), \quad x \in \mathbb{N}_0.$$

  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 2.37).

♦

Proposition 7.2.

- *If A is a decidable set, the complement of A is decidable.*
- *If A and B are decidable sets, the sets $A \cup B$, $A \cap B$, and $A \setminus B$ are decidable.*

Proof. Plainly, $\chi_{\overline{A}} = \operatorname{csg} \circ \chi_A$, $\chi_{A \cup B} = \operatorname{sgn} \circ (\chi_A + \chi_B)$, $\chi_{A \cap B} = \chi_A \cdot \chi_B$, and $\chi_{A \setminus B} = \chi_A \cdot \chi_{\overline{B}}$. □

There are two ways to prove that a set is undecidable. The first is *diagonalization* similar to Georg Cantor's (1845-1918) famous diagonalization proof, and the second is *reduction* of an already known undecidable problem to the problem under consideration. Here is a prototypical undecidable set.

Proposition 7.3. *The set* $K = \{x \in \mathbb{N}_0 \mid x \in \operatorname{dom} \phi_x\}$ *is undecidable.*

Proof. Assume that the set K would be decidable; i.e., the function χ_K would recursive. Then the function $f : \mathbb{N}_0 \to \mathbb{N}_0$ given by

$$f(x) = \begin{cases} 0 \text{ if } \chi_K(x) = 0, \\ \uparrow \text{ if } \chi_K(x) = 1, \end{cases} \tag{7.1}$$

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 thus $f = \mu g$ is partial recursive. It follows that the function f 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 above proof, the function $h : \mathbb{N}_0 \to \mathbb{N}_0$ defined by

$$h(x) = \begin{cases} 0 \text{ if } \chi_K(x) = 1, \\ \uparrow \text{ if } \chi_K(x) = 0, \end{cases} \tag{7.2}$$

is partial recursive. To see this, observe that for each $x \in \mathbb{N}_0$, $h(x) = 0 \cdot \phi_x(x) = 0 \cdot \psi^{(1)}_{\text{univ}}(x, x)$. Moreover, the function $h' : \mathbb{N}_0 \to \mathbb{N}_0$ given by

$$h'(x) = \begin{cases} x \text{ if } \chi_K(x) = 1, \\ \uparrow \text{ if } \chi_K(x) = 0, \end{cases} \tag{7.3}$$

is partial recursive. Indeed, for each $x \in \mathbb{N}_0$, $h'(x) = x \cdot \operatorname{sgn}(\phi_x(x) + 1) = x \cdot \phi^{(1)}_{\text{univ}}((x, x) + 1)$. It is interesting to note that the domain and range of the function h' are undecidable sets, since $\operatorname{dom} h' = \operatorname{ran} h' = K$.

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 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. Suppose the characteristic function χ_H of H would be recursive. Then also the function $g : \mathbb{N}_0 \to \mathbb{N}_0 : x \mapsto \chi_H(x, x)$ would be recursive. But this function equals the characteristic function of K, i.e., for all $x \in \mathbb{N}_0$, $\chi_K(x) = \chi_H(x, x)$, and the function χ_K is not recursive. This provides a contradiction. □

The halting problem has been proved by reduction. More generally, 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 $f : \mathbb{N}_0^k \to \mathbb{N}_0^l$ such that

$$\boldsymbol{x} \in A \quad \Longleftrightarrow \quad f(\boldsymbol{x}) \in B, \quad \boldsymbol{x} \in \mathbb{N}_0^k. \tag{7.4}$$

This assertion is equivalent to

$$\chi_A(\boldsymbol{x}) = \chi_B(f(\boldsymbol{x})), \quad \boldsymbol{x} \in \mathbb{N}_0^k. \tag{7.5}$$

This means that if B is decidable, A is also decidable; or by contraposition, if A is undecidable, B is also undecidable. For instance, in the proof of the halting problem, the set K has been reduced to the set H by the function $f : x \mapsto (x, x)$.

The next undecidability result makes use of the smn theorem.

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

$$f(x, y) = \begin{cases} 0 \text{ if } \chi_K(x) = 1, \\ \uparrow \text{ if } \chi_K(x) = 0. \end{cases} \tag{7.6}$$

This function is partial recursive, since it can be written as $f = h \circ \pi_1^{(2)}$, where h is the function given in (7.2). By the smn theorem, there is a recursive function g such that $f(x, y) = \phi_{g(x)}(y)$ for all $x, y \in \mathbb{N}_0$.

Suppose $x \in K$. Then $f(x, y) = 0$ for all $y \in \mathbb{N}_0$ and so $\phi_{g(x)}(y) = c_0^{(1)}(y)$ for all $y \in \mathbb{N}_0$, which in turn gives $g(x) \in C$.

Suppose $x \notin K$. Then $f(x, y)$ is undefined for all $y \in \mathbb{N}_0$ and hence $\phi_{g(x)}(y)$ is undefined for all $y \in \mathbb{N}_0$, which means that $g(x) \notin C$.

Thus the recursive function g provides a reduction of the set K to the set C. But K is undecidable and so C is undecidable, too. □

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. Consider the function $f : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by

$$f(x, y) = \begin{cases} y \text{ if } \chi_K(x) = 1, \\ \uparrow \text{ if } \chi_K(x) = 0. \end{cases} \tag{7.7}$$

This function is partial recursive, since it can be written as $f(x, y) = y \cdot \operatorname{sgn}(\phi_x(x) + 1) = y \cdot \operatorname{sgn}(\psi_{\text{univ}}^{(1)}(x, x) + 1)$. By the smn theorem, there is a recursive function $g : \mathbb{N}_0 \to \mathbb{N}_0$ such that

$$f(x, y) = \phi_{g(x)}(y), \quad x, y \in \mathbb{N}_0. \tag{7.8}$$

Suppose $x \in K$. Then $f(x, y) = y$ for all $y \in \mathbb{N}_0$ and thus $\operatorname{dom} \phi_{g(x)} = \operatorname{ran} \phi_{g(x)} = \mathbb{N}_0$.

Suppose $x \notin K$. Then $f(x, y)$ is undefined for all $y \in \mathbb{N}_0$ and so $\operatorname{dom} \phi_{g(x)} = \operatorname{ran} \phi_{g(x)} = \emptyset$.

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 $T = \{x \in \mathbb{N}_0 \mid \phi_x \text{ is total}\}$ is undecidable.*

Proof. Assume that T would be decidable, i.e., χ_T would be recursive. Consider the function $f : \mathbb{N}_0 \to \mathbb{N}_0$ defined as

$$f(x) = \begin{cases} \phi_x(x) + 1 & \text{if } \chi_T(x) = 1, \\ 0 & \text{if } \chi_T(x) = 0. \end{cases} \tag{7.9}$$

Since $f(x) = \chi_T(x) \cdot (\phi_x(x) + 1) = \chi_T(x) \cdot (\psi^{(1)}_{\text{univ}}(x, x) + 1)$, the function f is recursive. Thus there is an index $e \in T$ such that $f = \phi_e$. But then $\phi_e(e) = f(e) = \phi_e(e) + 1$ contradicting the assumption. □

The undecidability results established so far have some practical implications, which are briefly summarized using the formalism of SGOTO programs:

- The problem whether an SGOTO program P_x halts with input y or not is undecidable.
- The problem whether an SGOTO program P_x computes a specific function (here $c^{(1)}_0$) or not is undecidable.
- The problem whether two SGOTO programs P_x and P_y are semantically equivalent (input-output behaviour) or not is undecidable.
- The problem whether an SGOTO program P_x halts for a specific input a or not is undecidable.
- The problem whether an SGOTO program P_x always halts or not is undecidable.

These undecidability results generally refer to the input-output behaviour or the semantics of SGOTO programs. The following result of 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 an algorithm computes a partial recursive function with that property. 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, 1953) If $\mathcal{A}$ is a proper subclass of unary partial recursive functions, the corresponding index set*

$$\operatorname{prog}(\mathcal{A}) = \{x \in \mathbb{N}_0 \mid \phi_x \in \mathcal{A}\} \tag{7.10}$$

is undecidable.

By Example 7.1, if $\mathcal{A}$ is a class of unary partial recursive functions, $\operatorname{prog}(\mathcal{A})$ is decidable if and only if $\mathcal{A}$ is either empty or consists of all partial recursive functions.

Proof. By Proposition 7.2, if a set is decidable, also its complement is 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:

$$h(x,y) = \begin{cases} f(y) \text{ if } \chi_K(x) = 1, \\ \uparrow \quad \text{if } \chi_K(x) = 0. \end{cases}$$

Since $h(x,y) = f(y) \cdot \operatorname{sgn}(\phi_x(x)+1) = f(y) \cdot \operatorname{sgn}(\psi^{(1)}_{\text{univ}}(x,x)+1)$, the function h is partial recursive. Therefore by the smn theorem, there is an unary recursive function g such that

$$h(x,y) = \phi_{g(x)}(y), \quad x,y \in \mathbb{N}_0$$

Suppose $x \in K$. Then $h(x,y) = f(y)$ for all $y \in \mathbb{N}_0$ and thus $\phi_{g(x)} = f$. It follows that $g(x) \in \text{prog}(\mathcal{A})$. Suppose $x \notin K$. Then $h(x,y) = \uparrow$ for all $y \in \mathbb{N}_0$ and so $\phi_{g(x)} = f_\uparrow$. Hence, $g(x) \notin \text{prog}(\mathcal{A})$.

Therefore, the recursive function g reduces the set K to the set $\text{prog}(\mathcal{A})$. Since the set K is undecidable, the result follows. □

Note that an index for a function is essentially the same as a program computing it. Thus Rice's theorem informally states that it is impossible to decide anything about a function from its program. Practically, this means that there exists no systematic procedure telling whether or not an arbitrary program is correct.

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 generally, 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

$$f(\boldsymbol{x}) = \begin{cases} 1 \text{ if } \boldsymbol{x} \in A, \\ \uparrow \text{ otherwise}, \end{cases} \tag{7.11}$$

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 (7.11) 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 (7.11). Hence the set A is semidecidable. □

Example 7.11. The prototypical set K is semidecidable as it is the domain of the partial recursive function in (7.2). ♦

A program for the function f given in (7.11) 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 $\boldsymbol{x}$ halts giving a positive answer. Otherwise, the program does eventually not halt 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

$$g(\boldsymbol{x}, y) = (\mathrm{csg} \circ \chi_A)(\boldsymbol{x}) = \begin{cases} 0 \text{ if } \boldsymbol{x} \in A, \\ 1 \text{ otherwise,} \end{cases} \tag{7.12}$$

is recursive. It follows that the function $f = \mu g$ is partial recursive. But $\mu g(\boldsymbol{x}) = y$ if $g(\boldsymbol{x}, y) = 0$ and $g(\boldsymbol{x}, i) \neq 0$ for all $0 \leq i < y$, and $\mu g(\boldsymbol{x}) = \uparrow$ otherwise. Thus $\mu g(\boldsymbol{x}) = 0$ if $\boldsymbol{x} \in A$ and $\mu g(\boldsymbol{x}) = \uparrow$ otherwise. Hence, $\mathrm{dom}\, f = A$ as required. □

Proposition 7.13. *The set H corresponding to the halting problem is semidecidable.*

Proof. The universal function $\psi_{\mathrm{univ}}^{(1)}$ has the property

$$\psi_{\mathrm{univ}}^{(1)}(x, y) = \begin{cases} \phi_x(y) \text{ if } y \in \mathrm{dom}\, \phi_x, \\ \uparrow \quad \text{otherwise.} \end{cases} \tag{7.13}$$

It follows that $H = \mathrm{dom}\, \psi_{\mathrm{univ}}^{(1)}$ as required. □

Proposition 7.14. *Let A be a subset of $\mathbb{N}_0^k$ that is reducible to a semidecidable subset B of $\mathbb{N}_0^l$. Then A is also semidecidable.*

Proof. By definition, there is a recursive function $f : \mathbb{N}_0^k \to \mathbb{N}_0^l$ such that $\boldsymbol{x} \in A$ iff $f(\boldsymbol{x}) \in B$. Moreover, there is a partial recursive function $g : \mathbb{N}_0^l \to \mathbb{N}_0$ such that $B = \mathrm{dom}\, g$. Thus the composite function $g \circ f : \mathbb{N}_0^k \to \mathbb{N}_0$ is partial recursive. Then 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, $\mathrm{dom}\, g \circ f = A$ as claimed. □

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 subset A of $\mathbb{N}_0^k$ is semidecidable if and only if there is a decidable subset B of $\mathbb{N}_0^{k+1}$ 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

$$f(\boldsymbol{x}) = \mu(\mathrm{csg} \circ \chi_B)(\boldsymbol{x}) = \begin{cases} 0 \text{ if } (\boldsymbol{x}, y) \in B \text{ for some } y \in \mathbb{N}_0, \\ \uparrow \text{ otherwise.} \end{cases} \tag{7.14}$$

Clearly, $\mathrm{dom}\, f = A$.

Conversely, let A be a semidecidable subset of $\mathbb{N}_0^k$. Then there is an index e such that $\mathrm{dom}\, \phi_e^{(k)} = A$. By Kleene's normal form theorem, an element $\boldsymbol{x} \in \mathbb{N}_0^k$ satisfies $\boldsymbol{x} \in A$ if and only if $\boldsymbol{x} \in \exists y[(e, \boldsymbol{x}, y) \in T_k]$ if e is considered to be fixed. That is, $A = \exists y[(e, \boldsymbol{x}, y) \in T_k]$, where T_k is a (primitive) recursive set. □

The next result shows that the class of semidecidable sets is closed under unbounded existential minimalization.

Proposition 7.16. *If B is a semidecidable subset of $\mathbb{N}_0^{k+1}$, the set $A = \exists y[(\boldsymbol{x}, y) \in B]$ is a semidecidable subset of $\mathbb{N}_0^k$.*

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 $B = \exists z[(\boldsymbol{x}, y, z) \in C]$. 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 $(\boldsymbol{x}, K_2(u), L_2(u)) \in C$; that is, $A = \exists u[(\boldsymbol{x}, K_2(u), L_2(u)) \in C]$. Thus by Proposition 7.15, the set A is semidecidable. □

In opposition to this result, the class of decidable sets is not closed under unbounded existential minimalization. For instance, by Kleene's normal form theorem, the prototypic set K corresponds to the set $\exists y[(x, x, y) \in T_1]$ where T_1 is a (primitive) recursive set.

Another useful connection between decidable and semidecidable sets is the following.

Proposition 7.17. *A subset A of $\mathbb{N}_0^k$ is decidable if and only if A and $\overline{A}$ are semidecidable.*

Proof. If A is decidable, by Proposition 7.2, the set $\overline{A}$ is also decidable. Conversely, if A and $\overline{A}$ are semidecidable, a decision procedure for A can be established by using the partial decision procedures for A and $\overline{A}$ which are simultaneously considered. One of which will provide a positive answer in a finite number of steps giving the answer for the decision procedure for A. □

Example 7.18. The complement of the halting problem, $\overline{H} = \{(x, y) \in \mathbb{N}_0^2 \mid y \notin \operatorname{dom} \phi_x\}$, is not semidecidable, since the set H is semidecidable but not decidable. ♦

Define the *graph* of a partial function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ as the set

$$\operatorname{graph}(f) = \{(\boldsymbol{x}, y) \in \mathbb{N}_0^{k+1} \mid f(\boldsymbol{x}) = y\}. \tag{7.15}$$

Proposition 7.19. *A function $f : \mathbb{N}_0^k \to \mathbb{N}_0$ is partial recursive if and only if the graph of f is semidecidable.*

Proof. Let f be partially recursive. Then there is an index e for f, i.e., $f = \phi_e^{(k)}$. The relation S_K used to derive Kleene's normal form shows that $f(\boldsymbol{x}) = y$ is equivalent to $(\boldsymbol{x}, y) \in \exists t[(e, \boldsymbol{x}, y, t) \in S_k$. Thus the set $\operatorname{graph}(f)$ is obtained form the decidable set S_k, where e is kept fixed, by existential quantification and hence is semidecidable.

Conversely, let $\operatorname{graph}(f)$ be semidecidable. Then there is a decidable relation $A \subseteq \mathbb{N}_0^{k+2}$ such that

$$\operatorname{graph}(f) = \exists z[(\boldsymbol{x}, y, z) \in A].$$

To compute the function f, take an argument $\boldsymbol{x} \in \mathbb{N}_0^k$ and systematically search for a pair $(y, z) \in \mathbb{N}_0^2$ such that $(\boldsymbol{x}, y, z) \in A$. If such a pair exists, put $f(\boldsymbol{x}) = y$. Otherwise, set $f(\boldsymbol{x}) = \uparrow$. □

7.3 Recursively Enumerable Sets

The terminology for decidable and semidecidable sets of natural numbers 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* 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. To prove the first part, note that the function $\chi_{J_k(A)}$ is recursive iff $\chi_A \circ J_k^{-1}$ is recursive. To prove the second part, take a k-ary partial recursive function f such that $\mathrm{dom} f = A$. Then function $\chi_{J_k(A)}$ is recursive enumerable iff $f \circ J_k^{-1}$ is recursive enumerable. □

First, 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 $\chi_{\overline{A}} = \mathrm{csg} \circ \chi_A$, $\chi_{A \cup B} = \mathrm{sgn} \circ (\chi_A + \chi_B)$, $\chi_{A \cap B} = \chi_A \cdot \chi_B$, and $\chi_{A \setminus B} = \chi_A \cdot \chi_{\overline{B}}$ are also recursive. □

The Gödel numbering of GOTO programs yields an enumeration of all unary partial recursive functions

$$\phi_0, \phi_1, \phi_2, \ldots \tag{7.16}$$

By taking the domains of these functions, i.e., $D_e = \mathrm{dom}\, \phi_e$, this list provides an enumeration of all recursively enumerable sets

$$D_0, D_1, D_2, \ldots \tag{7.17}$$

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 an unary recursive function f such that $A = \mathrm{ran}\, f$.*
- *There is a k-ary partial recursive function g such that $A = \mathrm{ran}\, g$.*

Proof. • First, let A be a nonempty recursively enumerable set. Take $a \in A$ and let e be an index for A. By using Kleene's normal form theorem, define the unary function

$$f : x \mapsto \begin{cases} K_2(x) & \text{if } (e, K_2(x), L_2(x)) \in T_1, \\ a & \text{otherwise.} \end{cases} \tag{7.18}$$

Since $A = \mathrm{dom}\, \phi_e$ it follows that $A = \mathrm{ran}\, f$ as required.

- Second, let $A = \emptyset$ or $A = \mathrm{ran}\, f$ for some unary 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 $A = \mathrm{ran}\, g$ as required.
- Third, let g be a k-ary partial recursive function such that $A = \mathrm{ran}\, g$. 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 nonempty set of natural numbers A is recursively enumerable if and only if there is an unary recursive function f that allows to enumerate the elements of A, i.e.,

$$A = \{f(0), f(1), f(2), \ldots\}. \tag{7.19}$$

Such a function f is called an *enumerator* for A. By the Kleene normal form theorem, the enumerator f is actually 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 unary partial recursive functions such that $A = \operatorname{dom} f$ 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 nonempty sets, and let f and g be unary recursive functions such that $A = \operatorname{ran} f)$ and $B = \operatorname{ran} g$. Define the unary function h as

$$h : x \mapsto \begin{cases} f(\lfloor x/2 \rfloor) \text{ if } x \text{ is even,} \\ g(\lfloor x/2 \rfloor) \text{ otherwise.} \end{cases} \tag{7.20}$$

That is, $h(0) = f(0)$, $h(1) = g(0)$, $h(2) = f(1)$, $h(3) = g(1)$ 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 there is a strictly monotonous enumerator for A.*

Proof. Let A be an infinite and recursive set. Define the unary function f by minimalization and primitive recursion as follows:

$$f(0) = \mu y[y \in A], \tag{7.21}$$

$$f(n+1) = \mu y[y \in A \wedge y > f(n)]. \tag{7.22}$$

This function is recursive, strictly monotonous and satisfies $\operatorname{ran} f = A$.

Conversely, let $A = \operatorname{ran} f$ be an infinite set and let f be a strictly monotonous unary recursive function. Then $f(n) = y$ implies $y \geq n$ and thus

$$y \in A \quad \Longleftrightarrow \quad \exists n[n \leq y \wedge f(n) = y]. \tag{7.23}$$

The relation on the right-hand side is decidable and so A is recursive. □

7.4 The Theorem of Rice-Shapiro

A classical result in computability theory is the Rice-Shapiro theorem conjectured by Henry Gordan Rice and proved by Norman Shapiro (born 1932). It gives an extensional characterization of classes of recursively enumerable sets whose index set is recursively enumerable.

For this, an unary function g is called an *extension* of an unary 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 unary functions with smallest element given by the nowhere-defined function. The maximal elements are the unary recursive functions.

An unary function f is called *finite* if its domain is finite. Each finite function f is partial recursive, since

$$f(x) = \sum_{a \in \operatorname{dom} f} \operatorname{csg}(|x - a|) f(a), \quad x \in \mathbb{N}_0. \tag{7.24}$$

Theorem 7.25. *(Rice-Shapiro) Let $\mathcal{A}$ be a class of unary partial recursive functions such that the corresponding index set* $\operatorname{prog}(\mathcal{A}) = \{x \in \mathbb{N}_0 \mid \phi_x \in \mathcal{A}\}$ *is recursively enumerable. Then an unary 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 extended by f does lie in $\mathcal{A}$. Take the prototypical 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

$$g : (z, t) \mapsto \begin{cases} \uparrow & \text{if } P_e \text{ computes } \phi_e(z) \text{ in} \leq t \text{ steps,} \\ f(t) & \text{otherwise.} \end{cases} \tag{7.25}$$

The function g is obviously partial recursive. By the smn theorem, there is an unary recursive function s such that

$$g(z, t) = \phi_{s(z)}(t), \quad t, z \in \mathbb{N}_0. \tag{7.26}$$

Moreover, $\phi_{s(z)} \subseteq f$ for each $z \in \mathbb{N}_0$.

Consider two cases:

- If $z \in K$, the program $P_e(z)$ halts, say, after t_0 steps. Then

$$\phi_{s(z)}(t) = \begin{cases} \uparrow & \text{if } t_0 \leq t, \\ f(t) & \text{otherwise.} \end{cases} \tag{7.27}$$

  and so $\phi_{s(z)}$ is finite. 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 $\overline{K}$ onto the set $\operatorname{prog}(\mathcal{A})$ contradicting the assumption that $\operatorname{prog}(\mathcal{A})$ is recursively enumerable.

Conversely, let f be an unary partial recursive function that does not belong to $\mathcal{A}$ and let g be a finite function in $\mathcal{A}$ such that $g \subseteq f$. Define the function

$$h : (z, t) \mapsto \begin{cases} f(t) & \text{if } t \in \operatorname{dom} g \text{ or } z \in K, \\ \uparrow & \text{otherwise.} \end{cases} \tag{7.28}$$

The function h is obviously partial recursive. By the smn theorem, there is an unary recursive function s such that

$$h(z, t) = \phi_{s(z)}(t), \quad t, z \in \mathbb{N}_0. \tag{7.29}$$

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 $\overline{K}$ onto the set $\operatorname{prog}(\mathcal{A})$ contradicting the hypothesis that $\operatorname{prog}(\mathcal{A})$ is recursively enumerable. □

Corollary 7.26. *Let* $\operatorname{prog}(\mathcal{A}) = \{x \in \mathbb{N}_0 \mid \phi_x \in \mathcal{A}\}$ *be recursively enumerable. Then any extension of a function in* $\mathcal{A}$ *is itself in* $\mathcal{A}$.

Proof. Let h extend f, with $f \in \mathcal{A}$. Then there is a finite function g such that f extends g. But then also h extends g and it follows by the theorem of Rice-Shapiro that h lies in $\mathcal{A}$. □

Corollary 7.27. *Let* $\operatorname{prog}(\mathcal{A}) = \{x \in \mathbb{N}_0 \mid \phi_x \in \mathcal{A}\}$ *be recursively enumerable. If the nowhere-defined function is in* $\mathcal{A}$*, all unary partial recursive functions are in* $\mathcal{A}$.

Proof. Each unary computable function extends the nowhere-defined function and so by the theorem of Rice-Shaprio the result follows. □

Claim that the theorem of Rice is a consequence of the theorem of Rice-Shapiro. Indeed, let $\mathcal{A}$ be a set of unary partial recursive functions. If $\operatorname{prog}(\mathcal{A})$ is decidable, both the set and its complement are recursively enumerable. Thus $\mathcal{A}$ or $\overline{\mathcal{A}}$ must contain the nowhere-defined function and hence consists of all partial recursive functions. It follows that the set $\mathcal{A}$ is trivial and the claim is proved.

Example 7.28. • The set $\{x \mid \phi_x \text{ finite}\}$ is not recursively enumerable by Corollary 7.26.
- The set $\{x \mid \phi_x \text{ has infinite range}\}$ is not recursively enumerable by the theorem of Rice-Shapiro.

⧫

7.5 Diophantine Sets

David Hilbert (1982-1943) provided a list of 23 mathematical problems at the International Mathematical Congress in Paris (1900). The tenth problem on the list has posed the problem to decide for an arbitrary polynomial with integer coefficients whether or not it has integer solutions. 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*

$$p(X_1, X_2, \ldots, X_n) = 0 \tag{7.30}$$

asking for *integer* solutions of this equation. Note that a diophantine equation always has complex-valued solutions but may not have any integer solution such as $X^2 - 1 = 0$.

Example 7.29. 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. The solutions can be found by applying the extended Euclidean algorithm. 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 $\mathbb{Z}[X_1, \ldots, X_n, Y_1, \ldots, Y_m]$. The n-ary relation

$$\{(x_1, \ldots, x_n) \in \mathbb{N}_0^n \mid p(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0 \text{ for some } y_1, \ldots, y_m \in \mathbb{N}_0\} \tag{7.31}$$

is called a *diophantine set* and is denoted by $\exists y_1 \ldots \exists y_m[p(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0]$. It follows that each diophantine set arises from a decidable set by existential quantification. Thus Proposition 7.15 provides the following result.

Proposition 7.30. *Each diophantine set is semidecidable.*

Example 7.31. • The set of positive integers is diophantine, being $\{x \mid \exists y[x = y + 1]\}$.
- The predicates $\leq$ and $<$ are diophantine, since $x \leq y$ iff $\exists z[y = x+z]$, and $x < y$ iff $\exists z[y = x+z+1]$.
- 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 was proved by Yuri Matiyasevic in 1970.

Theorem 7.32. *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

$$A = \exists y_1 \ldots \exists y_m[p(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0]. \tag{7.32}$$

The negative solution to 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:

$$\begin{aligned}&(a^2 + b^2 + c^2 + d^2)(t^2 + u^2 + v^2 + w^2) = \\ &(at + bu + cv + dw)^2 + (au - bt + cw - dv)^2 + (av - ct - bw + du)^2 + (aw - dt + bv - cu)^2.\end{aligned} \tag{7.33}$$

This equation implies that the set of numbers which are the sum of four squares is closed under multiplication.

Theorem 7.33. *(Lagrange, 1770) 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 iff 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.p 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 interates 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

$$\begin{aligned} q(T_1,T_2,\ldots,T_n,U_1,U_2,\ldots,U_n,V_1,V_2,\ldots,V_n,W_1,W_2,\ldots,W_n) = \\ p(T_1^2+U_1^2+V_1^2+W_1^2,T_2^2+U_2^2+V_2^2+W_2^2,\ldots,T_n^2+U_n^2+V_n^2+W_n^2). \end{aligned} \tag{7.34}$$

Let $\boldsymbol{x} = (x_1,\ldots,x_n) \in \mathbb{N}_0^n$ be a solution of the diophantine equation

$$p(X_1,\ldots,X_n) = 0 \tag{7.35}$$

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,

$$x_i = t_i^2+u_i^2+v_i^2+w_i^2, \quad 1 \leq i \leq n. \tag{7.36}$$

Then $(\boldsymbol{t},\boldsymbol{u},\boldsymbol{v},\boldsymbol{w}) \in \mathbb{Z}^{4n}$ is a solution of the diophantine equation

$$q(T_1,\ldots,T_n,U_1,\ldots,U_n,V_1,\ldots,V_n,W_1,\ldots,W_n) = 0. \tag{7.37}$$

The converse assertion also holds. It follows that if there is an effective procedure to decide whether diophantine equation (7.37) has *integer* solutions, there is an effective procedure to decide whether diophantine equation (7.35) has *nonnegative integer* solutions.

Theorem 7.34. *Hilbert's tenth problem is undecidable.*

Proof. The set K is recursively enumerable and so there is a polynomial p in $\mathbb{Z}[X,Y_1,\ldots,Y_m]$ such that

$$K = \exists y_1 \ldots \exists y_m[p(x,y_1,\ldots,y_m) = 0]. \tag{7.38}$$

Suppose there is an effective procedure to decide whether or not a diophantine equation has nonnegative integer solutions. Then the question whether a number $x \in \mathbb{N}_0$ lies in K or not can be decided by finding a nonnegative integer solution of the equation $p(x,Y_1,\ldots,Y_m) = 0$. However, this contradicts the undecidability of K. □

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 problems.

8.1 Semi-Thue Systems

The word problem for a set is the algorithmic problem of deciding whether two given representatives represent 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 a *semi-Thue system*, is a pair (Σ, R) where Σ is an alphabet and R is a binary relation on nonempty 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 formally, the *one-step rewriting relation* $\to_R$ induced by R on Σ^* is a binary relation on Σ^* such that for any strings s and t in Σ^*,

$$s \to_R t \quad :\Longleftrightarrow \quad s = xuy,\ t = xvy,\ u \to v \text{ for some } x, y, u, v \in \Sigma^*. \tag{8.1}$$

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 are 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), where $\Sigma = \{a, b\}$ and $R = \{(ab, bb), (ab, a), (b, aba)\}$. The derivation $abb \to_R ab \to_R bb \to_R baba \to_R bbba$ shows that $abb \overset{*}{\to}_R bbba$. ♦

The *word problem* for semi-Thue systems can be stated as follows: Given a semi-Thue system (Σ, R) and two strings $s, t \in \Sigma^+$, can the strings s be transformed into the strings t by applying the rules from R; that is, $s \to_R^* t$?

This problem is undecidable. To see this, the halting problem for SGOTO-2 programs is reduced to this problem. For this, let $P = s_0; s_1; \ldots; s_{n-1}$ be an SGOTO-2 program consisting of n instructions such that the label n is the only one that does not address an instruction (and thus leads to 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, x is the content of the first register, and y is the content of the second register. These numbers are encoded in unary format

$$\overline{x} = \overbrace{LL\ldots L}^{x} \quad \text{and} \quad \overline{y} = \overbrace{LL\ldots L}^{y}. \tag{8.2}$$

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 $\Sigma = \{a, b, 0, 1, 2, \ldots, n, L\}$.

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)
$(j, x_1 \leftarrow x_1 - 1, k)$	$(Lj, k),\ (aj, ak)$
$(j, x_2 \leftarrow x_2 - 1, k)$	$(jL, k),\ (jb, kb)$
$(j, \texttt{if}\ \ x_1 = 0, k, l)$	$(Lj, Ll),\ (aj, ak)$
$(j, \texttt{if}\ \ x_2 = 0, k, l)$	$(jL, lL),\ (jb, kb)$

Moreover, the semi-Thue system contains two cleanup rewriting rules

$$(Ln, n) \quad \text{and} \quad (anL, an). \tag{8.4}$$

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 function

$$\|P\|_{2,1}(x, y) = \begin{cases} 0 \text{ if } x \geq y, \\ \uparrow \text{ otherwise.} \end{cases} \tag{8.5}$$

The corresponding semi-Thue system over the alphabet $\Sigma = \{a, b, 0, 1, 2, 3, 4, 5, L\}$ consists of the rewriting rules

$$(L0, L1), (a0, a3), (L1, 2), (a1, a2), (2L, 0), (2b, 0b), (3L, 4L), (3b, 5b), (4, 3L), (L5, 5), (a5L, a5). \quad (8.6)$$

Here is a sample computation, where the registers initially hold the values $x_1 = 3$ and $x_2 = 2$:

SGOTO-2 program	semi-Thue system
$(0, 3, 2)$	$aLLL0LLb$
$(1, 3, 2)$	$aLLL1LLb$
$(2, 2, 2)$	$aLL2LLb$
$(0, 2, 1)$	$aLL0Lb$
$(1, 2, 1)$	$aLL1Lb$
$(2, 1, 1)$	$aL2Lb$
$(0, 1, 0)$	$aL0b$
$(1, 1, 0)$	$aL1b$
$(2, 0, 0)$	$a2b$
$(0, 0, 0)$	$a0b$
$(3, 0, 0)$	$a3b$
$(5, 0, 0)$	$a5b$

♦

The contruction immediately yields the following result.

Proposition 8.3. *If (j, x, y) is a configuration of the SGOTO-2 program P with $j < n$, the semi-Thue system (Σ, R_P) provides the one-step rewriting rule*

$$a\overline{x}j\overline{y}b \;\rightarrow_{R_P}\; a\overline{u}k\overline{v}b, \quad (8.7)$$

where (k, u, v) is the successor configuration of (j, x, y). There is no other one-step rewriting rule in the semi-Thue system applicable to $a\overline{x}j\overline{y}b$.

The iterated use of this statement leads to the following result.

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*

$$a\overline{x}j\overline{y}b \;\stackrel{*}{\rightarrow}_{R_P}\; a\overline{u}k\overline{v}b. \quad (8.8)$$

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 cleanup rules (8.4) as follows:

$$a\overline{x}n\overline{y}b \;\stackrel{*}{\rightarrow}_{R_P}\; anb. \quad (8.9)$$

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,*

$$a\overline{x}0\overline{y}b \;\stackrel{*}{\rightarrow}_{R_P}\; anb. \quad (8.10)$$

This provides 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 the following result.

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 for Thue systems 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

$$R^{(s)} = R \cup R^{-1}, \tag{8.11}$$

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 hold in the corresponding semi-Thue system (Σ, R).

Example 8.7. Consider the semi-Thue system (Σ, R), where $\Sigma = \{a, b\}$ 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 $|u| > |v|$. For instance, $aabb \to_R abb \to_R bb$. On the other hand, $aabb \to_{R^{(s)}} abb \to_{R^{(s)}} abab$, but neither $aabb \overset{*}{\to}_R abab$ nor $abab \overset{*}{\to}_R aabb$. ♦

Theorem 8.8. (Post's Lemma) *Let P be an SGOTO-2 program with n instructions, 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,*

$$a\overline{x}j\overline{y}b \overset{*}{\to}_{R_P} anb \quad \Longleftrightarrow \quad a\overline{x}j\overline{y}b \overset{*}{\to}_{R_P^{(s)}} anb. \tag{8.12}$$

Proof. The direction from left-to-right follows directly from the definitions. 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

$$s_0 = a\overline{x}j\overline{y}b \to_{R_P^{(s)}} s_1 \to_{R_P^{(s)}} \ldots \to_{R_P^{(s)}} s_q = anb, \tag{8.13}$$

where it may be assumed that the length $q+1$ 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 (8.13) does not lie in the semi-Thue system. That is, there is 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. But there is no rewriting rule applicable to $s_q = anb$ and thus $p + 1 < q$. This yields the following situation:

$$s_p \leftarrow_{R_P} s_{p+1} \to_{R_P} s_{p+2}. \tag{8.14}$$

However, the string s_{p+1} encodes a configuration of P and there is at most one rewriting rule that is applicable to it. Thus the words s_p and s_{p+2} must be identical and hence the derivation (8.13) can be shortened by deleting the strings s_{p+1} and s_{p+2} contradicting the assumption. □

The above result due to the Jewish logician Emil Post (1897-1954) provides an effective reduction of derivation 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 result.

Theorem 8.9. *The word problem for Thue systems is undecidable.*

8.3 Semigroups

A *semigroup* is an algebraic structure consisting of a non-empty set S together with an associative binary operation. For instance, the set of all non-empty strings Σ^+ over an alphabet Σ together with the concatenation of strings is a semigroup, the *free semigroup* over Σ.

Each Thue systems gives rise to a semigroup in a natural way. To see this, let $(\Sigma, R^{(s)})$ be a Thue system. The rewriting relation $\overset{*}{\to}_{R^{(s)}}$ 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:

$$[s] = \{t \in \Sigma^+ \mid s \overset{*}{\to}_{R^{(s)}} t\}. \tag{8.15}$$

Proposition 8.10. *The set of equivalence classes $H = H(\Sigma, R^{(s)}) = \{[s] \mid s \in \Sigma^+\}$ forms a semigroup with the operation*

$$[s] \circ [t] = [st], \quad s, t \in \Sigma^+. \tag{8.16}$$

Proof. Claim that the operation is well-defined. Indeed, let $[s] = [s']$ and $[t] = [t']$ for some $s, s', t, t' \in \Sigma^+$. Then $s \overset{*}{\to}_{R^{(s)}} s'$, $s' \overset{*}{\to}_{R^{(s)}} s$, and $t \overset{*}{\to}_{R^{(s)}} t'$, $t' \overset{*}{\to}_{R^{(s)}} t$. Thus $st \overset{*}{\to}_{R^{(s)}} s't \overset{*}{\to}_{R^{(s)}} s't'$ and $s't' \overset{*}{\to}_{R^{(s)}} st' \overset{*}{\to}_{R^{(s)}} st$. It follows that $[st] = [s't']$ and so the claim is proved. □

Example 8.11. Consider the Thue system $(\Sigma, R^{(s)})$, where $\Sigma = \{a, b\}$ and $R = \{(ab, b), (ba, a)\}$. For instance, $a \to_{R^{(s)}} ba \to_{R^{(s)}} aba \to_{R^{(s)}} aa$ and $b \to_{R^{(s)}} ab \to_{R^{(s)}} bab \to_{R^{(s)}} bb$ and thus $[a] = [aa]$ and $[b] = [bb]$. ♦

The *word problem* for the semigroup $H = H(\Sigma, R^{(s)})$ asks whether arbitrary strings $s, t \in \Sigma^+$ describe the same element $[s] = [t]$ or not. By definition,

$$[s] = [t] \quad \Longleftrightarrow \quad s \overset{*}{\to}_{R^{(s)}} t, \quad s, t \in \Sigma^+. \tag{8.17}$$

This provides an effective reduction of the word problem Thue systems to the word problem for semigroups. This implies the following result which was independently established by Emil Post (1987-1954) and Andrey Markov Jr. (1903-1979).

Theorem 8.12. *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 is a finite set Π of pairs $(\alpha_i, \beta_i) \in \Sigma^+ \times \Sigma^+$, $1 \le i \le m$. For each finite sequence $\boldsymbol{i} = (i_1, \ldots, i_r) \in \{1, \ldots, m\}^+$ of indices, define the strings

$$\alpha(\boldsymbol{i}) = \alpha_{i_1} \circ \alpha_{i_2} \circ \cdots \circ \alpha_{i_r} \tag{8.18}$$

and

$$\beta(\boldsymbol{i}) = \beta_{i_1} \circ \beta_{i_2} \circ \cdots \circ \beta_{i_r}. \tag{8.19}$$

A *solution* of the PCS Π is a sequence $\boldsymbol{i}$ of indices such that $\alpha(\boldsymbol{i}) = \beta(\boldsymbol{i})$.

Example 8.13. The PCS $\Pi = \{(\alpha_1, \beta_1) = (a, aaa), (\alpha_2, \beta_2) = (abaa, ab), (\alpha_3, \beta_3) = (aab, b)\}$ over the alphabet $\Sigma = \{a, b\}$ has the solution $\boldsymbol{i} = (2, 1, 1, 3)$, since

$$\begin{aligned} \alpha(\boldsymbol{i}) &= \alpha_2 \circ \alpha_1 \circ \alpha_1 \circ \alpha_3 = abaa \circ a \circ a \circ aab \\ &= abaaaaaab \\ &= ab \circ aaa \circ aaa \circ b = \beta_2 \circ \beta_1 \circ \beta_1 \circ \beta_3 = \beta(\boldsymbol{i}). \end{aligned}$$

♦

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 is reduced to this problem. To this end, let (Σ, R) be a semi-Thue system, where $\Sigma = \{a_1, \ldots, a_n\}$ and $R = \{(u_i, v_i) \mid 1 \le i \le m\}$. 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 is 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 alphabet $\Sigma \cup \Sigma' \cup \{x, y, z\}$ by the following $2q + 4$ pairs:

$$\begin{aligned} (\alpha_i, \beta_i) &= (u_i, v'_i),\ 1 \le i \le m, \\ (\alpha_{m+i}, \beta_{m+i}) &= (a_i, a'_i),\ 1 \le i \le n, \\ (\alpha_{q+i}, \beta_{p+i}) &= (u'_i, v_i),\ 1 \le i \le m, \\ (\alpha_{q+m+i}, \beta_{p+m+i}) &= (a'_i, a_i),\ 1 \le i \le n, \\ (\alpha_{2q+1}, \beta_{2p+1}) &= (y, z), \\ (\alpha_{2q+2}, \beta_{2p+2}) &= (z, y), \\ (\alpha_{2p+3}, \beta_{2p+3}) &= (x, xsy), \\ (\alpha_{2p+4}, \beta_{2p+4}) &= (ztx, x). \end{aligned} \tag{8.20}$$

Example 8.14. Take the semi-Thue system (Σ, R), where $\Sigma = \{a, b\}$ and $R = \{(ab, bb), (ab, a), (b, aba)\}$. The corresponding PCS $\Pi = \Pi(\Sigma, R, s, t)$ over the alphabet $\{a, b, a', b', x, y, z\}$ consists of the pairs

$(ab, b'b')$,
(ab, a'),
$(b, a'b'a')$,
(a, a'),
(b, b'),
$(a'b', bb)$,
$(a'b', a)$,
(b', aba),
(a', a),
(b', b),
(y, z),
(z, y),
(x, xsy),
(ztx, x).

⧫

First, observe that the derivations of a semi-Thue system can be mimicked by the corresponding PCS.

Proposition 8.15. • *If $s, t \in \Sigma^+$ such that $s = t$ or $s \to_R t$, 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 $\beta(\boldsymbol{i}') = t$.*

- *If $s, t \in \Sigma^+$ such that there is a sequence $\boldsymbol{i} \in \{1, \ldots, q\}^+$ of indices such that $\alpha(\boldsymbol{i}) = s$ and $\beta(\boldsymbol{i}) = t'$, then $s \overset{*}{\to}_R t$.*
- *If $s, t \in \Sigma^+$ such that there is a sequence $\boldsymbol{i}' \in \{q+1, \ldots, 2q\}^+$ of indices such that $\alpha(\boldsymbol{i}') = s'$ and $\beta(\boldsymbol{i}') = t$, then $s \overset{*}{\to}_R t$.*

Example 8.16. (Cont'd) Let $s = aabb$ and $t = aab$. Then $s \to_R t$ by the rule $(ab, a) \in R$ and the sequences $\boldsymbol{i} = (4, 2, 5)$ and $\boldsymbol{i}' = (9, 7, 10)$ yield

$$\begin{aligned}
\alpha(\boldsymbol{i}) &= \alpha_4 \circ \alpha_2 \circ \alpha_5 = a \circ ab \circ b = aabb, \\
\beta(\boldsymbol{i}) &= \beta_4 \circ \beta_2 \circ \beta_5 = a' \circ a' \circ b' = a'a'b', \\
\alpha(\boldsymbol{i}') &= \alpha_9 \circ \alpha_7 \circ \alpha_{10} = a' \circ a'b' \circ b' = a'a'b'b', \\
\beta(\boldsymbol{i}') &= \beta_9 \circ \beta_7 \circ \beta_{10} = a \circ a \circ b = aab.
\end{aligned}$$

Let $s = bbaba = \alpha(5, 3, 2, 4)$ and $t' = b'a'b'a'a' = \beta(5, 3, 2, 4)$. Then $s = bbaba \to_R babaaba \to_R babaaa = t$ and so $s \overset{*}{\to}_R t$. ⧫

Second, the solutions of the constructed PCS can always be canonically decomposed.

Proposition 8.17. • *Let $\boldsymbol{i} = (i_1, \ldots, i_r)$ be a solution of the PCS $\Pi = \Pi(\Sigma, R, s, t)$ such that $w = \alpha(\boldsymbol{i}) = \beta(\boldsymbol{i})$. If $w = w_1 y w_2$ for some $w_1, w_2 \in \Sigma'^*$, there is an index j, $1 \le j \le r$, such that*

$$\alpha(i_1, \ldots, i_{j-1}) = w_1, \ \alpha(i_j) = y, \ \text{and} \ \alpha(i_{j+1}, \ldots, i_r) = w_2. \tag{8.21}$$

- *Let* $\boldsymbol{i} = (i_1, \ldots, i_r)$ *be a solution of the PCS* $\Pi = \Pi(\Sigma, R, s, t)$ *such that* $w = \alpha(\boldsymbol{i}) = \beta(\boldsymbol{i})$. *If* $w = w_1 z w_2$ *for some* $w_1, w_2 \in \Sigma'^*$, *there is an index* j, $1 \leq j \leq r$, *such that either*

$$\alpha(i_1, \ldots, i_{j-1}) = w_1,\ \alpha(i_j) = z,\ \textit{and}\ \alpha(i_{j+1}, \ldots, i_r) = w_2, \tag{8.22}$$

or

$$\alpha(i_1, \ldots, i_{j-1}) = w_1,\ \alpha(i_j) = ztx,\ \alpha(i_{j+1}, \ldots, i_r) = u,\ \textit{and}\ w_2 = txu. \tag{8.23}$$

Example 8.18. (Cont'd) Let $s = b$ and $t = abaa$. A solution of the PCS Π is given by

$$\boldsymbol{i} = (13, 3, 11, 6, 9, 12, 3, 5, 4, 11, 9, 10, 7, 9, 14),$$

since

$$\begin{aligned}\alpha(\boldsymbol{i}) &= x \circ b \circ y \circ a'b' \circ a' \circ z \circ b \circ b \circ a \circ y \circ a' \circ b' \circ a'b' \circ a' \circ zabaax \\ &= xbya'b'a'zbbaya'b'a'b'a'zabaax \\ &= xby \circ a'b'a' \circ z \circ bb \circ a \circ y \circ a'b'a' \circ b' \circ a' \circ z \circ a \circ b \circ a \circ a \circ x \\ &= \beta(\boldsymbol{i}).\end{aligned}$$

It follows that

$$w_1 = xbya'b'a'zbba = \alpha(13, 3, 11, 6, 9, 12, 3, 5, 4)$$

and

$$w_2 = a'b'a'b'a'zabaax = \alpha(9, 10, 7, 9, 14),$$

or

$$w_1 = xbya'b'a' = \alpha(13, 3, 11, 6, 9)$$

and

$$w_2 = bbaya'b'a'b'a'zabaax = \alpha(3, 5, 4, 9, 10, 7, 9, 14),$$

or

$$w_1 = xbya'b'a'zbbaya'b'a'b'a' = \alpha(13, 3, 11, 6, 9, 3, 5, 4, 9, 10, 7, 9),$$

and

$$ztx = zabaax = \alpha(14) \text{ and } w_2 = abaax = tx,$$

where u is the empty word. ⧫

Note that if the PCS $\Pi = \Pi(\Sigma, R, s, t)$ has a solution $\alpha(\boldsymbol{i}) = \beta(\boldsymbol{i})$, where $\boldsymbol{i} = (i_1, \ldots, i_r)$, the solution starts with $i_1 = 2q + 3$ and ends with $i_r = 2p + 4$, i.e., the corresponding string has the prefix xsy and the postfix ztx. The reason is that $(\alpha_{2p+3}, \beta_{2p+3}) = (x, xsy)$ is the only pair whose components have the same prefix and $(\alpha_{2p+4}, \beta_{2p+4}) = (ztx, x)$ is the only pair whose components have the same postfix.

Proposition 8.19. *Let $s, t \in \Sigma^+$. If there is a derivation $s \xrightarrow{*}_R t$ in the semi-Thue system (Σ, R), the PCS $\Pi = \Pi(\Sigma, R, s, t)$ has a solution.*

Proof. Let $s \xrightarrow{*}_R t$ such that

$$s = s_1 \to_R s_2 \to_R s_3 \to_R \ldots \to_R s_{m-1} \to_R s_m = t, \tag{8.24}$$

where $s_i \to_R s_{i+1}$ or $s_i = s_{i+1}$, $0 \leq i \leq m-1$. The inclusion of steps without effect allows to assume that m is odd. By Proposition 8.15, for each j, $1 \leq j \leq m$, 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. Then a solution of the PCS is given by the sequence

$$\boldsymbol{i} = (2p+3, \boldsymbol{i}^{(1)}, 2p+1, \boldsymbol{i}^{(2)}, 2p+2, \boldsymbol{i}^{(3)}, 2p+1, \ldots, \boldsymbol{i}^{(m-2)}, 2p+1, \boldsymbol{i}^{(m-1)}, 2p+4). \tag{8.25}$$

Indeed, the evaluation of the solution yields

$$\begin{array}{lccccccccccc}
\alpha(\boldsymbol{i}): & x & s_1 & y & s'_2 & z & s_3 & y & \ldots & s_{m-2} & y & s'_{m-1} & zs_mx \\
 & . & . & . & . & . & . & . & & . & . & . & . \\
\boldsymbol{i}: & 2p+3 & \boldsymbol{i}^{(1)} & 2p+1 & \boldsymbol{i}^{(2)} & 2p+2 & \boldsymbol{i}^{(3)} & 2p+1 & \ldots & \boldsymbol{i}^{(m-2)} & 2p+1 & \boldsymbol{i}^{(m-1)} & 2p+4 \\
 & . & . & . & . & . & . & . & & . & . & . & . \\
\beta(\boldsymbol{i}): & xs_1y & s'_2 & z & s_3 & y & s'_4 & z & \ldots & s'_{m-1} & z & s_m & x
\end{array}$$

□

Example 8.20. (Cont'd) Consider $s = aab \to_R abb \to abb \to_R aabab \to_R aaab \to_R aaa = t$. A solution of the PCS $\Pi = \Pi(\Sigma, R, aab, aaa)$ is given by

$$\boldsymbol{i} = (13, 4, 1, 11, 9, 8, 10, 12, 4, 2, 4, 5, 11, 9, 9, 7, 14),$$

where

$$\begin{array}{lccccccccc}
\alpha(\boldsymbol{i}): & x & aab & y & a'b'b' & z & aabab & y & a'a'a'b' & zaaax \\
 & . & . & . & . & . & . & . & . & . \\
\boldsymbol{i}: & 13 & (4,1) & 11 & (9,8,10) & 12 & (4,2,4,5) & 11 & (9,9,7) & 14 \\
 & . & . & . & . & . & . & . & . & . \\
\beta(\boldsymbol{i}): & xaaby & a'b'b' & z & aabab & y & a'a'a'b' & z & aaa & x
\end{array}$$

♦

Proposition 8.21. *Let $s, t \in \Sigma^+$. If the PCS $\Pi = \Pi(\Sigma, R, s, t)$ has a solution, there is a derivation $s \xrightarrow{*}_R t$ in the semi-Thue system (Σ, R).*

Proof. Let $\boldsymbol{i} = (i_1, \ldots, i_r)$ be a minimal solution of the PCS Π. The observation prior to Proposition 8.19 shows that the string $w = \alpha(\boldsymbol{i}) = \beta(\boldsymbol{i})$ must have the form $w = xsy \ldots ztx$.

By Proposition 8.17, there is a sequence $\boldsymbol{i}^{(1)}$ of indices such that $\alpha(\boldsymbol{i}^{(1)}) = s = s_1$. Put $s'_2 = \beta(\boldsymbol{i}^{(1)})$. It follows that $w = xs_1ys'_2z \ldots$ and by Proposition 8.15, $s_1 \xrightarrow{*}_R s_2$.

By Proposition 8.17, there is a sequence $\boldsymbol{i}^{(2)}$ of indices such that $\alpha(\boldsymbol{i}^{(2)}) = s'_2$. Put $s_3 = \beta(\boldsymbol{i}^{(2)})$. Thus $w = xs_1ys'_2zs_3 \ldots$ and by Proposition 8.15, $s_2 \xrightarrow{*}_R s_3$.

By Proposition 8.17, there are two possibilities:

- Let $\boldsymbol{i} = (2q+3, \boldsymbol{i}^{(1)}, 2q+1, \boldsymbol{i}^{(2)}, 2q+2, \ldots)$, i.e., $w = xs_1ys_2'zs_3y\ldots$. Then the sequence can be continued as indicated above.
- Let $\boldsymbol{i} = (2q+3, \boldsymbol{i}^{(1)}, 2q+1, \boldsymbol{i}^{(2)}, 2q+4, \ldots)$. This is only possible if $s_3 = t$ and $w = xs_1ys_2'ztx$. It follows that $\boldsymbol{i} = (2q+3, \boldsymbol{i}^{(1)}, 2q+1, \boldsymbol{i}^{(2)}, 2q+4)$ is already a solution due to minimality:

$$\begin{array}{lccccc} \alpha(\boldsymbol{i}): & x & s_1 & y & s_2' & ztx \\ & \cdot & \cdot & \cdot & \cdot & \cdot \\ \boldsymbol{i}: & 2q+3 & \boldsymbol{i}^{(1)} & 2q+1 & \boldsymbol{i}^{(2)} & 2q+4 \\ & \cdot & \cdot & \cdot & \cdot & \cdot \\ \beta(\boldsymbol{i}): & xs_1y & s_2' & z & t & x \end{array}$$

  Moreover, $s = s_1 \xrightarrow{*}_R s_2$ and $s_2 \xrightarrow{*}_R s_3 = t$ and thus $s \xrightarrow{*}_R t$.

By induction, there are strings $s = s_1, s_2, s_3, \ldots, s_r = t$ in Σ^+ such that

$$\alpha(\boldsymbol{i}) = x \circ s_1 \circ y \circ s_2' \circ z \circ s_3 \circ \ldots \circ s_{r-2} \circ y \circ s_{r-1}' \circ z \circ s_r \circ x = \beta(\boldsymbol{i})$$

and $s_i \xrightarrow{*}_R s_{i+1}$, $1 \le i \le r-1$. It follows that $s \xrightarrow{*}_R t$ as required. □

Example 8.22. (Cont'd) Let $s = b$ and $t = abaa$. A solution of the PCS Π is given by

$$\boldsymbol{i} = (13, 3, 11, 6, 9, 12, 3, 5, 4, 11, 9, 10, 7, 9, 14).$$

The corresponding string

$$\alpha(\boldsymbol{i}) = \beta(\boldsymbol{i}) = xbya'b'a'zbbaya'b'a'b'a'zabaax$$

provides a derivation in the semi-Thue system:

$$s = b \xrightarrow{*}_R aba \xrightarrow{*}_R bba \xrightarrow{*}_R ababa \xrightarrow{*}_R abaa = t.$$

♦

The last two propositions provide a 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 giving the following result.

Theorem 8.23. *The word problem for Post correspondence systems is undecidable.*

8.5 Context-free Languages

The context-free languages form an important class of formal languages which is 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 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 (u, ϵ) 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.,

$$L(G) = \{w \in \Sigma^* \mid S \stackrel{*}{\rightarrow}_R w\}. \tag{8.26}$$

The language $L(G)$ generated by a context-free grammar G is called *context-free*.

Example 8.24. • The archetypical context-free language $L(G) = \{a^n b^n a^k \mid n \geq 1, k \geq 1\}$ 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 *aabba* is derived as follows: $S \rightarrow_R AB \rightarrow_R Aa \rightarrow aAba \rightarrow aabba$.

- An *arithmetic expression* in a programming language is a combination of constants, variables, and operators. A typical context-free grammar describing (simple) arithmetic expressions consists of the rewriting rules

$$\begin{aligned}
\text{expr} &\rightarrow \text{expr op expr}\\
\text{expr} &\rightarrow (\text{expr})\\
\text{expr} &\rightarrow -\text{expr}\\
\text{expr} &\rightarrow \text{id}\\
\text{op} &\rightarrow +\\
\text{op} &\rightarrow -\\
\text{op} &\rightarrow *\\
\text{op} &\rightarrow /\\
\text{op} &\rightarrow \uparrow
\end{aligned}$$

  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 is reduced to this problem. For this, let (Σ, Π) be a PCS, where Σ contains 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 $\{S\}$ and so the same start symbol S. Moreover, the rewriting rules of G_α are

$$R_\alpha = \{(S, i\alpha_i \mid 1 \leq i \leq m\} \cup \{(S, iS\alpha_i \mid 1 \leq i \leq m\} \tag{8.27}$$

and the rewriting rules of G_β are

$$R_\beta = \{(S, i\beta_i \mid 1 \leq i \leq m\} \cup \{(S, iS\beta_i \mid 1 \leq i \leq m\}. \tag{8.28}$$

The construction immediately implies the following result.

Proposition 8.25. *The languages of the grammars G_α and G_β are*

$$L(G_\alpha) = \{i_n \ldots i_1 \alpha_{i_1} \ldots \alpha_{i_n} \mid 1 \leq i_1, \ldots, i_n \leq m\} \tag{8.29}$$

and

$$L(G_\beta) = \{i_n \dots i_1 \beta_{i_1} \dots \beta_{i_n} \mid 1 \le i_1, \dots, i_n \le m\}. \tag{8.30}$$

Moreover,

$$L(G_\alpha) \cap L(G_\beta) = \{i_n \dots i_1 \alpha_{i_1} \dots \alpha_{i_n} \mid \boldsymbol{i} = (i_1, \dots, i_n) \text{ solves } (\Sigma, \Pi)\}. \tag{8.31}$$

Example 8.26. Take the PCS (Σ, Π), where $\Sigma = \{a, b\}$ and $\Pi = \{(a, aaa), (abaa, ab), (aab, b)\}$. The corresponding grammars G_α and G_β are given by the respective sets of rewriting rules

$$R_\alpha = \{(S, 1a), (S, 2abaa), (S, 3aab), (S, 1Sa), (S, 2Sabaa), (S, 3Saab)\} \tag{8.32}$$

and

$$R_\beta = \{(S, 1aaa), (S, 2ab), (S, 3b), (S, 1Saaa), (S, 2Sab), (S, 3Sb)\}. \tag{8.33}$$

A solution of the PCS is $\boldsymbol{i} = (2, 1, 1, 3)$, since

$$\begin{aligned} \alpha(\boldsymbol{i}) &= \alpha_2 \circ \alpha_1 \circ \alpha_1 \circ \alpha_3 = abaa \circ a \circ a \circ aab \\ &= abaaaaaab \\ &= ab \circ aaa \circ aaa \circ b = \beta_2 \circ \beta_1 \circ \beta_1 \circ \beta_3 = \beta(\boldsymbol{i}). \end{aligned}$$

The corresponding derivations in G_α and G_β are

$$S \to 3Saab \to 31Saaab \to 311Saaaab \to 3112abaaaaaab$$

and

$$S \to 3Sb \to 31Saaab \to 311Saaaaaab \to 3112abaaaaaab.$$

Thus the intersection $L(G_\alpha) \cap L(G_\beta)$ is non-empty. ⧫

The above proposition shows that the context-free languages $L(G_\alpha)$ and $L(G_\beta)$ intersect non-trivially if and only if the given PCS has a solution. But the word problem for Post correspondence systems is undecidable giving rise to the following result.

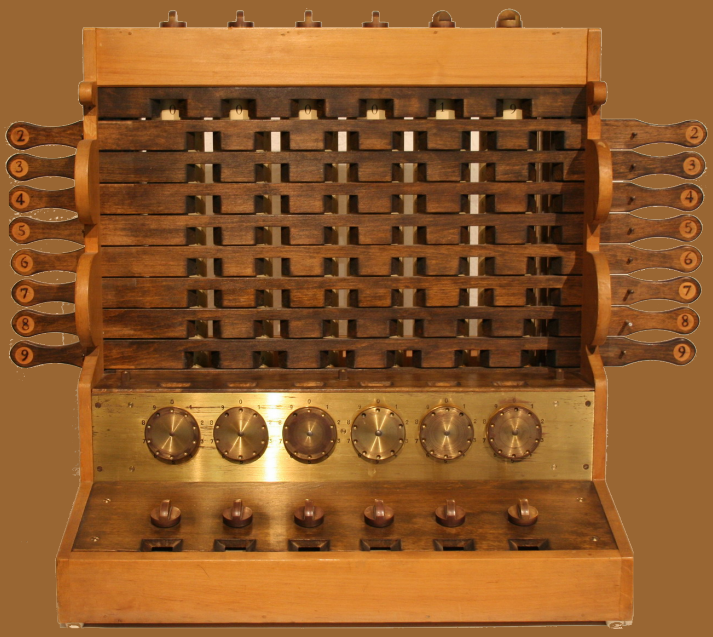
Theorem 8.27. *The problem to decide whether the intersection of two context-free languages is non-empty or not is undecidable.*

Index

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 solvable by an algorithm, a rigorous formalism is necessary that allows mathematical proofs. The need for such a formalism became apparent in the works of David Hilbert (1900) on the foundations of mathematics and Kurt Gödel (1931) on the incompleteness of elementary arithmetic.

The book is a development of class notes for a lecture on computability held for Bachelor students of Computer Science at the Hamburg University of Technology in the summer term 2011. The aim of the course was to present the basic results of the field:

- mathematical models of computation (Turing machine, unlimited register machine, LOOP and GOTO programs),
- primitive recursive and partial recursive functions, Ackermann's function,
- Gödel numbering,
- acceptable programming systems (universal functions, smn theorem, Kleene's normal form),
- undecidable sets, theorems of Rice, and word problems (diophantine sets, semi-Thue systems, semigroups, Post's correspondence problem).



The Schickard machine is a non-programable calculating machine for integers invented by Wilhelm Schickard (1592-1646).

Front Page: The „Analytical Engine“ is a mechanical programmable machine designed by Charles Babbage (1792–1871).

urn:nbn:de:gbv:830-tubdok-11072