Teorija izračunljivosti 2022–23

Alex Simpson

Alex.Simpson@fmf.uni-lj.si

Version of March 9, 2023

Contents

Mathematical preliminaries

Partial functions A *partial function* f from a set X to a set Y (notation $f\colon X \rightharpoonup Y$) is given by a subset $\mathsf{dom}(f) \subseteq X$ (the *domain* of f) together with a function $f\colon \mathsf{dom}(f) \to Y$.

When we write $f(x) = y$ this implies that $x \in \mathsf{dom}(f)$. If $x \notin \mathsf{dom}(f)$, we say that f is *undefined* on x. If e and e' are mathematical expressions that are potentially undefined (such as $f(x)$, where f is a partial function) then we write $e \simeq e'$ (*Kleene equality*) to mean that each of e and e' is defined if and only if the other is and if they are defined then they are equal. We further write $f(x)\!\downarrow$ to say that $f(x)$ is defined, and $f(x)\!\uparrow$ or $f(x) \simeq \uparrow$ to say that $f(x)$ is undefined.

Sets of words For any set Σ, we write Σ^* for the set of all *words* (i.e., finite sequences) of elements from Σ. We write $|x|$ for the length of a word x. We typically expand an individual word x as $x = x_0 \dots x_{|x|-1}$, and we write ε for the unique word of length 0 (the *empty* word). For $a \in \Sigma$, we write a^n for the constant word $aa \dots a \in \Sigma^*$ of length n.

Infinite words We write Σ^ω for the set of all *infinite words* (i.e., infinite sequences) of elements from Σ. We typically expand an individual word $p \in \Sigma^\omega$ as $p = p_0 p_1 p_2 \dots$. We write $p\restriction_n$ for the length n prefix of p; i.e., $p\restriction_n := p_0 \dots p_{n-1}$. For $a \in \Sigma$, we write a^ω for the infinite constant word $aaaa \cdots \in \Sigma^\omega$.

Updating a function Given a function $f : X \to Y$ and any $a \in X$ and $b \in Y$ the *update function* $f[a \mapsto b]$ is defined by

$$f[a \mapsto b](x) \;=\; \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases} .$$

Set-theoretic notation Although our notation is standard, one subtle point is that we write $A \subset B$ for the *strict* subset relation and $A \subseteq B$ for the *non-strict* one. For example, $A \subset B$ implies $A \neq B$. Similarly, $A \subseteq B$ if and only if $A \subset B$ or $A = B$.

Equivalence classes If $\sim$ is an equivalence relation on a set A then we write $A/\!\sim$ for the set of equivalence classes (the *quotient set*). For any $x \in A$, we write $[x]$ for the equivalence class containing x.

Acknowledgements Thank you to Agustin Rodriguez Agudo, Boštjan Gec, Svenja Griesbach, Andraž Jelenc, Severin Mejak, Andraž Pustoslemšek and especially Davorin Lešnik for suggesting improvements to earlier versions of these notes.

1 Algorithms and Turing Machines

1.1 Algorithms

Throughout the history of mathematics, *computation* has been an intrinsic part of mathematics. Intimately associated with computation is the notion of *algorithm*: a precise description of the sequence of steps required to carry out a computation.

The very incomplete list below recalls some historically notable algorithms.

- Euclid's algorithm to compute gcd.
- Algorithms to add, substract, multiply and divide numbers in decimal.
- Algorithms for approximating real numbers to arbitrary precision; e.g., π, e, computing trigonometric functions, logarithm, square root, etc.
- Many widely applied modern algorithms such as: the simplex algorithm for linear programming, algorithms to efficiently test the primality of a large number, etc.

Until the end of the 19th century, mathematicians were content to recognise algorithms on a case-by-case basis. However, the situation changed in the 20th century. In 1900, right at the turn of the century, Hilbert's famous 23 problems in mathematics included a challenge directly involving the notion of algorithm.

Hilbert's 10th Problem. Find an algorithm to determine whether a given polynomial equation with integer coefficients (Diophantine equation) has an integer solution.

Some years later, in 1928, Hilbert asked:

Hilbert's Entscheidungsproblem. Find an algorithm to determine whether a sentence in first-order logic is valid in all structures (equivalently is provable in first-order logic).

The two problems are related. A positive solution to Hilbert's *Entscheidungsproblem* would *a fortiori* provide a positive solution to his 10th Problem.

In 1936, Alonzo Church and Alan Turing independently proved that there does not exist any algorithm solving the *Entscheidungsproblem*. In technical language: Hilbert's *Entscheidungsproblem* is *undecidable*. Many years later, in 1970, Yuri Matiyasevich proved that Hilbert's 10th Problem is also undecidable.

In order to be in a position to give a mathematical proof that there exists no possible algorithm, one needs to answer a fundamental question:

- What is an algorithm?

We shall begin this course with Turing's answer to this question, defined in terms of his eponymous *Turing machines*. This answer has withstood the test of time. It is mathematically simple, and it has conceptual clarity in directly appealing to our intuitions about the mechanics of the process of computation.

Some of the considerations motivating the definition of a Turing machine are listed below.

- An algorithm should be a finite description of a process of computation.

- An algorithm should specify the computation process entirely.
- The process of following an algorithm should be one that can be carried out in practice without any creative input.
- Thus the process of following an algorithm should be one that could be carried out by a suitable machine.
- The machine will perform steps in time and will need enough "working space" to carry out calculations. For maximum generality, we should not bound time and space in advance.

The motivation and how it leads to the definition of a Turing machine are discussed both thoroughly and eloquently in Turing's original 1936 article, *"On computable numbers, with an application to the Entscheidungsproblem"*. This can be found on the course webpage. You are recommended to have a look at this classic paper.

1.2 Turing Machines

Definition 1.1 (Turing machine)**.** A *(deterministic) Turing machine* is specified by:

- a finite set Γ (the *tape alphabet*) with $\Box \in \Gamma$ (the *blank* symbol);
- a finite set Q of *states* with $\mathsf{start} \in Q$ (the *start* state);
- a partial function $\delta \colon Q \times \Gamma \rightharpoonup Q \times \Gamma \times \{-1, 0, +1\}$ (the *transition function*).

Definition 1.2 (Tape configuration)**.** A *tape configuration* is a function $t \colon \mathbb{Z} \to \Gamma$ satisfying:

$$\exists n \geq 0 \;\; \forall m \in \mathbb{Z} \quad |m| \geq n \implies t(m) = \Box \;\; .$$

One thinks of t as an infinite (in both directions) tape divided into squares indexed by integers. The function t gives the symbol written on each square. The condition says that only finitely many squares are non-blank.

Every word $x = x_0 x_1 \ldots x_{k-1} \in \Gamma^*$ and $j \in \mathbb{Z}$ determine a tape configuration $x@j$

$$x@j(i) \;\; = \;\; \begin{cases} x_{i-j} & \text{if } j \leq i < j + k \\ \Box & \text{otherwise} \end{cases} \;\; .$$

That is, the word x is written on the tape, with its first symbol at position j, and, apart from x, the tape is blank.

Note that, for example, ε (the empty word), $\Box$ and $\Box\Box$ are distinct words that give rise to the same tape configuration.

Definition 1.3 (Machine configuration)**.** A *(machine) configuration* is a triple (q, t, i) where:

- $q \in Q$ (the current *state*)
- t is a tape configuration;
- $i \in \mathbb{Z}$ (the *head position*).

We write Config for the set of all possible machine configurations.

Single computation steps are given by the partial function

$$\mathsf{step} : \mathsf{Config} \rightharpoonup \mathsf{Config}$$

defined by:

$$\mathsf{step}(q,t,i) \;=\; \begin{cases} (q',\, t[i \mapsto a],\, i+d) & \text{if } \delta(q,t(i)) = (q',a,d) \\ \text{undefined} & \text{if } \delta(q,t(i)) \text{ is undefined} \end{cases} .$$

Here we use the *update function* $t[i \mapsto a]$ as defined on page ii.

Definition 1.4 (Halting)**.** A configuration (q,t,i) is called a *halting configuration* if step is undefined on (q,t,i). We write $t \Downarrow (q,t',i)$ if (q,t',i) is a halting configuration and there exists $n \geq 0$ with

$$\mathsf{step}^n(\mathsf{start},t,0) \;=\; (q,t',i) \;.$$

A state q is called a *halting state* if, for every $a \in \Gamma$, the transition function δ is undefined on (q,a).

Observe that if q is a halting state then every configuration of the form (q,t,i) is a halting configuration.

Turing machines have different *modi operandi* depending on their purpose. We consider two purposes:

- recognising formal languages.
- computing partial functions (especially on the natural numbers);

1.3 Recognising formal languages

We assume a Turing machine comes with:

- a distinguished *input alphabet* $\Sigma \subseteq \Gamma - \{\Box\}$;
- distinct halting states $\mathsf{accept}, \mathsf{reject} \in Q$.

Recall that a *language* is a subset $L \subseteq \Sigma^*$.

Definition 1.5 (Acceptance/rejection)**.**

- A TM *accepts* $x \in \Sigma^*$ if $x@0 \Downarrow (\mathsf{accept},t,i)$ for some t,i.
- A TM *rejects* $x \in \Sigma^*$ if $x@0 \Downarrow (\mathsf{reject},t,i)$ for some t,i.

Definition 1.6 (Recognised language)**.** The *language recognised* by a TM M is

$$\mathcal{L}(M) \;:=\; \{x \in \Sigma^* \mid M \text{ accepts } x\}$$

Definition 1.7 (Decidability/semidecidability)**.**

- $L \subseteq \Sigma^*$ is said to be *semidecidable* (or *semicomputable*, or *computably enumerable*) if there exists a TM M such that $L = \mathcal{L}(M)$.

  (We also say that the machine M *semidecides* L.)

- $L \subseteq \Sigma^*$ is said to be *decidable* (or *computable*) if there exists a TM M such that $L = \mathcal{L}(M)$ and also M rejects every $x \in \Sigma^* - L$.

  (We also say that the machine M *decides* L.)

Proposition 1.8. *If a language L is decidable then it is semidecidable.*

1.4 Computing partial functions

We assume a Turing machine comes with:

- a distinguished *input/output alphabet* $\Sigma \subseteq \Gamma - \{\Box\}$;
- a distinguished halting state $\mathsf{halt} \in Q$.

Definition 1.9 (Computing a partial function)**.** A TM is said to *compute* a partial function $f\colon \Sigma^* \rightharpoonup \Sigma^*$ if:

- for all $x \in \mathsf{dom}(f)$, it holds that $x@0 \Downarrow (\mathsf{halt}, f(x)@i, i)$ for some $i \in \mathbb{Z}$; and
- for all $x \notin \mathsf{dom}(f)$, it is not the case that $x@0 \Downarrow (\mathsf{halt}, t, i)$, for any t, i.

Definition 1.10 (Computable partial function)**.** A partial function $f\colon \Sigma^* \rightharpoonup \Sigma^*$ is said to be *computable* if there exists some TM that computes it.

(Terminology issue: many authors use *partial computable function*, whereas we say *computable partial function*.)

1.5 Variant Turing machines

Many variations on the notion of Turing machine can be defined. In every case it is possible to simulate the variant machines using Turing machines in the sense we defined.

Commonly considered such variations include, for example, the following.

1. Machines with k tapes, where $k > 1$, each with its own head.
2. Machines with multiple heads on the same tape.
3. Machines with heads that can move between tapes.
4. Machines with two (or higher) dimensional grids of symbols instead of one-dimensional tapes.
5. Machines in which individual tape cells contain words rather than just single letters.
6. etc.

We look at the first of these examples, k-tape machines in more detail below. It is a worthwhile exercise to consider a couple of other variants and convince yourself (at a high level) that you understand how to simulate such more complicated machines on an ordinary Turing machine.

1.6 Multi-tape Turing machines

For $k \geq 1$, a k-tape Turing machine has k distinct two-way infinite tapes, each with its own read/write head.

Definition 1.11 (k-tape Turing machine)**.** For $k \geq 1$ a *(deterministic) k-tape Turing machine* is specified by:

- a finite set Γ (the *tape alphabet*) with $\square \in \Gamma$;
- a finite set Q of *states* with $\mathsf{start} \in Q$;
- a partial function $\delta \colon Q \times \Gamma^k \rightharpoonup Q \times \Gamma^k \times \{-1, 0, +1\}^k$.

Definition 1.12 (Machine configuration)**.** A *(machine) configuration* is a tuple $(q, t_1, \ldots, t_k, i_1, \ldots, i_k)$ where:

- $q \in Q$ (the current *state*)
- $t_1, \ldots, t_k$ are k tape configurations;
- $i_1, \ldots, i_k \in \mathbb{Z}$ (the *head positions*).

We again write Config for the set of all possible machine configurations.

The single-computation-step partial function

$$\mathsf{step} : \mathsf{Config} \rightharpoonup \mathsf{Config}$$

is defined in the natural way:

$$\mathsf{step}(q, t_1, \ldots, t_k, i_1, \ldots, i_k) \;=\; \begin{cases} (q',\, t_1[i_1 \mapsto a_1], \ldots, t_k[i_k \mapsto a_k],\, i_1{+}d_1, \ldots, i_k{+}d_k) \\ \qquad \text{if } \delta(q, t_1(i_1), \ldots, t_k(i_k)) = (q', a_1, \ldots, a_k, d_1, \ldots, d_k) \\ \text{undefined} \\ \qquad \text{if } \delta(q, t_1(i_1), \ldots, t_k(i_k)) \text{ is undefined} \end{cases} .$$

1.7 Simulating a k-tape machine by a single-tape machine

We show how to *simulate* a k-tape Turing machine using a single tape Turing machine.

Suppose we have a k-tape TM:

$$M \;=\; (\Gamma,\; Q,\; \delta \colon Q \times \Gamma^k \rightharpoonup Q \times \Gamma^k \times \{-1, 0, +1\}^k) \;.$$

We define an associated single tape machine:

$$\widetilde{M} \;=\; (\widetilde{\Gamma},\; \widetilde{Q},\; \widetilde{\delta} \colon \widetilde{Q} \times \widetilde{\Gamma} \rightharpoonup \widetilde{Q} \times \widetilde{\Gamma} \times \{-1, 0, +1\}) \;.$$

To do this, we need to define $\widetilde{\Gamma}$, $\widetilde{Q}$, and $\widetilde{\delta}$, and we need to say how the execution of the resulting single tape Turing machine relates to the execution of the original multi-tape machine.

We shall give only the main ideas of the construction. Constructions on Turing machines can be rather tedious if given in overmuch detail.

The new tape alphabet $\widetilde{\Gamma}$ is defined by

$$\widetilde{\Gamma} \;=\; \Gamma \cup \{\triangle, \|\} \;.$$

The new set of states $\widetilde{Q}$ is quite complex, and we shall not describe it fully. But, importantly, it contains within it a subset

$$\{\mathsf{now}_q \mid q \in Q\} \;\subseteq\; \widetilde{Q} \;.$$

The main idea is as follows. The machine $\widetilde{M}$ will start in the state $\mathsf{now}_{\mathsf{start}}$ in a machine configuration that encodes the starting configuration of the k-tape machine M. As the execution of the k-tape machine M proceeds as a sequence of steps going from state to state, $\mathsf{start} \to q_1 \to q_2 \to \ldots$, the single-tape machine $\widetilde{M}$ will go through states $\mathsf{now}_{\mathsf{start}} \to^+ \mathsf{now}_{q_1} \to^+ \mathsf{now}_{q_2} \to \ldots$, where the symbol $\to^+$ means many steps of computation by the single-tape machine $\widetilde{M}$. Moreover, whenever the single-tape machine $\widetilde{M}$ is in the state now_{q_i}, the machine configuration at the time will correctly encode the machine configuration of M when it is in state q_i.

Definition 1.13 (Configuration encoding)**.** A machine configuration $(\widetilde{q}, \widetilde{t}, \widetilde{i})$ of $\widetilde{M}$ is said to *encode* a machine configuration $(q, t_1, \ldots, t_k, i_1, \ldots, i_k)$ of M if all of the following hold.

- $\widetilde{q} = \mathsf{now}_q$.
- The tape $\widetilde{t}$ contains exactly $k+1$ occurrences of the symbol $\|$. Let $j_0 < j_1 < \cdots < j_k$ be the positions at which $\widetilde{t}$ is $\|$.
- $\widetilde{i} = j_0$.
- The tape $\widetilde{t}$ is blank at every $i < j_0$ and at every $i > j_k$.
- The tape $\widetilde{t}$ contains exactly k occurrences of the symbol $\triangle$. Moreover, if we write $h_1 < \cdots < h_k$ for the positions at which $\widetilde{t}$ is $\triangle$ then, for every $l = 1 \ldots k$, we have $j_{l-1} < h_l < j_l - 1$.
- For every $l = 1 \ldots k$, we have

$$t_l(n) \;=\; \begin{cases} \widetilde{t}(n - i_l + h_l + 1) & \text{if } n \geq i_l \text{ and } n - i_l + h_l + 1 < j_l \\ \widetilde{t}(n - i_l + h_l) & \text{if } n < i_l \text{ and } j_{l-1} < n - i_l + h_l \\ \square & \text{otherwise.} \end{cases}$$

  (Informally, this means that, if the $\triangle$ square at h_l is removed from $\widetilde{t}$ then the remaining squares from $j_{l-1}+1$ to $j_l - 1$ fully represent the non-blank portion of the tape t_l. The role of the $\triangle$ square at h_l is to mark that the l-th head of M, which is at position i_l of t_l, is pointing to the square on t_l that appears at position $h_l + 1$ of $\widetilde{t}$.)

The remaining states and transition relation of $\widetilde{M}$ are then defined to ensure that the following *simulation property* is true.

Proposition 1.14. *Let $(q, t_1, \ldots, t_k, i_1, \ldots, i_k)$ be a configuration of the k-tape machine M, and let $(\mathsf{now}_q, \widetilde{t}, \widetilde{i})$ be any configuration of $\widetilde{M}$ that encodes it. Then the following hold.*

- *If* $(q, t_1, \ldots, t_k, i_1, \ldots, i_k) \to (q', t'_1, \ldots, t'_k, i'_1, \ldots, i'_k)$ *in one step of computation of* M *then* $(\mathsf{now}_q, \widetilde{t}, \widetilde{i}) \to^+ (\mathsf{now}_{q'}, \widetilde{t'}, \widetilde{i'})$ *via possibly many steps of computation of* $\widetilde{M}$*, where* $(\mathsf{now}_{q'}, \widetilde{t'}, \widetilde{i'})$ *encodes* $(q', t'_1, \ldots, t'_k, i'_1, \ldots, i'_k)$ *and no state of the form* $\mathsf{now}_{q''}$ *is encountered at any intermediate step in the* $\widetilde{M}$ *computation.*

- *If* $(\mathsf{now}_q, \widetilde{t}, \widetilde{i}) \to^+ (\mathsf{now}_{q'}, \widetilde{t'}, \widetilde{i'})$ *via possibly many steps of computation of* $\widetilde{M}$*, where none of the intermediate steps involve states of the form* $\mathsf{now}_{q''}$*, then* $(q, t_1, \ldots, t_k, i_1, \ldots, i_k) \to (q', t'_1, \ldots, t'_k, i'_1, \ldots, i'_k)$*, in one step of computation of* M*, such that* $(\mathsf{now}_{q'}, \widetilde{t'}, \widetilde{i'})$ *encodes* $(q', t'_1, \ldots, t'_k, i'_1, \ldots, i'_k)$*.*

- *If* q *is a halting state of* M *then* now_q *is a halting state of* $\widetilde{M}$*.*

While we don't give the details of the definitions of $\widetilde{Q}$ and $\widetilde{\delta}$, the idea behind the full definition of $\widetilde{M}$ is that it should compute as follows.

1. The head scans the tape from left to right until the last $\|$ is encountered (this can be recognised because it is the $(k+1)$-th $\|$ symbol). Along the way, the squares pointed to by the k heads of M are read, and the information about which symbols these squares contain (which can be picked up as these are the symbols that lie to the right of $\triangle$ symbols on $\widetilde{t}$) is recorded in the state of $\widetilde{M}$.

2. Now that all k head symbols have been read, the machine $\widetilde{M}$ goes into a state that records what action M takes in this situation, as determined by the transition function δ. This information consists of: a destination state q', a k-tuple of write symbols, and a k-tuple of head displacements.

3. The tape $\widetilde{t}$ is now scanned from right to left. Each time a $\triangle$ symbol is encountered the appropriate action is taken that correctly encodes the action that M takes on the relevant tape. The actions may involve moving the head and/or overwriting a symbol. This is achieved by appropriately editing the tape $\widetilde{t}$ by overwriting symbols and by moving the $\triangle$ symbols. Sometimes, if a head needs to move over (or too close to) a $\|$ symbol, then a new square may need to be 'inserted' into the tape to create enough room.

4. Once the head arrives at the leftmost $\|$ symbol (this can again be detected because it will be the $(k+1)$-th such symbol encountered) the machine goes into state $\mathsf{now}_{q'}$, where q' is the destination state from step 2 above, which is accessible to $\widetilde{M}$ at this point, as it is recorded in the current state.

If you like such things, it is quite a fun exercise to work out the full details of the definition of $\widetilde{M}$ for yourself.

2 Undecidability and the Universal Turing Machine

As Turing machines have finite specifications, there is a countable infinity of Turing machines (up to a natural notion of equivalence of Turing machines). Since a Turing machine recognises exactly one language, only countably many languages can be recognised by a Turing machine. That is, there are countably many semidecidable languages. On the other hand, for any nonempty finite Σ, there are continuum-many (that is $2^{\aleph_0}$ -many) languages $L \subseteq \Sigma^*$. The cardinality difference between continuum-many and countably-many shows that there exist (continuum-many) different languages that cannot be recognised by any TM. Such languages are not semidecidable, and hence *a fortiori* undecidable (i.e., not decidable).

It is much more interesting to see that there exist languages that are semidecidable but not decidable. A major goal of todays lecture is to show that such languages exist. Thus decidability is in general a stronger property than semidecidability.

2.1 Encoding Turing machines

The specification of any particular Turing machine involves only a finite amount of information: a finite tape alphabet, a finite number of states, and a finite transition function. It is not hard to see that all this information can be captured in a single word over a suitable alphabet. The resulting word can then be given as input to another Turing machine. This provides a way of potentially passing one Turing machine as an input to another Turing machine.

We give an explicit encoding of a Turing machine $M = (\Gamma, Q, \delta)$ as a word over the alphabet

$$\Sigma_U \;=\; \{\, 0,\, 1,\, -1,\, [\,,\,]\,,\, \|\,,\, \bullet \,\}\ .$$

We do this for a TM M that is set up for language recognition (in particular it possesses accept and reject states), but the same idea works just as easily for Turing machines that have other *modi operandi*.

First, we encode each state $q \in Q$ by a chosen word

$$\langle q \rangle \;\in\; \{-1, 0, 1\}^l\ ,$$

for any chosen $l \geq \lceil \log_3(|Q|) \rceil$. (If one wishes to be economical one should choose $l = \lceil \log_3(|Q|) \rceil$.) In doing so we require that

$$\begin{aligned} \langle \mathsf{start} \rangle &= 0^l \quad := \quad \overbrace{0\ 0\ \ldots\ 0}^{l} \\ \langle \mathsf{accept} \rangle &= 1^l \\ \langle \mathsf{reject} \rangle &= (-1)^l \end{aligned}$$

Similarly, we choose an encoding of each character $a \in \Gamma$ as a string

$$\langle a \rangle \;\in\; \{-1, 0, 1\}^m\ ,$$

for any chosen $m \geq \lceil \log_3(|\Gamma|) \rceil$. In doing so we require: $\langle \Box \rangle = 0^m$.

For every *instruction*

$$\delta(q, a) \;=\; (q', b, d) \quad q, q' \in Q \quad a, b \in \Gamma \quad d \in \{-1, 0, 1\}\ ,$$

we encode the individual instruction as the word

$$[\langle q\rangle \bullet \langle a\rangle \parallel \langle q'\rangle \bullet \langle b\rangle \bullet d] \ \in \ \Sigma_U^* \ .$$

Note that this word always has length $2l + 2m + 7$.

Suppose that M's transition function δ is defined on exactly k distinct pairs (q, a). That is, the Turing machine can be defined by giving a list of k instructions. Then the encoding $\langle M\rangle$ of the entire TM is defined by:

$$\langle M\rangle \ = \ \langle \mathsf{start}\rangle \bullet \langle \square\rangle \ [\langle q_1\rangle \bullet \langle a_1\rangle \parallel \langle q_1'\rangle \bullet \langle b_1\rangle \bullet d_1] \ \ldots \ [\langle q_k\rangle \bullet \langle a_k\rangle \parallel \langle q_k'\rangle \bullet \langle b_k\rangle \bullet d_k]$$

Here, the string $\langle \mathsf{start}\rangle$ records (an upper bound on) the number of states, $\langle \square\rangle$ records (a bound on) the size of the encoded tape alphabet, and the remaining k components list all instructions needed to fully specify the transition function of M.

We can similarly (and more easily) encode any word $w = w_0 \ldots w_{n-1}$ over Γ as a word $\langle w\rangle$ over Σ_U:

$$\langle w\rangle \ := \ \langle w_0\rangle \bullet \cdots \bullet \langle w_{n-1}\rangle$$

2.2 An undecidable language

We can now define the language that we shall show to be semidecidable but not decidable.

$$L_{\mathsf{accept}} \ = \ \{\langle M\rangle \bullet \langle w\rangle \mid M \text{ is a TM with tape alphabet } \Gamma \supseteq \Sigma_U, \ w \in {\Sigma_U}^*, \text{ and } M \text{ accepts } w\} \ .$$

Theorem 2.1. *The language L_{accept} is undecidable.*

Proof. Suppose, for contradiction that L_{accept} is decidable, and let D be a TM with input alphabet Σ_U and tape alphabet $\Gamma \supseteq \Sigma_U$ that decides L_{accept}.

Define a new TM, N, with input alphabet Σ_U that runs as follows.

- N reads its input string v and converts it to the string $v \bullet \langle v\rangle$, resetting the head position to the left of the string.
- N then proceeds as the machine D, except that:
- if the machine D halts in the accept state then N halts in the reject state; and
- if the machine D halts in the reject state then N halts in the accept state.

We now consider the behaviour of the TM N when it is given an input string of the form $v = \langle M\rangle$ for some TM M. In this case, N proceeds as the machine D on input string $\langle M\rangle \bullet \langle\langle M\rangle\rangle$. Since D decides the language L_{accept}, the execution of D necessarily terminates, with final state:

- accept iff $\langle M\rangle \bullet \langle\langle M\rangle\rangle \in L_{\mathsf{accept}}$ iff M accepts $\langle M\rangle$; and
- reject iff $\langle M\rangle \bullet \langle\langle M\rangle\rangle \notin L_{\mathsf{accept}}$ iff M does not accept $\langle M\rangle$.

Thus the execution of N on $\langle M\rangle$ terminates, in final state:

- reject iff M accepts $\langle M\rangle$; and

- accept iff M does not accept $\langle M \rangle$.

In particular, if we run N with its own encoding $\langle N \rangle$ as input string then the last point above gives us that

- the execution of N on input string $\langle N \rangle$ terminates in the accept state iff N does not accept $\langle N \rangle$.

However, by definition of acceptance, N accepts $\langle N \rangle$ iff the execution of N on input $\langle N \rangle$ terminates in the accept state.

This gives the required contradiction. □

2.3 The universal Turing machine

In order to prove that the language L_{accept} is semidecidable, we require a fundamental construction, due to Turing: the construction of a *universal* Turing machine. For convenience, we shall define the universal machine as a 3-tape Turing machine. It can then be converted to a single-tape Turing machine by following the prescription in Lecture 1.

The idea of the universal machine U is as follows. Suppose M is any (single tape) Turing machine with tape alphabet Γ, and w is any word over $\Gamma - \{\Box\}$ then if we run the TM U on input string $\langle M \rangle \bullet \langle w \rangle$, the resulting execution will imitate the execution of the TM M on input string w. This is a *universal* Turing machine because it is able to simulate *every* Turing machine. (The restriction to single tape machines M is no limitation because of Lecture 1.)

We begin by preparing the ground for how U will imitate M. In order to achieve this, every machine configuration encountered during the execution of M needs to have a corresponding 3-tape machine configuration that will arise during the execution of U. To permit this, the TM U has a special state snapshot, which will record that U is currently in a configuration that encodes a configuration of M. In more detail, a configuration of U is said to *encode* a configuration (q, t, i) of M if:

- The current state of U is snapshot.
- Tape 1 of U contains $\langle M \rangle$, with its tape head pointing to the first character.
- Tape 2 contains $\langle q \rangle$, with its tape head pointing to the first character.
- Tape 3 contains $\langle w \rangle$, where $w \in \Gamma^*$ is a subword of the tape configuration t that includes all non-blank symbols of t. Tape head 3 must point to the first character of the occurrence of the substring $\langle t(i) \rangle$ that corresponds to the occurrence of the character $t(i)$ at position i of t, which the tape head of M is pointing to.

The lecture will discuss the construction of U (at a high level). In these notes, instead of giving the construction, we summarise the important properties of U in a sequence of lemmas leading to a theorem that characterises the behaviour of U.

Lemma 2.2 (Initial preparation)**.** *When U is executed from the* start *state on input string $\langle M \rangle \bullet \langle w \rangle$ on tape 1 (with the other two tapes empty) then, after finitely many steps of computation, U reaches a configuration that encodes the M-configuration* $(\mathsf{start}, w@0, 0)$*, moreover this is the first configuration in which U is in the* snapshot *state.*

Lemma 2.3 (Single-step simulation)**.** *Let C be a configuration of U that encodes a configuration (q, t, i) of M.*

- *If $(q, t, i) \to (q', t', i')$ in a single step of computation of M then $C \to^+ C'$ in many steps of computation of U, where C' encodes (q', t', i'), and no intermediate* snapshot *state arises between C and C'.*
- *If $C \to^+ C'$ in many steps of computation of U, where C' is in a* snapshot *state and no intermediate* snapshot *state arises between C and C', then $(q, t, i) \to (q', t', i')$ in a single step of computation of M where C' encodes (q', t', i').*

Lemma 2.4 (End computation)**.** *Suppose U is in a configuration that encodes a halting M-configuration (q, t, i).*

- *If q is* accept *then after finitely many steps of computation U terminates in the* accept *state.*
- *If q is* reject *then after finitely many steps of computation U terminates in the* reject *state.*
- *If q is any other state then after finitely many steps of computation U terminates in a state that is neither* accept *nor* reject*.*

Theorem 2.5 (Universal machine)**.** *There is a universal Turing machine U with the following behaviour. Suppose M is any Turing machine with tape alphabet Γ, and w is any word over $\Gamma - \{\Box\}$. If we run U on the input string $\langle M \rangle \bullet \langle w \rangle$, then the resulting execution of U enjoys the following properties.*

- *It terminates if and only if M terminates on input string w.*
- *It terminates in the* accept *state if and only if M accepts w.*
- *It terminates in the* reject *state if and only if M rejects w.*

Proof. This is just a matter of piecing together the lemmas above. □

2.4 The semidecidability of L_{accept}

Theorem 2.6. *The language L_{accept} is semidecidable.*

Proof. Construct a TM S that does the following.

- First it reads the input string $v \in \Sigma_U^*$ and checks if this is of the form $\langle M \rangle \bullet \langle w \rangle$ for some Turing machine M with tape alphabet Γ and word $w \in (\Gamma - \{\Box\})^*$.
- If v is not of the form $\langle M \rangle \bullet \langle w \rangle$ then S rejects v. (We could equally well allow it to go into a loop.)
- If v is of form $\langle M \rangle \bullet \langle w \rangle$ then S proceeds as the universal machine U on input string v.

Clearly S accepts v if and only if $v \in L_{\mathsf{accept}}$.

□

3 Representations

In Lecture 1, we defined what it means for a TM to compute a (partial) function on words over an alphabet Σ. It is straightforward to generalise the definition to (partial) functions that map words over one input alphabet Σ_1 to words over a potentially different output alphabet Σ_2. Turing machines computing such functions are assumed to come with:

- distinguished *input* and *output* alphabets Σ_1 and Σ_2 respectively with $\Sigma_1 \cup \Sigma_2 \subseteq \Gamma - \{\Box\}$,
- and a distinguished halting state $\mathsf{halt} \in Q$.

Definition 3.1. Such a TM is said to *compute* a partial function $f\colon \Sigma_1^* \rightharpoonup \Sigma_2^*$ if:

- for all $x \in \mathsf{dom}(f)$, it holds that $x@0 \Downarrow (\mathsf{halt}, f(x)@i, i)$ for some $i \in \mathbb{Z}$; and
- for all $x \notin \mathsf{dom}(f)$, it is not the case that $x@0 \Downarrow (\mathsf{halt}, t, i)$, for any t, i.

Definition 3.2. A partial function $f\colon \Sigma_1^* \rightharpoonup \Sigma_2^*$ is said to be *computable* if there exists some TM that computes it.

For any alphabet Σ, it is trivial that the identity function $x \mapsto x$ on Σ^* is computable. It also holds that computable partial functions are closed under composition.

Before stating this formally, we consider how to define the composition $g \circ f\colon X \rightharpoonup Z$ of two partial functions $f\colon X \rightharpoonup Y$ and $g\colon Y \rightharpoonup Z$. As expected, this composition has the action $x \mapsto g(f(x))$. The point to pay attention to is that $(g \circ f)(x)\!\downarrow$ only if both $f(x)\!\downarrow$ and $g(f(x))\!\downarrow$. In the case that $f(x)\!\uparrow$, we do not give a value to the expression $g(f(x))$. On occasion, a little care is needed with this point. For example, if g is a constant function (such as $g(y) = 0$, for all y) then nonetheless $g(f(x))$ is considered as undefined on values x for which $f(x)\!\uparrow$.

Proposition 3.3. *Let $f\colon \Sigma_1^* \rightharpoonup \Sigma_2^*$ and $g\colon \Sigma_2^* \rightharpoonup \Sigma_3^*$ be computable. Then the composite partial function $g \circ f\colon \Sigma_1^* \rightharpoonup \Sigma_3^*$ is computable.*

Proof. Suppose we have TMs M computing f and M' computing g. Define a TM for $g \circ f$ as follows. First run M on the input. If the execution of M terminates in halt then continue from the current tape-head configuration by running M'. □

The above development is all well and good if one is only concerned with computing *unary* (that is, single argument) functions on *words*. However, it is frequently useful to compute other forms of function; for example, functions of several arguments, and functions on forms of data that are not directly given as words. Let's start with the second of these points.

3.1 Computing functions on $\mathbb{N}$

Functions on the natural numbers $\mathbb{N}$ play a fundamental role in mathematics. There is an obvious approach to computing such functions using Turing machines, encode natural numbers as words.

One natural encoding is to represent numbers as binary strings, using the *binary alphabet* $\Sigma_b := \{0, 1\}$. One way of doing this is to encode each number $n \in \mathbb{N}$ uniquely as the word $\mathsf{bin}(n) \in \Sigma_b^*$ defined as the the binary representation of n with no leading 0s (so $\mathsf{bin}(0) = \varepsilon$).

This encoding leads to an obvious definition of computable (partial) function on $\mathbb{N}$.

Definition 3.4 (Computing a partial function on $\mathbb{N}$)**.** A TM is said to *compute a partial function* $f\colon \mathbb{N} \rightharpoonup \mathbb{N}$ if it computes a partial function $g\colon \Sigma_b^* \rightharpoonup \Sigma_b^*$ satisfying:

- for all $n \in \mathsf{dom}(f)$, it holds that $g(\mathsf{bin}(n))\downarrow$ and $g(\mathsf{bin}(n)) = \mathsf{bin}(f(n))$, and
- for all $n \in \mathbb{N} - \mathsf{dom}(f)$, it holds that $g(\mathsf{bin}(n))\uparrow$.

(When the above conditions are satisfied we say that g *realises* f.)

Notice that the above definition leaves unspecified how g behaves on words whose first character is 0. For this reason, the function g on words is not uniquely determined by the function f on $\mathbb{N}$. In the other direction, however, the function g does uniquely determine the function f. That is, any computable partial function $g\colon \Sigma_b^* \rightharpoonup \Sigma_b^*$ on words realises at most one partial function $f\colon \mathbb{N} \rightharpoonup \mathbb{N}$.

Definition 3.5 (Computable partial function on $\mathbb{N}$)**.** A partial function $f\colon \mathbb{N} \rightharpoonup \mathbb{N}$ is said to be *computable* if there exists some TM that computes it (equivalently if there exists a computable partial function $g\colon \Sigma_b^* \rightharpoonup \Sigma_b^*$ that realises f).

3.2 Representations

There is some arbitrariness in the choice of encoding of natural numbers used above. A second possibility, for example, would be to encode natural numbers non-uniquely, by allowing leading 0s and considering every word $w \in \Sigma_b^*$ as representing a corresponding natural number in binary notation. So, for example, the strings 11, 011, 0011 would be three different representations of the number 3. Similarly, ε, 0, 00 would be three different representations of the number 0.

A third (albeit less efficient) possibility is to forego binary representations and encode numbers as strings over the *unary alphabet* $\Sigma_u := \{0\}$. In this encoding, each number n is encoded uniquely as the word $0^n \in \Sigma_u^*$.

Still further possibilities arise by using larger alphabets (e.g., the alphabet $\{0, \ldots, 9\}$ of decimal digits) and using base N notation (with leading 0s or without) where $N > 2$.

It turns out that any of the above approaches to encoding natural numbers can be used to define the notion of computable partial function on $\mathbb{N}$. Moreover, all choices give rise to the same robust notion of computability. We illustrate this, by comparing the three encodings discussed above in detail: binary representation without leading 0s, binary representation allowing leading 0s, and unary representation. Each of these three encodings has the following ingredients in common.

- There is a representing alphabet Σ.
- There is a *surjective partial function* $\gamma\colon \Sigma^* \twoheadrightarrow \mathbb{N}$, whose domain $\mathsf{dom}(\gamma)$ is the set of words encoding numbers, and whose action is to map every such word to the unique number it represents. (Here the notation $\twoheadrightarrow$ indicates a surjective partial function. Recall that surjectivity means that, for every $x \in X$, there exists $w \in \Sigma^*$ with $\gamma(w) = x$.)

We explicitly exhibit the representation functions $\gamma\colon \Sigma^* \twoheadrightarrow \mathbb{N}$ that arise in each of the three encodings of $\mathbb{N}$ under consideration. In the case of the first encoding (binary representation

with no leading 0s), the alphabet is Σ_b and the surjective partial function $\gamma_{\mathbb{N}} \colon \Sigma_b^* \twoheadrightarrow \mathbb{N}$ is:

$$\mathsf{dom}(\gamma_{\mathbb{N}}) \;:=\; \{\varepsilon\} \cup \{1w \mid w \in \Sigma_b^*\}$$

$$\gamma_{\mathbb{N}} \;:\; w \in \mathsf{dom}(\gamma_{\mathbb{N}}) \;\mapsto\; \sum_{i=0}^{|w|-1} w_{|w|-i-1} \cdot 2^i$$

For the second encoding (binary representations allowing leading 0s), the alphabet is Σ_b and the surjective partial function $\gamma'_{\mathbb{N}} \colon \Sigma_b^* \twoheadrightarrow \mathbb{N}$ is defined by:

$$\mathsf{dom}(\gamma'_{\mathbb{N}}) \;:=\; \Sigma_b^*$$

$$\gamma'_{\mathbb{N}} \;:\; w \in \mathsf{dom}(\gamma'_{\mathbb{N}}) \;\mapsto\; \sum_{i=0}^{|w|-1} w_{|w|-i-1} \cdot 2^i$$

For the third encoding (unary representation), the alphabet is Σ_u and the surjective partial function $\gamma''_{\mathbb{N}} \colon \Sigma_u^* \twoheadrightarrow \mathbb{N}$ is defined by:

$$\mathsf{dom}(\gamma''_{\mathbb{N}}) \;:=\; \Sigma_u^*$$

$$\gamma''_{\mathbb{N}} \;:\; w \in \mathsf{dom}(\gamma''_{\mathbb{N}}) \;\mapsto\; |w|$$

Notice that in the second and third encodings the representing functions are total. Also, in the first and third encodings, the representing functions are injective: every natural number has a unique representation.

We shall see that one can define what it means for a (partial) function on $\mathbb{N}$ to be computable, whenever one has a representation of $\mathbb{N}$ of the general form above. This definition is not restricted to (partial) functions on the natural numbers. It makes sense for (partial) functions between any two sets equipped with a *representation*.

Definition 3.6 (Representation)**.** A *representation* of a set X by words over an alphabet Σ is a surjective partial function $\gamma \colon \Sigma^* \twoheadrightarrow X$.

When $\Sigma \neq \emptyset$, the set Σ^* has countably infinite cardinality (i.e., cardinality $\aleph_0$). Thus any represented set X is necessarily countable. (Finite sets are considered as being countable.)

Representations can be specified as triples (X, Σ, γ). However, we often simply refer to γ, leaving X and Σ implicit. Given a representation $\gamma \colon \Sigma^* \twoheadrightarrow X$, we call any word w such that $\gamma(w) = x$ a *name* (or *realiser*) for x.

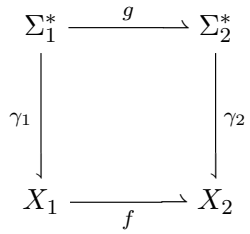
Definition 3.7 (Computable partial function)**.** Given representations $\gamma_1 \colon \Sigma_1^* \twoheadrightarrow X_1$ and $\gamma_2 \colon \Sigma_2^* \twoheadrightarrow X_2$, we say that a partial function $f \colon X_1 \rightharpoonup X_2$ is $(\gamma_1 \to \gamma_2)$*-computable* if there exists a computable partial function $g \colon \Sigma_1^* \rightharpoonup \Sigma_2^*$ such that, for every $x \in X_1$ and γ_1-name w for x,

- $g(w)\!\downarrow$ if and only if $f(x)\!\downarrow$, and
- $g(w)\!\downarrow$ implies $g(w)$ is a γ_2-name for $f(x)$.

(Equivalently, for any $w \in \mathsf{dom}(\gamma_1)$, (i) $\gamma_2(g(w)) \simeq f(\gamma_1(w))$, and (ii) $g(w)\!\downarrow \Rightarrow g(w) \in \mathsf{dom}(\gamma_2)$.)

A partial function $g \colon \Sigma_1^* \rightharpoonup \Sigma_2^*$ satisfying the conditions above is said to be a *realiser* for the function f.

If f is $(\gamma_1 \to \gamma_2)$-computable and g realises f then the following diagram of partial functions commutes, for all $w \in \mathsf{dom}(\gamma_1)$.



For total functions f the definition of computability is equivalent to the commutativity of the above diagram for all $p \in \mathsf{dom}(\gamma_1)$. For a partial function f, the definition of computability is stronger than commutativity. (**Exercise:** why is this?)

Observe that there may be many g realising the same f. However, for any partial function $g\colon \Sigma_1^* \rightharpoonup \Sigma_2^*$, there is at most one partial function $f\colon X_1 \to X_2$ realised by g. That is, realisers determine the functions they realise. This is conceptually natural. Realisers are TM-computable functions considered as computing abstract functions between represented sets. It seems natural that a real-world computation on words should determine the set-theoretic function between represented sets that it computes.

As examples, we can instantiate the above definition using our three different representations of $\mathbb{N}$ defined earlier, namely $\gamma_{\mathbb{N}}\colon \Sigma_b^* \twoheadrightarrow \mathbb{N}$, $\gamma'_{\mathbb{N}}\colon \Sigma_b^* \twoheadrightarrow \mathbb{N}$ and $\gamma''_{\mathbb{N}}\colon \Sigma_u^* \twoheadrightarrow \mathbb{N}$. This gives three potential redefinitions of what it means for a partial function $f\colon \mathbb{N} \rightharpoonup \mathbb{N}$ to be computable namely that it is $(\gamma_{\mathbb{N}} \to \gamma_{\mathbb{N}})$-computable, that it is $(\gamma'_{\mathbb{N}} \to \gamma'_{\mathbb{N}})$-computable, or that it is $(\gamma''_{\mathbb{N}} \to \gamma''_{\mathbb{N}})$-computable. Fortunately, all three redefinitions give rise to the same notion of computability. Moreover, all three coincide with Definition 3.5.

One could also consider what notions of computability one obtains for partial functions on $\mathbb{N}$ if one uses different representation of $\mathbb{N}$ for the domain and codomain; for example, asking what the $(\gamma'_{\mathbb{N}} \to \gamma''_{\mathbb{N}})$-computable functions are. Again, in every case, one obtains the notion defined in Definition 3.5. This notion thus proves to be very robust.

The above claims are proved in the following way. First, one proves that $(\gamma_{\mathbb{N}} \to \gamma_{\mathbb{N}})$-computability is literally a reformulation of Definition 3.5. (**Exercise:** verify this.) One then proves that the representations $\gamma_{\mathbb{N}}$, $\gamma'_{\mathbb{N}}$ and $\gamma''_{\mathbb{N}}$ are all equivalent in the sense defined in the next section. The coincidence of notions of computability across all combinations of such representations is then a consequence of Proposition 3.10 below.

3.3 Composition and equivalence

We consider some generalities about represented sets and computable functions between them. For any representation $\gamma\colon \Sigma^* \twoheadrightarrow X$, it is trivial that the identity function $x \mapsto x$ on X is $(\gamma \to \gamma)$-computable. It also holds that the notion of computable partial function is closed under composition.

Proposition 3.8. *Suppose $\gamma_1\colon \Sigma_1^* \twoheadrightarrow X_1$, $\gamma_2\colon \Sigma_2^* \twoheadrightarrow X_2$ and $\gamma_3\colon \Sigma_3^* \twoheadrightarrow X_3$ are representations,. Let $f_1\colon X_1 \rightharpoonup X_2$ and $f_2\colon X_2 \rightharpoonup X_3$ be $(\gamma_1 \to \gamma_2)$- and $(\gamma_2 \to \gamma_3)$-computable respectively. Then the composite partial function $f_2 \circ f_1\colon X_1 \rightharpoonup X_3$ is $(\gamma_1 \to \gamma_3)$-computable*

The proof of the above result is routine in the sense that the proof falls out from the definitions. Nonetheless, since the definition of computable partial function is somewhat involved there

is a line of argument that needs to be followed. Since we are still at an early point in the course, we write out this argument in detail.

Proof. Suppose that $g_1 \colon \Sigma_1^* \rightharpoonup \Sigma_2^*$ realises f_1 and $g_2 \colon \Sigma_2^* \rightharpoonup \Sigma_3^*$ realises f_2. The composite function $g_2 \circ g_1 \colon \Sigma_1^* \rightharpoonup \Sigma_3^*$ is computable by Proposition 3.3. We show that $g_2 \circ g_1$ realises $f_2 \circ f_1$. We need to show that, for all $w \in \mathsf{dom}(\gamma_1)$, (i) $\gamma_3((g_2 \circ g_1)(w)) \simeq (f_2 \circ f_1)(\gamma_1(w))$, and (ii) $(g_2 \circ g_1)(w)\!\downarrow \Rightarrow (g_2 \circ g_1)(w) \in \mathsf{dom}(\gamma_3)$.

For (i), we have

$$\gamma_3(g_2(g_1(w))) \;\simeq\; f_2(\gamma_2(g_1(w))) \;\simeq\; f_2(f_1(\gamma_1(w))) \;.$$

The second Kleene equality holds because g_1 realises f_1, hence $\gamma_2(g_1(w)) \simeq f_1(\gamma_1(w))$. Similarly g_2 realises f_2, so $\gamma_3(g_2(v)) \simeq f_2(\gamma_2(v))$, for all $v \in \mathsf{dom}(\gamma_2)$. This gives the first Kleene equality in the case that $g_1(w) \in \mathsf{dom}(\gamma_2)$, which holds whenever $g_1(w)\!\downarrow$ (because $w \in \mathsf{dom}(\gamma_1)$ and g_1 realises f_1). The first Kleene equality also holds in the remaining case that $g_1(w)\!\uparrow$, since both sides of it are then undefined.

For (ii), suppose $g_2(g_1(w))\!\downarrow$. Then $g_1(w)\!\downarrow$ so $g(w) \in \mathsf{dom}(\gamma_2)$, because $w \in \mathsf{dom}(\gamma_1)$ and g_1 realises f_1. Since $g(w) \in \mathsf{dom}(\gamma_2)$ and $g_2(g_1(w))\!\downarrow$, it holds that $g_2(g_1(w)) \in \mathsf{dom}(\gamma_3)$, because g_2 realises f_2. □

Definition 3.9 (Equivalent representations)**.** Two representations $\gamma_1 \colon \Sigma_1^* \twoheadrightarrow X$ and $\gamma_2 \colon \Sigma_2^* \twoheadrightarrow X$ of the same set X are said to be *equivalent* if the identity function $x \mapsto x$ on X is both $(\gamma_1 \to \gamma_2)$- and $(\gamma_2 \to \gamma_1)$-computable.

Proposition 3.10. *Suppose that $\gamma_1 \colon \Sigma_1^* \twoheadrightarrow X_1$ and $\gamma_1' \colon (\Sigma_1')^* \twoheadrightarrow X_1$ are equivalent representations and that $\gamma_2 \colon \Sigma_2^* \twoheadrightarrow X_2$ and $\gamma_2' \colon (\Sigma_2')^* \twoheadrightarrow X_2$ are also equivalent. Then a partial function $X_1 \rightharpoonup X_2$ is $(\gamma_1 \to \gamma_2)$-computable if and only if it is $(\gamma_1' \to \gamma_2')$-computable.*

Proof. Suppose $f \colon X_1 \rightharpoonup X_2$ is $(\gamma_1 \to \gamma_2)$-computable. By equivalence, the identity functions 1_{X_1} and 1_{X_2} are $(\gamma_1' \to \gamma_1)$- and $(\gamma_2 \to \gamma_2')$-computable respectively. By Proposition 3.8, the composite $1_{X_2} \circ f \circ 1_{X_1}$ is $(\gamma_1' \to \gamma_2')$-computable. That is, f is $(\gamma_1' \to \gamma_2')$-computable. The converse, that $(\gamma_1' \to \gamma_2')$-computability implies $(\gamma_1 \to \gamma_2)$-computability is proved similarly. □

Proposition 3.11. *The three representations $\gamma_{\mathbb{N}}$, $\gamma_{\mathbb{N}}'$ and $\gamma_{\mathbb{N}}''$ of $\mathbb{N}$ defined above are equivalent.*

The proof is left as an **exercise**.

3.4 Product representations

At the start of the lecture, we raised the question of computability of functions of several arguments. One way of defining computability for functions taking k arguments, say, would be to use a Turing machine with at least k tapes, and to ask for each argument of the function to be given as input on a separate tape. There is nothing technically wrong with this, but there is a more useful and flexible alternative that does not have such a dependency on the underlying TM model. Moreover, the alternative gives rise to the same notion of computability.

Given k represented sets

$$\gamma_1 \colon \Sigma_1^* \twoheadrightarrow X_1 \quad \ldots \quad \gamma_k \colon \Sigma_k^* \twoheadrightarrow X_k$$

we define a *product representation*

$$\gamma\colon \Sigma^* \twoheadrightarrow (X_1 \times \cdots \times X_k)$$

as follows.

$$\begin{aligned} \Sigma &:= (\Sigma_1 \cup \cdots \cup \Sigma_k) \uplus \{\text{`,'}\} \\ \mathsf{dom}(\gamma) &:= \{w_1\,\text{,}\,\ldots\,\text{,}\,w_k \mid w_1 \in \mathsf{dom}(\gamma_1) \text{ and } \ldots \text{ and } w_k \in \mathsf{dom}(\gamma_k)\} \\ \gamma(w_1\,\text{,}\,\ldots\,\text{,}\,w_k) &:= (\gamma_1(w_1), \ldots, \gamma_k(w_k)) \end{aligned}$$

Here the $\uplus$ symbol is a *disjoint union*: the comma symbol is chosen so that it does not appear in any of the alphabets $\Sigma_1, \ldots, \Sigma_k$. An element $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ is then represented as tuple of words $w_1\,\text{,}\,\ldots\,\text{,}\,w_k$ separated by the comma symbol, where each word w_i is a name for x_i. The requirement that the comma symbol does not appear in the Σ_i alphabets ensures that the word $w_1\,\text{,}\,\ldots\,\text{,}\,w_k$ can only be parsed as a k-tuple in one way.

We shall usually write the product representation as

$$\gamma_1 \times \cdots \times \gamma_k \;:\; \Sigma^*_{\gamma_1 \times \cdots \times \gamma_k} \twoheadrightarrow (X_1 \times \cdots \times X_k)$$

to have a more precise notation for its components.

Given representations $\gamma_1\colon \Sigma_1^* \twoheadrightarrow X_1$, …, $\gamma_k\colon \Sigma_k^* \twoheadrightarrow X_k$ and $\gamma\colon \Sigma^* \twoheadrightarrow X$, we consider a partial function $f\colon X_1 \times \cdots \times X_k \rightharpoonup X$ to be *computable* with respect to the representations if it is $((\gamma_1 \times \cdots \times \gamma_k) \to \gamma)$-computable. We usually abbreviate this to $(\gamma_1, \ldots, \gamma_k \to \gamma)$-*computable*.

Very often, we shall be in the situation that we are concerned with computability on a set for which we have a standard representation. For example, we take the representation $\gamma_{\mathbb{N}}$ defined above as our standard representation on $\mathbb{N}$. In such cases, we omit explicit reference to representations when we talk about computability of multi-argument functions on the set in question. The definition below illustrates this point.

Definition 3.12. A partial function $f\colon \mathbb{N}^k \rightharpoonup \mathbb{N}$ is *computable* if it is $(\gamma_{\mathbb{N}}, \ldots, \gamma_{\mathbb{N}} \to \gamma_{\mathbb{N}})$-computable (where there are of course k-occurrences of $\gamma_{\mathbb{N}}$ on the left side of the arrow).

As might be expected, the notion does not change if the standard representation $\gamma_{\mathbb{N}}$ is replaced by any of our other equivalent representations of $\mathbb{N}$. To prove this, one shows that the product construction on representations preserves equivalence of representations (**exercise**).

3.5 Decidable and semidecidable subsets

Definition 3.13 (Demidecidable and semidecidable subset)**.** Let $\gamma\colon \Sigma^* \twoheadrightarrow X$ be a representation. A subset $Z \subseteq X$ is said to be γ-*decidable* if there exists a language-recognition TM M that satisfies, for all $w \in \mathsf{dom}(\gamma)$,

- if $\gamma(w) \in Z$ then M accepts w; and
- if $\gamma(w) \notin Z$ then M rejects w.

Such a TM M is said to γ-*decide* Z

A subset $Z \subseteq X$ is said to be γ-*semidecidable* if there exists a language-recognition TM M that satisfies, for all $w \in \mathsf{dom}(\gamma)$,

- M accepts $w \;\Leftrightarrow\; \gamma(w) \in Z$.

Such a TM M is said to *γ-semidecide Z*

Note that the above definition only constrains the behaviour of the deciding/semideciding TMs on inputs $w \in \mathsf{dom}(\gamma)$. For example, a deciding TM is not required to halt on input words $w \in \Sigma^* - \mathsf{dom}(\gamma)$.

Normally, we shall utilise the above concepts in contexts in which the given representation is understood, and accordingly we shall simply refer to *decidable* and *semidecidable* subsets without explicitly mentioning γ. Nonetheless, for the remainder of this section we shall continue to make representations explicit, in order to minimise any scope for ambiguity while we develop the core material on representations.

We write $\mathbb{B} := \{0,1\}$ for the set of bits (or booleans). We give this a standard representation $\gamma_{\mathbb{B}} \colon \Sigma_b^* \rightharpoonup\!\!\!\!\!\rightarrow \mathbb{B}$:

$$\mathsf{dom}(\gamma_{\mathbb{B}}) \;:=\; \{0,1\}$$

$$\gamma_{\mathbb{B}} \;:\; w \in \mathsf{dom}(\gamma_{\mathbb{B}}) \;\mapsto\; w$$

For any subset $Z \subseteq X$, we define its *characteristic function* $\chi_Z \colon X \to \mathbb{B}$ and *partial-characteristic function* $\chi_Z^p \colon X \rightharpoonup \mathbb{B}$.

$$\chi_Z(x) \;=\; \begin{cases} 1 & \text{if } x \in Z \\ 0 & \text{otherwise} \end{cases} \qquad\qquad \chi_Z^p(x) \;\simeq\; \begin{cases} 1 & \text{if } x \in Z \\ \uparrow & \text{otherwise} \end{cases}$$

Proposition 3.14. *Let $\gamma \colon \Sigma^* \rightharpoonup\!\!\!\!\!\rightarrow X$ be a representation. For any subset $Z \subseteq X$ it holds that:*

1. *Z is γ-decidable if and only if its characteristic function $\chi_Z \colon X \to \mathbb{B}$ is $(\gamma \to \gamma_{\mathbb{B}})$-computable.*
2. *Z is γ-semidecidable if and only if its partial characteristic function $\chi_Z^p \colon X \rightharpoonup \mathbb{B}$ is $(\gamma \to \gamma_{\mathbb{B}})$-computable.*

Proof. We prove statement (1), leaving the argument for statement (2) as an **exercise**.

For the left-to-right implication, suppose that Z is γ-decidable, and let M be a TM that decides it. We convert the language-recognition TM M to a function-type TM M' that behaves as follows. M' first behaves as M on the input word w. In the case that the execution of M terminates in accept, the machine M' continues by erasing the tape, writing 1 on it and terminating in halt. If the execution of M terminates in reject, then M' erases the tape, writes 0 on it and terminates in halt. Let $g \colon \Sigma^* \rightharpoonup \Sigma_b^*$ be the partial function computed by M'. We claim that g realises χ_Z. Indeed, for any input word $w \in \mathsf{dom}(\gamma)$, we have:

- if $\gamma(w) \in Z$ then M accepts w, so $g(w) = 1 = \chi_Z(\gamma(w))$, and
- if $\gamma(w) \notin Z$ then M rejects w, so $g(w) = 0 = \chi_Z(\gamma(w))$.

Conversely, suppose that χ_Z is computable. Let M be a function-type TM computing a realiser $g \colon \Sigma^* \rightharpoonup \Sigma_b^*$ for χ_Z. We define a language-recognition TM M' as follows. M' first behaves as M on the input word w. In the case that the execution of M on w terminates in halt with the output word 1 on the tape, M' halts in the accept state. In the case that M on w terminates in halt with 0 on the tape, M' halts in the reject state. One now easily verifies that M' decides Z (**exercise**). □

4 Computable Partial Functions

In Note 3, we defined what it means for a partial function $f\colon \mathbb{N}^k \rightharpoonup \mathbb{N}$ to be computable (Definition 3.12). We call the collection

$$\{f\colon \mathbb{N}^k \rightharpoonup \mathbb{N} \mid f \text{ is computable}\}_{k\geq 0}$$

the *computable partial functions* (also known as the *partial computable functions*) on $\mathbb{N}$.

(**Question:** What are the computable partial functions $\mathbb{N}^k \rightharpoonup \mathbb{N}$ in the case $k = 0$? In the literature, this rather trivial case is often excluded from the definition. We include it because it permits a slightly more elegant treatment in Section 4.1 below.)

The notion of computable partial function is one of fundamental mathematical importance. It will pervade the rest of this course.

4.1 A mathematical characterisation of the computable partial functions

The main task today is to develop an alternative characterisation of the computable partial functions, one that is purely mathematical in nature.

Definition 4.1 (primitive recursive functions)**.** The *primitive recursive functions* are defined to be the smallest collection

$$\mathcal{F} \;\subseteq\; \{\mathbb{N}^k \rightharpoonup \mathbb{N}\}_{k\geq 0}$$

of partial functions that satisfies properties 1–3 below.

1. *(Basic functions)*

   $\mathcal{F}$ contains the *zero* and *successor* functions $\mathsf{Z}\colon \mathbb{N}^0 \to \mathbb{N}$ and $\mathsf{S}\colon \mathbb{N} \to \mathbb{N}$ defined by:

   $$\begin{aligned} \mathsf{Z}() &= 0 \ , \\ \mathsf{S}(x) &= x+1 \ . \end{aligned}$$

   Also, for each $k \geq 1$ and $1 \leq i \leq k$, $\mathcal{F}$ contains the *projection* function $\mathsf{U}^k_i\colon \mathbb{N}^k \to \mathbb{N}$ defined by:

   $$\mathsf{U}^k_i(x_1, \ldots, x_k) \;=\; x_i \ .$$

2. *(Composition)*

   If $f\colon \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $g_1, \ldots, g_k\colon \mathbb{N}^l \rightharpoonup \mathbb{N}$ are in $\mathcal{F}$ then so is the *composition*

   $$f \circ (g_1, \ldots, g_k)\colon \mathbb{N}^l \rightharpoonup \mathbb{N} \ ,$$

   defined by

   $$f \circ (g_1, \ldots, g_k)\,(x_1, \ldots, x_l) \;\simeq\; f(g_1(x_1, \ldots, x_l), \ldots, g_k(x_1, \ldots, x_l)) \ .$$

3. *(Primitive recursion)*

   If $f\colon \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $g\colon \mathbb{N}^{k+2} \rightharpoonup \mathbb{N}$ are in $\mathcal{F}$ then so is the partial function

   $$\mathsf{R}fg\colon \mathbb{N}^{k+1} \rightharpoonup \mathbb{N} \ ,$$

defined by

$$\begin{aligned} \mathsf{R}fg\,(x_1,\ldots,x_k,0) &\simeq f(x_1,\ldots,x_k) \\ \mathsf{R}fg\,(x_1,\ldots,x_k,x+1) &\simeq g(x_1,\ldots,x_k,x,\mathsf{R}fg\,(x_1,\ldots,x_k,x)) \ . \end{aligned}$$

The function $\mathsf{R}fg$ is said to be defined by *primitive recursion* from f and g.

Since the basic functions in 1 are total, and conditions 2 and 3 obviously preserve totality, we have the result below.

Proposition 4.2. *Every primitive recursive function is total.*

Definition 4.3 (Partial recursive functions)**.** The *partial recursive functions* are defined to be the smallest collection

$$\mathcal{F} \ \subseteq \ \{\mathbb{N}^k \rightharpoonup \mathbb{N}\}_{k\geq 0}$$

of partial functions that satisfies properties 1–3 above and also property 4 below.

4. *(Minimisation)*
   If $f\colon \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ is in $\mathcal{F}$ then so is $\mu f\colon \mathbb{N}^k \rightharpoonup \mathbb{N}$ defined by:

$$\mu f\,(x_1,\ldots,x_k) \ \simeq \ \begin{cases} \text{the smallest } n \text{ such that:} & \\ \quad f(x_1,\ldots,x_k,n)=0, \text{ and} & \\ \quad f(x_1,\ldots,x_k,i) \text{ is defined whenever } 0\leq i<n & \text{if such } n \text{ exists} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The theorem below justifies the above definition.

Theorem 4.4 (The characterisation theorem)**.** *The partial recursive functions are exactly the computable partial functions.*

This significant result gives a characterisation of the computable partial functions that has an entirely different flavour from the Turing-machine-based definition of what it means to be a computable partial function.

4.2 The computability of partial recursive functions

This section proves the easier direction of Theorem 4.4, namely that every partial recursive function is computable. Since the partial recursive functions are defined as the smallest class of functions closed under 1–4, it is enough to show that the computable partial functions are closed under 1–4.

Firstly, it is easy to see that the computable partial functions include the basic functions required by condition 1.

For closure under conditions 2–4, it is easier to use a variant characterisation of computable k-ary partial function using a multi-tape TM. We state this as a proposition.

Proposition 4.5. *A k-ary partial function $f\colon \mathbb{N}^k \to \mathbb{N}$ is computable if and only if there exists a $(k+1)$-tape TM M with the following behaviour.*

If M is run in a starting configuration with $\mathsf{bin}(x_1)$ on tape 1 and ... and $\mathsf{bin}(x_k)$ on tape k with tape $k+1$ blank then it reaches a halt *configuration with $y \in \{0,1\}^*$ on tape $k+1$ if and only if $y = \mathsf{bin}(f(x_1, \ldots, x_k))$. Moreover tapes 1–k are left in their starting configuration.*

This result is once again shown by giving a transformation between the relevant kinds of TM, using similar ideas to Note 1.

Using the above characterisation, it will be shown in the lecture that the computable partial functions are closed under primitive recursion (condition 3). Closure under composition (condition 2) and minimisation (condition 4) are left as an exercise.

Putting all this together we have shown the following.

Proposition 4.6. *Every partial recursive function is computable.*

4.3 Encoding data as numbers

The general idea is that discrete data structures can always be encoded as natural numbers, and computation with such data can be mimicked by computation on encodings.

As example such encodings, we encode pairs $\mathbb{N} \times \mathbb{N}$ of natural numbers and finite sequences $\mathbb{N}^*$ of natural numbers.

To encode pairs, we define $p \colon \mathbb{N}^2 \to \mathbb{N}$ by

$$p(x,y) \;=\; \frac{1}{2}(x+y)(x+y+1)+x \;\;.$$

The following hold.

- The *pairing* function $p \colon \mathbb{N}^2 \to \mathbb{N}$ is a bijection from $\mathbb{N}^2$ to $\mathbb{N}$.
- p is primitive recursive.
- The *projection* functions $q_1, q_2 \colon \mathbb{N} \to \mathbb{N}$ defined (implicitly) by

  $$q_1(p(x,y)) \;=\; x \qquad q_2(p(x,y)) \;=\; y \;\;,$$

  are primitive recursive.

To encode finite sequences, we define $\ulcorner \cdot \urcorner : \mathbb{N}^* \to \mathbb{N}$ by:

$$\ulcorner n_0 \ldots n_{k-1} \urcorner \;=\; 2^{n_0} + 2^{n_0+n_1+1} + 2^{n_0+n_1+n_2+2} + \cdots + 2^{n_0+n_1+\cdots+n_{k-1}+k-1} \;\;.$$

That is, by:

$$\ulcorner w \urcorner \;=\; \sum_{i=1}^{|w|} 2^{\left(\sum_{j=1}^{i} w_{j-1}\right)+i-1} \;\;.$$

The following properties hold.

- $\ulcorner \cdot \urcorner \colon \mathbb{N}^* \to \mathbb{N}$ is a bijection from $\mathbb{N}^*$ to $\mathbb{N}$.
- $\ulcorner \cdot \urcorner \colon \mathbb{N}^* \to \mathbb{N}$ is computable.[1] (**Exercise:** define what this means, by giving $\mathbb{N}^*$ a suitable representation.)

[1] This fact is not actually used in the proof below, but is anyway worth noting.

- The functions $\sigma\colon \mathbb{N}^2 \to \mathbb{N}$ and $l\colon \mathbb{N} \to \mathbb{N}$ defined below are primitive recursive.

$$\begin{aligned} \sigma(\ulcorner w \urcorner, i) &= \begin{cases} 0 & \text{if } i \geq |w| \\ w_i + 1 & \text{if } i < |w| \end{cases} \\ l(\ulcorner w \urcorner) &= |w| \ . \end{aligned}$$

4.4 Every computable partial function is partial recursive

We complete Theorem 4.4 with an outline proof of the remaining inclusion.

Proposition 4.7. *Every computable partial function is partial recursive.*

We make direct use the representation-based definition of k-ary computable partial function (Definition 3.12). Suppose then that M is a single tape machine with tape alphabet $\Gamma \supseteq \{0, 1, \text{`,'}\} \cup \{\Box\}$ that computes $f\colon \mathbb{N}^k \to \mathbb{N}$. The main idea is to show how the execution process of the TM M can be simulated using partial recursive functions.

Choose injective (i.e., one-to-one) functions

$$\begin{aligned} r &\colon \Gamma \to \text{odd numbers} \\ s &\colon Q \to \text{even numbers} \end{aligned}$$

encoding the tape alphabet and states of M respectively.

We consider a machine configuration $C = (q, t, i)$ as given by a word $a_1\, a_2\, \ldots\, a_{j-1}\, q\, a_j\, \ldots\, a_l$ over the alphabet $\Gamma \cup Q$ where:

- the word $a_1\, a_2\, \ldots\, a_l$ represents a portion of the tape t that includes within it the head position as well as all non-blank symbols; and
- $1 \leq j \leq l$, where j represents the position of the tape head i relative to the start of the extracted tape portion.

The configuration C can thus be encoded as a number

$$\ulcorner C \urcorner \ := \ \ulcorner r(a_1)\, r(a_2)\, \ldots\, r(a_{j-1})\, s(q)\, r(a_j)\, \ldots\, r(a_l) \urcorner \ .$$

The following functions are then primitive recursive.

$$\begin{aligned} \mathsf{step}\colon \mathbb{N} \to \mathbb{N} \qquad \mathsf{step}(x) &= \begin{cases} \ulcorner C' \urcorner & \text{if } x = \ulcorner C \urcorner \text{ and } \mathsf{step}_M(C) = C' \\ 0 & \text{otherwise} \end{cases} \\ \mathsf{run}\colon \mathbb{N}^2 \to \mathbb{N} \qquad \mathsf{run}(n, x) &= \mathsf{step}^n(x) \\ \mathsf{extract}\colon \mathbb{N} \to \mathbb{N} \qquad \mathsf{extract}(x) &= \begin{cases} n & \text{if } x = \ulcorner s(\mathsf{halt})\, r(\mathsf{bin}(n)) \urcorner \\ 0 & \text{otherwise} \end{cases} \\ \mathsf{halt?}\colon \mathbb{N} \to \mathbb{N} \qquad \mathsf{halt?}(x) &= \begin{cases} 0 & \text{if } x = \ulcorner s(\mathsf{halt})\, r(w) \urcorner \text{ for some } w \\ 1 & \text{otherwise} \end{cases} \\ \mathsf{init}\colon \mathbb{N}^k \to \mathbb{N} \qquad \mathsf{init}(x_1, \ldots, x_k) &= \ulcorner s(\mathsf{start})\, r(\mathsf{bin}(x_1)\, \text{`,'}\, \ldots\, \text{`,'}\, \mathsf{bin}(x_k)) \urcorner \end{aligned}$$

Then f is explicitly given as a partial recursive function by:

$$f(x_1, \ldots, x_k) \ = \ \mathsf{extract}(\mathsf{run}(\mu(n \mapsto \mathsf{halt?}(\mathsf{run}(n, \mathsf{init}(x_1, \ldots, x_k)))), \mathsf{init}(x_1, \ldots, x_k))) \ .$$

5 Enumerating the Computable Partial Functions

We can view the course thus far as having established two alternative approaches to defining computability. On the one hand, we have the *Turing machine model*, according to which the basic notion of computability concerns partial functions on words $\Sigma_1^* \rightharpoonup \Sigma_2^*$. On the other, we have the notion of *partial recursive function*, which, in a self-contained way, defines a basic notion of computability for partial functions on natural numbers $\mathbb{N}^k \rightharpoonup \mathbb{N}$. Theorem 4.4 establishes that both approaches lead to the same notion of *computable partial function* on $\mathbb{N}^k \rightharpoonup \mathbb{N}$.

When considering the TM model as basic, we introduced the notion of *representation* to account for computability with forms of data beyond words over an alphabet. Indeed, it was via suitable representations of natural numbers and products that the notion of TM-computable function $\mathbb{N}^k \rightharpoonup \mathbb{N}$ was defined.

Given that the computable partial functions on $\mathbb{N}$ can be viewed in their own right as the fundamental notion of computability, an alternative approach to computability with general forms of data is to take the partial recursive functions on $\mathbb{N}$ as basic, and to define computability on other sets X via representations of the elements of X using natural-number encodings for them instead of the word encodings used in Note 3. Since it is standard in computability theory to adopt this second approach, we now make the small change of perspective required to effect this.

For the remainder of the course, we redefine the notion of representation. Henceforth, a *representation* of a set X is a surjective partial function $\rho\colon \mathbb{N} \rightharpoonup\!\!\!\!\!\rightarrow X$. Given this redefinition, it is easy to similarly redefine concepts related to representations by making the necessary adjustments. For example, the notion of computable partial function between represented sets, Definition 3.7, becomes the following.

Definition 5.1 (Computable partial function)**.** Given representations $\rho_1\colon \mathbb{N} \rightharpoonup\!\!\!\!\!\rightarrow X_1$ and $\rho_2\colon \mathbb{N} \rightharpoonup\!\!\!\!\!\rightarrow X_2$, we say that a partial function $f\colon X_1 \rightharpoonup X_2$ is $(\rho_1 \to \rho_2)$-*computable* if there exists a computable partial function $g\colon \mathbb{N} \rightharpoonup \mathbb{N}$ such that, for every $x \in X_1$ and ρ_1-name n for x,

- $g(n)\!\downarrow$ if and only if $f(x)\!\downarrow$, and
- $g(n)\!\downarrow$ implies $g(n)$ is a ρ_2-name for $f(x)$.

(Equivalently, for any $n \in \mathsf{dom}(\rho_1)$, (i) $\rho_2(g(n)) \simeq f(\rho_1(n))$, and (ii) $g(n)\!\downarrow \Rightarrow g(n) \in \mathsf{dom}(\rho_2)$.)

A partial function $g\colon \mathbb{N} \rightharpoonup \mathbb{N}$ satisfying the conditions above is said to be a *realiser* for the function f.

We write $\rho_{\mathbb{N}}$ for the identity representation $\rho_{\mathbb{N}} := n \mapsto n\colon \mathbb{N} \twoheadrightarrow \mathbb{N}$. Given two representations $\rho_1\colon \mathbb{N} \rightharpoonup\!\!\!\!\!\rightarrow X_1$ and $\rho_2\colon \mathbb{N} \rightharpoonup\!\!\!\!\!\rightarrow X_2$, the *product representation* $(\rho_1 \times \rho_2)\colon \mathbb{N} \rightharpoonup\!\!\!\!\!\rightarrow (X_1 \times X_2)$ is defined by:

$$(\rho_1 \times \rho_2)(n) \;\simeq\; \begin{cases} (x_1, x_2) & \text{if } n = p(n_1, n_2) \text{ and } \rho_1(n_1) = x_1 \text{ and } \rho_2(n_2) = x_2 \\ \uparrow & \text{otherwise} \end{cases}$$

where p is the pairing function from Note 4.

A representation ρ is said to be *total* if ρ is a total function (defined on all numbers). Clearly $\rho_{\mathbb{N}}$ is total. Also, the product operation preserves the totality of representations.

5.1 Enumerating computable partial functions

For every $n \geq 1$, we shall enumerate all the n-ary computable partial functions:

$$\phi_0^n,\ \phi_1^n,\ \phi_2^n,\ \phi_3^n,\ \ldots$$

(In the case $n = 1$, we usually write simply ϕ_e rather than ϕ_e^1.) This enumeration enjoys the following properties.

- For every $e \in \mathbb{N}$, ϕ_e^n is an n-ary computable partial function.
- Every n-ary computable partial function arises as ϕ_e^n for some $e \in \mathbb{N}$.

Thus the function $e \mapsto \phi_e^n$ is a representation of the set

$$\mathsf{Comp}(\mathbb{N}^n \rightharpoonup \mathbb{N}) \ := \ \{f \colon \mathbb{N}^n \rightharpoonup \mathbb{N} \mid \ f \text{ is computable}\}$$

To define the enumeration, recall our encoding, from Lecture 2, of any Turing machine M as a string:[2]

$$\langle M \rangle \in {\Sigma_U}^* = \{\, 0,\, 1,\, -1,\, [\,,\,]\,,\, \|\,,\, \bullet\,\}^* \ .$$

By renaming the characters in Σ_U as integers $0, 1, \ldots, 6$, we have $\langle M \rangle \in \mathbb{N}^*$. By combining with the function $\ulcorner \cdot \urcorner \colon \mathbb{N}^* \to \mathbb{N}$ defined in Lecture 4, we thus encode a TM as a single natural number

$$\ulcorner \langle M \rangle \urcorner \in \mathbb{N} \ .$$

We define the partial function $\phi_e^n \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ by

$$\phi_e^n(x_1, \ldots, x_n) \ \simeq \ \begin{cases} y & \text{if there is a TM } M \text{ such that } e = \ulcorner \langle M \rangle \urcorner \text{ and} \\ & \quad M \text{ halts in the } \mathsf{halt} \text{ state with output } \mathsf{bin}(y) \\ & \quad \text{when run on input } \mathsf{bin}(x_1)\,\text{`,'}\,\ldots\,\text{`,'}\,\mathsf{bin}(x_n) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Clearly the enumeration $(\phi_e^n)_{e \in \mathbb{N}}$ enjoys the two bullet-pointed properties above.

We use the enumeration of computable partial functions to prove:

Proposition 5.2. *The total function $h \colon \mathbb{N} \to \mathbb{N}$ below is not computable.*

$$h(x) \ = \ \begin{cases} \phi_x(x) + 1 & \textit{if } \phi_x(x) \textit{ is defined} \\ 0 & \textit{otherwise} \end{cases}$$

Proof. Suppose, for contradiction, that h is computable. Then there exists $e \in \mathbb{N}$ such that $h = \phi_e$. So we have:

$$\phi_e(e) \ = \ h(e) \ = \ \phi_e(e) + 1 \ .$$

This is the required contradiction. □

[2] We modify this encoding very slightly by adapting it for Turing machines with a distinguished halt state, rather than for TMs with accept and reject states.

5.2 The universal computable partial function

Theorem 5.3 (Universal function)**.** *For any $n \geq 1$, the $(n+1)$-ary function ψ_U^n defined below is computable.*

$$\psi_U^n(e, x_1, \ldots, x_n) \;\simeq\; \phi_e^n(x_1, \ldots, x_n) \;.$$

We call ψ_U^n the *universal function* for n-ary computable partial functions. In the special case $n = 1$, we shall often write ψ_U rather than ψ_U^1. The universal function ψ_U^n can be understood as a realiser for the *evaluation function*

$$(f, x_1, \ldots, x_n) \mapsto f(x_1, \ldots, x_n) \colon \mathsf{Comp}(\mathbb{N}^n \rightharpoonup \mathbb{N}) \times \mathbb{N}^n \rightharpoonup \mathbb{N} \,.$$

The existence of the universal function thus means that the evaluation function is computable as a partial function between represented sets.

Proof. We outline the construction of a TM, V, that computes ψ_U^n.

V takes an input string of the form:

$$\mathsf{bin}(e)\ \text{`,'}\ \mathsf{bin}(x_1)\ \text{`,'}\ \ldots\ \text{`,'}\ \mathsf{bin}(x_n)$$

It then proceeds as follows.

- It checks that $e = \ulcorner \langle M \rangle \urcorner$ for some TM M. This can be done by extracting the unique string w such that $e = \ulcorner w \urcorner$ and then inspecting w to see if it is a legitimate encoding of a TM. If not, V goes into a loop.
- If the above check succeeds, then V simulates the execution of M on the input string $\mathsf{bin}(x_1)\,\text{`,'}\,\ldots\,\text{`,'}\,\mathsf{bin}(x_n)$ by running the universal Turing machine on the input string $\langle M \rangle\, \langle \mathsf{bin}(x_1)\,\text{`,'}\,\ldots\,\text{`,'}\,\mathsf{bin}(x_n) \rangle$.
- If the execution of the universal TM terminates, then the final configuration is inspected to see if the machine is in the halt state with $\mathsf{bin}(y)$ on the output tape, for some y. If so, V terminates in the halt state with output $\mathsf{bin}(y)$.
- Otherwise, V goes into a loop.

□

As an application of the universal function, we prove the non-computability of a particular function of interest.

Proposition 5.4. *The unary (total) function below is not computable.*

$$g(x) \;=\; \begin{cases} 1 & \textit{if } \phi_x \textit{ is a total function} \\ 0 & \textit{otherwise} \end{cases}$$

Proof. Suppose, for contradiction, that g is computable.

Consider the function

$$h(x) \;=\; \begin{cases} \phi_x(x) + 1 & \text{if } \phi_x \text{ is total} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Using the assumption that g is computable, together with the computability of the universal function, it follows that h is computable, because

$$h(x) \;=\; \begin{cases} \psi_U(x,x)+1 & \text{if } g(x)=1 \\ 0 & \text{if } g(x)=0 \end{cases}$$

So $h = \phi_e$ for some e, and we have

$$\phi_e(e) \;=\; h(e) \;=\; \phi_e(e)+1 \;\;,$$

because h is total by definition. This gives the required contradiction. □

5.3 The s-m-n theorem

Theorem 5.5 (The s-m-n theorem)**.** *For every $n > m \geq 0$, there exists an $(m+1)$-ary primitive recursive function $s_n^m \colon \mathbb{N}^{m+1} \to \mathbb{N}$ such that, for all $e, x_1, \ldots, x_n$,*

$$\phi^{n-m}_{s_n^m(e,\,x_1,\ldots,x_m)}(x_{m+1},\ldots,x_n) \;\simeq\; \phi^n_e(x_1,\ldots,x_n) \;\;.$$

In other words, the s-m-n theorem states that, for every $n > m \geq 0$, the function

$$(f, x_1, \ldots, x_m) \mapsto ((x_{m+1},\ldots,x_n) \mapsto f(x_1,\ldots,x_n)) : \mathsf{Comp}(\mathbb{N}^n \rightharpoonup \mathbb{N}) \times \mathbb{N}^m \to \mathsf{Comp}(\mathbb{N}^{n-m} \rightharpoonup \mathbb{N})$$

is computable as a function between represented sets, and has a primitive recursive realiser.

Proof. The $m = 0$ case is trivial as s_n^0 can be the identity function.

If $m > 0$ define:

$$s_n^m(e,x_1,\ldots,x_m) \;=\; \begin{cases} \ulcorner\langle M'\rangle\urcorner & \text{if } e = \ulcorner\langle M\rangle\urcorner, \text{ where } M' \text{ is the TM that begins execution} \\ & \quad \text{by writing ``}\mathsf{bin}(x_1)\text{`,'} \ldots \text{`,'}\mathsf{bin}(x_m)\text{`,'}\text{'' in front of its} \\ & \quad \text{given input and then proceeds as } M \\ e & \text{otherwise} \end{cases}$$

(Note that the machine M' is designed so that it behaves correctly when it is given the input ``$\mathsf{bin}(x_{m+1})$`,' … `,'$\mathsf{bin}(x_n)$''.)

The function s_n^m is computable because there is a clear algorithmic process for going from $(\ulcorner\langle M\rangle\urcorner, x_1, \ldots, x_m)$ to $\ulcorner\langle M'\rangle\urcorner$. This can be formalised by defining a corresponding Turing machine.

The function s_n^m can be shown to be primitive recursive using the machinery developed in Exercise Classes. We do not go into the (rather involved) details. □

5.4 Kleene's normal form theorem

Kleene's normal form theorem shows that every computable partial function can be obtained from primitive recursive functions using only one invocation of the minimisation (μ) operation. Moreover, it can be so obtained in a standard way.

The result is often useful for allowing informal arguments about computability (e.g., appeals to the Church-Turing thesis, which will be discussed in Lecture 6) to be replaced by rigorous mathematical arguments.

Theorem 5.6 (Kleene's normal form theorem)**.** *There exists a unary primitive recursive function $U : \mathbb{N} \to \mathbb{N}$ and, for each $n \geq 1$, an $(n+2)$-ary primitive recursive function T^n such that, for all $e, x_1, \ldots, x_n$,*

$$\phi_e^n(x_1, \ldots, x_n) \text{ is defined iff there exists } z \in \mathbb{N} \text{ such that } T^n(e, x_1, \ldots, x_n, z) = 0; \quad (2)$$

and

$$\phi_e^n(x_1, \ldots, x_n) \;\simeq\; U((\mu T^n)(e, x_1, \ldots, x_n)) \;. \quad (3)$$

So $\phi_e^n(x_1, \ldots, x_n) = U(z)$, where z is the least number such that $T^n(e, x_1, \ldots, x_n, z) = 0$ if such a z exists.

Proof. The idea is that $T^n(e, x_1, \ldots, x_n, z) = 0$ holds if and only if $e = \ulcorner \langle M \rangle \urcorner$ for some TM M and z encodes the entire sequence of machine configurations, from initial configuration to final configuration, encountered when executing M on input $\mathsf{bin}(x_1)\,\text{`,'}\,\ldots\,\text{`,'}\,\mathsf{bin}(x_n)$. Then $U(z)$ extracts from an encoded sequence of TM configurations, the number y such that $\mathsf{bin}(y)$ is on the tape of the final halt configuration.

In more detail, we define

$$T^n(e, x_1, \ldots, x_n, z) = \begin{cases} 0 & \text{if } e = \ulcorner \langle M \rangle \urcorner \text{ for some TM } M \text{ and } z \text{ encodes a sequence } \ulcorner \langle C_1 \rangle \urcorner, \ldots, \ulcorner \langle C_t \rangle \urcorner \\ & \text{such that:} \\ & \quad \bullet\ C_1 \text{ is the configuration } (\mathsf{start}, (\mathsf{bin}(x_1)\,\text{`,'}\,\ldots\,\text{`,'}\,\mathsf{bin}(x_n))@0, 0); \\ & \quad \bullet\ \text{each config. } C_{i+1} \text{ is obtained by one step of } M \text{ computation from } C_i; \text{ and} \\ & \quad \bullet\ C_t \text{ is a configuration of the form } (\mathsf{halt}, \mathsf{bin}(y)@i, i) \\ 1 & \text{otherwise} \end{cases}$$

The function T^n can be shown to be primitive recursive using the machinery developed in exercise classes, and using ideas similar to those used in the proof of Proposition 4.7 in Lecture 4. We do not give further details.

We finally define

$$U(z) \;=\; \begin{cases} y & \text{if } z \text{ encodes a sequence } z_1, \ldots, z_t \text{ whose last entry } z_t \\ & \quad \text{is of the form } \ulcorner \langle C \rangle \urcorner \text{ where } C \text{ is a TM configuration} \\ & \quad \text{of the form } (\mathsf{halt}, \mathsf{bin}(y)@i, i) \\ 17 & \text{otherwise} \end{cases}$$

Again this function can be shown to be primitive recursive. □

We shall often write T for the function T^1, which will play a prominent role in future lectures.

6 The Church-Turing Thesis

Over the last century, numerous models of computation have been proposed.

1. *Partial recursive functions* (Gödel/Herbrand, early 1930s)
2. *λ-calculus* (Church 1936)
3. *Turing machines* (Turing 1936)
4. *String rewriting systems* (Post 1920s–40s)
5. *Unrestricted grammars* (Chomsky 1950s)
6. *State/register machines* (1960s)
7. *Nondeterministic and probabilistic TMs* (1960s and 70s)
8. *Quantum Turing machines* (Deutsch 1985)
9. *Hypercomputation* (2000s)
10. *Real world computers* (1940s–future)

Regarding the above, we have the following *mathematical* result.

> *Models 1–8 are equivalent in terms of:*
> - *which (partial) functions they compute, and*
> - *which problems they (semi)decide.*

We do not state this as a theorem because it is really a whole collection of theorems, each stating a pairwise equivalence between two different models. Moreover every such statement of pairwise equivalence needs a careful formulation in order to be made mathematically precise. In this course, we have seen one such precise statement: Theorem 4.4, which asserts the equivalence between models 1 and 3 above. Many of you will have encountered other such equivalence theorems in your research for your presentation.

There is another direction in which models 1–8 can be compared, namely in terms of computational efficiency. This comparison is less straightforward. For example, models 1 and 6 work directly with computation on natural numbers (sometimes integers), and do not directly take account of the space requirements needed in reality to store and compute with large numbers. In contrast, models 2–5, 7 and 8, do take account of such requirements. For example, in a TM, $O(\log(n))$ tape squares are needed to record a natural number n, which indeed corresponds to the reality of storing numbers. Nevertheless, assuming that the space requirements of computation are accounted for by, in the case of models 1 and 6, imposing an $O(\log(n))$ cost on integer storage, it can be proved that models 1–6 are equivalent in terms of the time/space complexity of computation, modulo a possible polynomial overhead in translating between models.[3] It is an open question whether the various models included in 7 and 8 are also equivalent to TMs in terms of their time/space complexity. For example, the

[3] In order to obtain this equivalence, certain nonstandard 'efficient' encodings of numbers are needed in some models (e.g., λ-calculus).

equality P = NP, which is a famous open question, asserts the polynomial-time-complexity equivalence of deterministic and nondeterministic TMs.

In this course, we concern ourselves only with *computability* not with *complexity*. For our purposes, therefore, we can assert that models 1–8 are equivalent.

The notion of *hypercomputation* does not describe a single model of computation, but rather refers in general to models of computation that go strictly beyond Turing machines in terms of computation power. In all known cases this is achieved by including features in the model that are in practice impossible (or at least highly unlikely) to be achievable in a physical computation device. Some examples of such features include: accessing "oracles" that know the answer to undecidable questions; computing for an infinite amount of time and then returning a result; testing infinite-precision real numbers for equality, etc. Such notions of hypercomputation can be useful as mathematical models. However, they do not bear much resemblance to computation as performed in practice.

In contrast, modern computers directly correspond to computation as performed in practice. However, compared with a Turing machine, a modern computer has a significant limitation. Even if gigantically large, the memory of a computer is finite. Because of this, real-world computers have less computation power than Turing machines.

Although the infinite tape of a Turing machine is clearly a physical impossibility, there is a good argument that it is the correct abstraction for modelling the abstract idea of physically realisable computation. Even though any physical computation process will necessarily only use a finite amount of storage and time, it is natural not to give *a priori* bounds to the amount. This exactly corresponds to the situation with a Turing machine, where, at any point in time, only a finite portion of the tape has been written on, although there is no upper bound to the amount of free tape available.

Modulo accepting the idealisation of allowing the potential of unlimited time and space, Turing machines model a notion of computation that is clearly *physically realisable*. Let us evaluate all the models under discussion from this perspective.

1.–6. These models are clearly physically realisable.

7. Nondeterminism is physically realisable modulo having a scheduler that makes the (correct!) nondeterministic choices. Probabilistic computation is physically realisable if we can tap into a physical source of randomness.

8. Quantum computation is physically realisable *in principle*. The extent to which it is physically realisable *in practice* is a major research question.

9. The physical realisability of hypercomputation is the realm of science fiction.

10. Computers are not just physically realisable, they are actually physically realised.

In his 1936 paper, Turing gave a compelling argument that Turing computability corresponds to the informal notion of computability by a person/machine/agent following an algorithm. Moreover, from the above discussion, we see that all the physically realisable *abstract* models of computation we have considered (models 1–8) have the same computation power as a Turing machine. Thus we have both philosophical and empirical reasons to believe that, whatever notion of computability one comes up with, as long as the model is physically reasonable, the notion of computability cannot properly go beyond that of Turing computability.

Historically, this realisation was achieved in the following steps.

- Church (1936) explicitly *proposed* that λ-calculus computability coincides with informal computability, though he did not provide further argument supporting this.
- Turing (1936) *argued* that TM-computability coincides with informal computability, and *proved* that TM-computability coincides with λ-calculus computability.
- Subsequently, numerous notions of computability have been given, and all physically feasible such notions have been *proven* to coincide with TM-computability.

Taking the above into consideration, Kleene explicitly formulated:

The Church-Turing Thesis: *The informal notion of computability coincides with the mathematical notion of TM-computability.*

Equivalently, this thesis may be reformulated as stating that informal computability coincides with any of the other notions of computability that have been shown to be mathematically equivalent to TM-computability.

By accepting the Church-Turing thesis, we can *prove* that mathematical functions (sets, relations, etc.) are computable merely by giving an informal algorithm to compute them. The Church-Turing thesis then tells us that there necessarily exists a corresponding Turing machine. In practice, the level of rigour of proof achieved in this way is acceptable, as the apparent reliance on the Church-Turing thesis is deceptive. We have enough experience on this course with manipulating Turing machines that we should be capable of building, from any informal algorithm description, a corresponding Turing machine. So we are confident that we could fill in all mathematical details in a proof of computability if pressed to do so (although it would certainly be a tedious exercise).

In regard to negative results, the Church-Turing thesis plays a more significant role. Typically we will give a *mathematical proof* that a function (set, relation, etc.) is not computable, making use of TM-computability (or one of the other equivalent models of computability) to prove the result. Having done this, we can invoke the Church-Turing thesis and more generally conclude that there is no hope of ever finding any sort of informal algorithm whatsoever. This general conclusion is entirely dependent on the Church-Turing thesis.

In my view, the Church-Turing thesis is a very rare (perhaps even unique) example of a philosophical question (*What is computability?*) receiving a precise mathematical answer (*TM-computability or equivalent*). Moreover, not only is this answer widely accepted, but there are not even any really plausible alternative answers on offer.

7 Computable and Computably Enumerable Sets

This lecture studies properties of *decidable* and *semidecidable* sets of natural numbers. Following standard practice we introduce new terminology for these sets, referring to them as *computable* and *computably enumerable* sets respectively.

7.1 Definitions and basic properties

Definition 7.1 (Computable set)**.** A subset $A \subseteq \mathbb{N}$ is said to be *computable* (alternative words: *decidable*, *recursive*) if its (total) *characteristic function* $\chi_A \colon \mathbb{N} \to \mathbb{N}$ is computable.

$$\chi_A(x) \;=\; \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases}$$

Definition 7.2 (Computably enumerable set)**.** A subset $A \subseteq \mathbb{N}$ is said to be *computably enumerable (c.e.)* (alternative words: *semidecidable*, *recursively enumerable*) if its *partial characteristic function* $\chi_A^p \colon \mathbb{N} \to \mathbb{N}$ is computable.

$$\chi_A^p(x) \;\simeq\; \begin{cases} 1 & \text{if } x \in A \\ \uparrow & \text{otherwise} \end{cases}$$

Definitions 7.1 and 7.2 are in fact special cases of the definitions of *decidable* and *semidecidable* subset from Lecture 3, adjusted from word representations to $\mathbb{N}$-representations and instantiated in the case of the representation $\rho_{\mathbb{N}}$. We give the above special cases of the definitions directly, since the concepts are fundamental.

The following important set (the *halting set*) will be use frequently in our study of computable and c.e. sets.

$$K \;:=\; \{e \in \mathbb{N} \mid \phi_e(e)\!\downarrow\}$$

We shall show that K is c.e. but not computable. These properties of K are fundamental to much of what follows in the course.

Proposition 7.3. *The set K is not computable.*

Proof. Suppose, for contradiction, that χ_K is computable. Then so is the partial function

$$h(x) \;\simeq\; \begin{cases} \uparrow & \text{if } \chi_K(x) = 1 \\ 0 & \text{if } \chi_K(x) = 0\,. \end{cases}$$

Since h is computable, we have $h = \phi_e$ for some e. Then:

$$\phi_e(e)\!\downarrow \;\Leftrightarrow\; e \in K \;\Leftrightarrow\; h(e)\!\uparrow \;\Leftrightarrow\; \phi_e(e)\!\uparrow\ .$$

This gives the required contradiction. □

Proposition 7.4. *The set K is computably enumerable.*

This will be proved below.

7.2 Characterisations of computable and c.e. sets

Proposition 7.5. *A set $A \subseteq \mathbb{N}$ is c.e. if and only if it is the domain of a unary computable partial function.*

Proof. The left-to-right implication is immediate since A is the domain of χ_A^p.

For the converse, suppose $f\colon \mathbb{N} \rightharpoonup \mathbb{N}$ is a computable partial function. Then the partial function $g\colon \mathbb{N} \rightharpoonup \mathbb{N}$ defined below is also computable.

$$g(x) \;\simeq\; \begin{cases} 1 & \text{if } f(x)\!\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

By definition $g = \chi^p_{\mathsf{dom}(f)}$. So $\mathsf{dom}(f)$ is indeed c.e. □

Proof of Proposition 7.4. K is the domain of the computable partial function $x \mapsto \psi_U(x,x)$. □

Proposition 7.5 gives us a standard enumeration of the c.e. sets:

$$W_e \;:=\; \mathsf{dom}(\phi_e) \;.$$

In other words, the function

$$e \mapsto W_e \;:\; \mathbb{N} \to \mathsf{CE}$$

gives a total representation of the set CE of all computably enumerable sets

Lemma 7.6. *A set $A \subseteq \mathbb{N}$ is c.e. if and only if there exists a computable total function $t\colon \mathbb{N}^2 \to \mathbb{N}$ such that, for all $x \in \mathbb{N}$,*

$$x \in A \quad \Leftrightarrow \quad \exists z\; t(x,z) = 0 \;. \tag{4}$$

Proof. For the left-to-right implication, suppose that A is c.e. Then $A = W_e$ for some e. By Kleene's normal form theorem (Theorem 5.6)

$$x \in W_e \quad \Leftrightarrow \quad \exists z\; T(e,x,z) = 0 \;.$$

Therefore

$$t(x,z) \;:=\; T(e,x,z)$$

is a computable total function with the required properties.

For the converse, given any computable total function $t\colon \mathbb{N}^2 \to \mathbb{N}$, let (4) define A. Since computable partial functions are closed under minimisation, the partial function $\mu t\colon \mathbb{N} \rightharpoonup \mathbb{N}$ is computable. Because t is total, the domain of μt is A. It thus follows from Proposition 7.5 that A is c.e. □

Given a subset $A \subseteq \mathbb{N}$, we write $\overline{A}$ for the complement subset

$$\overline{A} \;:=\; \mathbb{N} - A \;.$$

Theorem 7.7. *A set $A \subseteq \mathbb{N}$ is computable if and only if both A and $\overline{A}$ are c.e.*

Proof. The left-to-right implication is easy and left as an **exercise**.

For the right-to-left implication, suppose A and $\overline{A}$ are both c.e. By Lemma 7.6 there exist total computable $s, s' \colon \mathbb{N}^2 \to \mathbb{N}$ such that

$$\begin{aligned} x \in A \quad &\Leftrightarrow \quad \exists z \;\; s(x,z) = 0 \\ x \notin A \quad &\Leftrightarrow \quad \exists z \;\; s'(x,z) = 0 \;\; . \end{aligned}$$

It follows that the computable partial function g defined below is total.

$$g \;\; := \;\; \mu((x,z) \mapsto \min(s(x,z),\, s'(x,z))$$

Often we shall write such definitions using minimisation using a (hopefully) more readable notation. For example, in the case of g,

$$g(x) \;\; = \;\; \mu z \;\; (\, \min(s(x,z),\, s'(x,z)) = 0 \,) \;\; ,$$

read as: "g is the function that maps x to the smallest z such that $\min(s(x,z),\, s'(x,z)) = 0$".

The characteristic function χ_A can be computed using s, s' and g by:

$$\chi_A(x) \;\; = \;\; \begin{cases} 1 & \text{if } s(x, g(x)) = 0 \\ 0 & \text{otherwise} \;\; . \end{cases}$$

□

Since the set K is c.e. but not computable, the corollary below is an immediate consequence of Theorem 7.7.

Corollary 7.8. *The set $\overline{K}$ is not c.e.*

The terminology "computably enumerable" is motivated by an important property that characterises c.e. sets. Any (nonempty) c.e. set can be enumerated by a computable (total) function. This is established by the theorem below.

Theorem 7.9. *The following are equivalent for a set $A \subseteq \mathbb{N}$.*

1. *A is c.e.*
2. *$A = \emptyset$ or A is the range of a (unary) total computable function.*
3. *A is the range of a computable partial function.*

Proof.
1. ⇒ 2. Suppose A is c.e. and $A \neq \emptyset$. Select any $a_0 \in A$. By Lemma 7.6, there exists a total computable $s \colon \mathbb{N}^2 \to \mathbb{N}$ satisfying

$$y \in A \quad \Leftrightarrow \quad \exists z \;\; s(y,z) = 0 \;\; .$$

Using the pairing function $p \colon \mathbb{N}^2 \to \mathbb{N}$ from Lecture 4, it follows that A is the range of the total computable function below.

$$x \;\; \mapsto \;\; \begin{cases} y & \text{if } s(y,z) = 0 \text{ where } y, z \text{ are such that } x = p(y,z) \\ a_0 & \text{otherwise} \,. \end{cases}$$

2. ⇒ 3. This is trivial, because the empty set is the range of the everywhere undefined partial function, and every total function is *a fortiori* a partial function.

3. ⇒ 1. Suppose A is the range of ϕ_e. Let T and U be as in Kleene's normal form theorem. Then the total function $s\colon \mathbb{N}^2 \to \mathbb{N}$ below is computable (indeed primitive recursive).

$$s(w,y) \;=\; \begin{cases} 0 & \text{if } T(e,x,z)=0 \text{ and } w = U(z) \text{ where } y = p(x,z) \quad (w = \phi_e(x) \text{ follows}) \\ 1 & \text{otherwise}\,. \end{cases}$$

Then $w \in A$ if and only if $\exists y\ s(w,y) = 0$. So A is c.e. by Lemma 7.6. □

By Theorem 7.9, we have a second standard enumeration of the c.e. sets:

$$E_e \;:=\; \text{the range of } \phi_e\ .$$

That is, the function

$$e \mapsto E_e \;:\; \mathbb{N} \to \mathsf{CE}$$

gives a second total representation of the set of all computably enumerable sets. In the exercise class, it will be shown that the $e \mapsto W_e$ and $e \mapsto E_e$ representations are equivalent.

We say that a total function $f\colon \mathbb{N} \to \mathbb{N}$ is *strictly increasing* if $x < y$ implies $f(x) < f(y)$.

Theorem 7.10. *The following are equivalent for a set $A \subseteq \mathbb{N}$.*

1. *A is computable*
2. *A is finite or A is the range of a strictly increasing computable total function.*

Proof.
1. ⇒ 2. Suppose A is computable and infinite. Define $f(-1) = -1$. Then, for $x \geq 0$, define:

$$f(x) \;:=\; \text{the smallest } n > f(x-1) \text{ such that } \chi_A(n) = 1\,.$$

Because χ_A is a computable total function, and because A is infinite, this definition defines a strictly increasing computable total function $f{\restriction_{\mathbb{N}}}\colon \mathbb{N} \to \mathbb{N}$. By construction, $A = \mathsf{range}(f)$.

2. ⇒ 1. It is easy to see that every finite set is computable (**exercise**).

Suppose A is the range of a strictly increasing computable total function f. Define g by:

$$g(n) \;=\; \begin{cases} 1 & \text{if } n = f(x) \text{ where } x \text{ is the smallest number s.t. } f(x) \geq n \\ 0 & \text{if } n \neq f(x) \text{ where } x \text{ is the smallest number s.t. } f(x) \geq n\,. \end{cases}$$

Since f is strictly increasing and total, g is total too. Moreover g is easily seen to be computable. By construction, $g = \chi_A$. Therefore, A is computable. □

Corollary 7.11. *Every infinite c.e. set has an infinite computable subset.*

Proof. Let A be an infinite c.e. set. By Theorem 7.9, there exists a total computable function f such that $A = \mathsf{range}(f)$. Define $g\colon \mathbb{N} \to \mathbb{N}$ as follows.

$$\begin{aligned} g(0) \;&:=\; f(0) \\ g(n+1) \;&:=\; f(m), \text{ where } m \text{ is the smallest number s.t. } f(m) > g(n)\,. \end{aligned}$$

Since $\mathsf{range}(f)$ is infinite, g is total. By construction, g is also computable and increasing.

Clearly $\mathsf{range}(g) \subseteq \mathsf{range}(f) = A$, and $\mathsf{range}(g)$ is infinite. By Theorem 7.10, $\mathsf{range}(g)$ is thus an infinite computable subset of A. □

8 Rice's Theorem and the Rice-Shapiro Theorem

Let $\mathcal{C}$ be the set of unary computable partial functions. Today's two main theorems address the problem of understanding for which sets $\mathcal{B} \subseteq \mathcal{C}$ the question

Does a computable function belong to $\mathcal{B}$?

is decidable or semidecidable.

Here the relevant notions of decidability and semidecidability are those pertaining to subsets of a represented set, as defined in Lecture 3, but adapted to $\mathbb{N}$-based representations. In our case, the representation in question is given by our standard enumeration $\phi\colon \mathbb{N} \twoheadrightarrow \mathcal{C}$, which is a *total* representation. Using this fact, one can unwind the definitions to obtain the following easy characterisation (which, for the purposes of this lecture can equivalently be taken as a redefinition) of decidable and semidecidable subsets of $\mathcal{C}$. For any $\mathcal{B} \subseteq \mathcal{C}$ define:

$$I_{\mathcal{B}} \;:=\; \{e \in \mathbb{N} \mid \phi_e \in \mathcal{B}\} \ .$$

That is, $I_{\mathcal{B}}$ is the set of indices of functions in $\mathcal{B}$. Then:

- $\mathcal{B}$ is *decidable* if and only if $I_{\mathcal{B}}$ is computable; and
- $\mathcal{B}$ is *semidecidable* if and only if $I_{\mathcal{B}}$ is computably enumerable.

8.1 Total and partial reductions

Reductions provide a very useful method for showing that sets are or are not computable or computably enumerable. Suppose A and B are subsets of $\mathbb{N}$.

Definition 8.1 (Total reduction)**.** A *(total) reduction* of A to B is given by a computable total function $f\colon \mathbb{N} \to \mathbb{N}$ such that

$$x \in A \quad \Leftrightarrow \quad f(x) \in B \ .$$

(Equivalently $A = f^{-1}B$.) We say that A is *(totally) reducible* to B if there exists a total reduction of A to B.

Our terminology is non-standard. In the literature, the standard terminology is *many-to-one* reducible (or *m-reducible* for short) rather than totally reducible.

Definition 8.2 (Partial reduction)**.** A *partial reduction* of A to B is given by a computable partial function $g\colon \mathbb{N} \rightharpoonup \mathbb{N}$ such that

$$x \in A \quad \Leftrightarrow \quad g(x) \in B \ .$$

(Equivalently $A = g^{-1}B$, note that this implies that $g(x)\!\downarrow$ for all $x \in A$.) We say that A is *partially reducible* to B if there exists a partial reduction of A to B.

Note that every total reduction is *a fortiori* a partial reduction.

Lemma 8.3 (Reduction lemma)**.**

1. *If A is totally reducible to a computable set B then A is also computable.*
2. *If A is partially reducible to a c.e. set B then A is also c.e.*

Proof. The statements are easy consequences of the definitions of computable and c.e. set respectively. It is left as an **exercise** to check the details. □

8.2 Rice's theorem

Theorem 8.4 (Rice's theorem)**.** *The only decidable subsets of $\mathcal{C}$ are $\emptyset$ and $\mathcal{C}$.*

Proof. The asserted property expands to: for any $\mathcal{B} \subseteq \mathcal{C}$, it holds that $I_{\mathcal{B}}$ is computable if and only if $\mathcal{B} = \emptyset$ or $\mathcal{B} = \mathcal{C}$.

It is trivial that $\mathcal{B} = \emptyset$ and $\mathcal{B} = \mathcal{C}$ both imply that the set $I_{\mathcal{B}}$ is computable.

For the converse, suppose that we have proper inclusions $\emptyset \subset \mathcal{B} \subset \mathcal{C}$. We need to show that $I_{\mathcal{B}}$ is not computable.

Without loss of generality, suppose that $f_\emptyset \notin \mathcal{B}$ where $f_\emptyset$ is the everywhere undefined partial function. (This is no loss of generality, because if $f_\emptyset \in \mathcal{B}$ then $f_\emptyset \notin \mathcal{C} - \mathcal{B}$, so the argument below can be used to show that $I_{\mathcal{C}-\mathcal{B}}$ is not computable. But $I_{\mathcal{C}-\mathcal{B}} = \overline{I_{\mathcal{B}}}$, and $I_{\mathcal{B}}$ is computable if and only if $\overline{I_{\mathcal{B}}}$ is.)

Select any function $g \in \mathcal{B}$, and define a binary partial function

$$f(x,y) \;\simeq\; \begin{cases} g(y) & \text{if } x \in K \\ \uparrow & \text{otherwise}\,. \end{cases}$$

Then f is computable because K is c.e. By the s-m-n theorem, there is a total computable $s\colon \mathbb{N} \to \mathbb{N}$ such that

$$\phi_{s(x)}(y) \;\simeq\; f(x,y)\;\;.$$

Then we have:

$$\begin{aligned} x \in K \;&\Rightarrow\; \phi_{s(x)} = g \;\Rightarrow\; s(x) \in I_{\mathcal{B}} \\ x \notin K \;&\Rightarrow\; \phi_{s(x)} = f_\emptyset \;\Rightarrow\; s(x) \notin I_{\mathcal{B}}\,. \end{aligned}$$

That is, s is a total reduction of K to $I_{\mathcal{B}}$. Since K is not computable, it follows from the reduction lemma that $I_{\mathcal{B}}$ is also not computable. □

8.3 The Rice-Shapiro theorem

We say that a partial function $f\colon \mathbb{N} \rightharpoonup \mathbb{N}$ is *finite* if $\mathsf{dom}(f)$ is a finite set. It is easy to see (**exercise**) that every finite partial function is computable. For partial functions $f, g\colon \mathbb{N} \rightharpoonup \mathbb{N}$, we say that f is a *restriction* of g (notation $f \subseteq g$) if:

$$\text{for all } x \in \mathbb{N},\;\; f(x)\!\downarrow \;\Rightarrow\; g(x) = f(x)\;\;.$$

(In particular, it follows that $f(x)\!\downarrow \Rightarrow g(x)\!\downarrow$.)

Theorem 8.5 (The Rice-Shapiro theorem)**.** *If $\mathcal{B} \subseteq \mathcal{C}$ is semidecidable then, for any computable partial function f,*

$$f \in \mathcal{B} \;\Leftrightarrow\; \textit{there exists a finite } f' \subseteq f \textit{ with } f' \in \mathcal{B}\,. \tag{5}$$

Proof. We show that the failure of either implication of (5) allows us to reduce the set $\overline{K}$ to $I_{\mathcal{B}}$, implying that $I_{\mathcal{B}}$ is not c.e.; that is, $\mathcal{B}$ is not semidecidable.

Suppose the left-to-right implication of (5) fails; i.e.,

$$f \in \mathcal{B}, \text{ but for all finite } f' \subseteq f \text{ we have } f' \notin \mathcal{B}\,. \tag{6}$$

Because K is c.e., there is, by Lemma 7.6, a total computable $t\colon \mathbb{N}^2 \to \mathbb{N}$ such that

$$x \in K \;\Leftrightarrow\; \exists z \;\; t(x,z) = 0 \;\; .$$

Define a computable partial function

$$g(x,z) \;\simeq\; \begin{cases} f(z) & \text{if } t(x,y) \neq 0 \text{ for all } 0 \leq y \leq z \\ \uparrow & \text{if } t(x,y) = 0 \text{ for some } 0 \leq y \leq z \,. \end{cases}$$

By the s-m-n theorem, there is a total $s\colon \mathbb{N} \to \mathbb{N}$ such that

$$\phi_{s(x)}(z) \;\simeq\; g(x,z) \;\; .$$

By construction, $\phi_{s(x)} \subseteq f$ for all x. Also:

$$\begin{array}{lll} x \in K \;\Rightarrow\; \phi_{s(x)} \text{ is finite} \;\Rightarrow\; s(x) \notin I_{\mathcal{B}} & & \text{by (6)} \\ x \in \overline{K} \;\Rightarrow\; \phi_{s(x)} = f \;\Rightarrow\; s(x) \in I_{\mathcal{B}} \;\; . & & \end{array}$$

Thus s is a reduction of $\overline{K}$ to $I_{\mathcal{B}}$. Since $\overline{K}$ is not c.e. (Corollary 7.8), it follows from the reduction lemma that $I_{\mathcal{B}}$ is also not c.e.

Lastly, suppose the right-to-left implication of (5) fails; i.e., f is a computable partial function such that $f' \in \mathcal{B}$, for some finite $f' \subseteq f$, but $f \notin \mathcal{B}$. Define a computable partial function

$$g(x,z) \;\simeq\; \begin{cases} f(z) & \text{if } z \in \mathsf{dom}(f') \text{ or } x \in K \\ \uparrow & \text{otherwise.} \end{cases}$$

The s-m-n theorem provides a total computable $s\colon \mathbb{N} \to \mathbb{N}$ such that

$$\phi_{s(x)}(z) \;\simeq\; g(x,z) \;\; .$$

Since $f' \subseteq f$, the definition of g gives us that:

$$\begin{array}{l} x \in K \;\Rightarrow\; \phi_{s(x)} = f \;\Rightarrow\; s(x) \notin I_{\mathcal{B}} \\ x \in \overline{K} \;\Rightarrow\; \phi_{s(x)} = f' \;\Rightarrow\; s(x) \in I_{\mathcal{B}} \;\; . \end{array}$$

Once again, s is a reduction of $\overline{K}$ to $I_{\mathcal{B}}$, from which it follows that $I_{\mathcal{B}}$ is not c.e. $\square$

9 Varieties of Non-computable Set

9.1 Productive and creative sets

Since $(W_e)_{e\geq 0}$ enumerates the computably enumerable sets, the following are equivalent for a set $A \subseteq \mathbb{N}$.

- A is *not* computably enumerable.
- For every $e \geq 0$ such that $W_e \subseteq A$, there exists an element $a_e \in A - W_e$.

One can view a number $a_e \in A - W_e$ as a witness to the fact that $A \neq W_e$. An important class of non-c.e. sets is defined by asking for it to be possible to compute such a witness a_e from e.

Definition 9.1 (Productive set)**.** A set $A \subseteq \mathbb{N}$ is *productive* if there is a computable total function $g\colon \mathbb{N} \to \mathbb{N}$ such that, for all $e \geq 0$, it holds that $W_e \subseteq A$ implies $g(e) \in A - W_e$.

It is immediate that any productive set is *a fortiori* not c.e.

Proposition 9.2. *The set $\overline{K}$ is productive.*

Proof. It is convenient to use the following description of K which is just a reformulation of its definition.

$$K \;=\; \{e \mid e \in W_e\} \tag{7}$$

We show that the identity function $g(x) = x$ demonstrates the productivity of $\overline{K}$. Suppose $W_e \subseteq \overline{K}$. We need to show that $e \in \overline{K} - W_e$.

Suppose, for contradiction, that $e \in K$. By (7), $e \in W_e$. Since $W_e \subseteq \overline{K}$, we have $e \in \overline{K}$. This contradicts $e \in K$.

So $e \in \overline{K}$. By (7), it follows that $e \notin W_e$. Thus indeed $e \in \overline{K} - W_e$. □

Recall the notions of total and partial reduction from $A \subseteq \mathbb{N}$ to $B \subseteq \mathbb{N}$ (Definitions 8.1 and 8.2). By the reduction lemma (Lemma 8.3), if A is reducible to B and A is not c.e. then neither is B. In the case that the reduction is total, one gets a similar preservation of productivity.

Lemma 9.3 (Reduction lemma, bis)**.** *If a productive set $A \subseteq \mathbb{N}$ is totally reducible to $B \subseteq \mathbb{N}$ then B is also productive.*

Proof. Let $f\colon \mathbb{N} \to \mathbb{N}$ be a total reduction from A to B.

Using the universal function, the partial function

$$(e, x) \mapsto \phi_e(f(x))$$

is computable. By the s-m-n theorem,[4] there is a computable total $h\colon \mathbb{N} \to \mathbb{N}$ such that

$$\phi_{h(e)}(x) \;\simeq\; \phi_e(f(x)) \;,$$

that is $\phi_{h(e)} = \phi_e \circ f$.

[4]Explicitly, suppose $(e, x) \mapsto \phi_e(f(x))$ is given as ϕ^2_d. Then define h to be the total computable function $e \mapsto s^1_2(d, e)$.

Let g be the computable total function that shows the productivity of A. We show that $f \circ g \circ h$ witnesses the productivity of B.

Accordingly, suppose $W_e \subseteq B$, i.e., $\mathsf{dom}(\phi_e) \subseteq B$. Since f reduces A to B, we have that $\mathsf{dom}(\phi_e \circ f) \subseteq A$; i.e., $\mathsf{dom}(\phi_{h(e)}) \subseteq A$; i.e., $W_{h(e)} \subseteq A$.

As g withesses the productivity of A, we have that $g(h(e)) \in A - W_{h(e)}$. That is $g(h(e)) \in A - \mathsf{dom}(\phi_e \circ f)$. Again using that f reduces A to B, we have that $f(g(h(e))) \in B - \mathsf{dom}(\phi_e)$. That is $f(g(h(e))) \in B - W_e$ as required. □

Recall from Lecture 7 that we write $I_{\mathcal{B}}$ for the set of indices of partial functions in a set $\mathcal{B} \subseteq \mathcal{C}$, and we write $f_\emptyset$ for the everywhere undefined partial function. The next result is a strengthening of Rice's theorem (Theorem 8.4).

Theorem 9.4. *Suppose $\mathcal{B} \subseteq \mathcal{C}$ is such that $f_\emptyset \in \mathcal{B} \neq \mathcal{C}$. Then $I_{\mathcal{B}}$ is productive.*

Proof. The proof of Rice's theorem (following the bracketed alternative to the "without loss of generality" part) defines a reduction from $\overline{K}$ to $I_{\mathcal{B}}$. Since $\overline{K}$ is productive (Proposition 9.2), it follows from Lemma 9.3 that $I_{\mathcal{B}}$ is too. □

Since productive sets are sets that strongly fail to be c.e., one can (via Theorem 7.7) view a c.e. set whose complement is productive as a c.e. set that strongly fails to be computable. Such sets are called *creative*.

Definition 9.5 (Creative set)**.** A subset of $\mathbb{N}$ is creative if it is c.e. and its complement is productive.

Proposition 9.6. *The set K is creative.*

Proof. Immediate from Propositions 7.4 and 9.2. □

Theorem 9.7. *Suppose $\emptyset \subset \mathcal{B} \subset \mathcal{C}$ is such that $I_{\mathcal{B}}$ is c.e. then $I_{\mathcal{B}}$ is creative.*

Proof. Suppose $I_{\mathcal{B}}$ is c.e. and $\emptyset \neq \mathcal{B} \neq \mathcal{C}$. Since $I_{\mathcal{B}}$ is c.e., it is not productive, so $f_\emptyset \notin \mathcal{B}$, by Theorem 9.4. Thus $f_\emptyset \in \mathcal{C} - \mathcal{B}$, so $I_{\mathcal{C}-\mathcal{B}}$ is productive, by Theorem 9.4. But $I_{\mathcal{C}-\mathcal{B}} = \overline{I_{\mathcal{B}}}$. So indeed $I_{\mathcal{B}}$ is creative. □

9.2 An extended example: arithmetic truth

The *first-order formulas* in the language of arithmetic are defined by:

- If p_1 and p_2 are (multivariate) polynomials with coefficients in $\mathbb{N}$ then $p_1 = p_2$ is a formula (*equality*). (N.b., the set of multivariate polynomials includes the univariate polynomials as a special case, and even individual natural numbers as a particularly degenerate case.)
- If ϕ is a formula then so is $\neg\phi$ (*negation*).
- If ϕ and ψ are formulas then so are: $\phi \wedge \psi$ (*conjunction*), $\phi \vee \psi$ (*disjunction*), and $\phi \to \psi$ (*implication*).
- If ϕ is a formula then so are: $\forall x\ \phi$ (*universal quantification*), and $\exists x\ \phi$ (*existential quantification*), for any variable x.

Formulas allow us to express properties of natural numbers. For example

$$\exists z\ x + z = y \tag{8}$$

expresses the relation $x \leq y$. More interestingly,

$$2 \leq x\ \wedge\ \forall y \forall z\, (x = y.z \rightarrow x = y \vee x = z) \tag{9}$$

expresses the property that x is prime number.

A formula is called a *sentence* if every variable x in the formula appears within the scope of a quantifier $\forall x$ of $\exists x$ mentioning the same variable. Sentences express statements about numbers that are either true or false. For example,

$$\forall x \exists y\ x \leq y\ \wedge\ \mathsf{Prime}(y)$$

states the true property that there are infinitely many prime numbers. Here, the expressions $x \leq y$ and $\mathsf{Prime}(y)$ are abbreviations for the formulas (8) and (9) above, with appropriate renamings of variables when necessary.[5]

Another similar example of a sentence is

$$\forall x\ (2 \leq x\ \rightarrow\ \exists y \exists z\ (\mathsf{Prime}(y) \wedge \mathsf{Prime}(z) \wedge 2.x = y + z))$$

which states the property that every even number greater than or equal to 4 is the sum of two prime numbers. This statement is a famous open question in number theory known as *Goldbach's conjecture*.

The formulas (hence also sentences) of arithmetic are written as sequences of symbols. Each formula ϕ can thus be encoded as a number $\ulcorner \phi \urcorner$, using the coding techniques of Lecture 4 (Section 4.2). Define:

$$\mathsf{Sent}\ :=\ \{n \in \mathbb{N} \mid n = \ulcorner \phi \urcorner \text{ for some sentence } \phi\}\ .$$

The set Sent is computable because one can decode any $n \in \mathbb{N}$ as a sequence of symbols, and algorithmically check that this sequence satisfies the definition of a sentence.

A major goal of number theory in mathematics is to establish whether interesting sentences ϕ (such as Goldbach's conjecture) are true or not. Truth amounts to membership of the set

$$\mathsf{True}\ :=\ \{n \in \mathbb{N} \mid n = \ulcorner \phi \urcorner \text{ for some true sentence } \phi\}\ .$$

The result below is a computability-theoretic version of the famous *incompleteness theorem* of Kurt Gödel.

Theorem 9.8. *The set* True *is productive.*

This theorem makes an important statement about the limitations of algorithmic processes in mathematics. Suppose we are given any method of enumerating (or semideciding) a set of truths in arithmetic. (For example, any computable list of axioms, such as the Peano axioms, generate such a method.) We can (routinely) convert this method (which presents a computably enumerable set of truths) to an index e such that W_e is the enumerated set of truths, hence $W_e \subseteq \mathsf{True}$. By applying the function g that witnesses the productivity of True,

[5] For example, $\mathsf{Prime}(y)$ is $(\exists z\ 2 + z = y)\ \wedge\ \forall y' \forall z\, (y = y'.z\ \rightarrow\ y = y' \vee y = z)$.

we obtain $g(e) \in \mathsf{True} - W_e$. Thus we (algorithmically) find a new truth that was omitted in our original enumeration.

In summary, the truths of arithmetic are not axiomatisable in their entirety, nor indeed enumerable by any algorithmic means whatsoever (assuming the Church-Turing thesis is true).

While a detailed proof of Theorem 9.8 is beyond the scope of this course, it is possible to give an outline of the high-level argument. First one shows that, for any primitive recursive function $f\colon \mathbb{N}^k \to \mathbb{N}$, there exists a formula with $\phi_f(x_1, \ldots, x_k, y)$ with *free variables* (those not *bound* by quantifiers) $x_1, \ldots, x_k, y$ such that, for all $n_1, \ldots, n_k, m \in \mathbb{N}$,

$$\phi_f(n_1, \ldots, n_k, m) \text{ is true } \Leftrightarrow \ f(n_1, \ldots, n_k) = m \ .$$

The construction of ϕ_f is surprisingly involved.

Given the general construction of ϕ_f, we in particular have the formula ϕ_T, where T is as in Kleene's normal form theorem (Theorem 5.6). Now consider the total function

$$e \ \mapsto \ \ulcorner \neg \, \exists z \ \phi_T(e, e, z, 0) \urcorner \ , \tag{10}$$

which is computable because it involves calculating the code of a sentence that is constructed in a straightforward way from ϕ_T and e. For any e, it holds that $\ulcorner \neg \, \exists z \ \phi_T(e, e, z, 0) \urcorner \in \mathsf{True}$ if and only if

$$\text{there is no } z \text{ such that } T(e, e, z) = 0 \, .$$

But this holds if and only if $e \notin W_e$, that is if and only if $e \in \overline{K}$.

We have shown that (10) defines a total reduction from $\overline{K}$ to True. Since $\overline{K}$ is productive, it follows from Lemma 9.3 that the set True is too.

9.3 Immune and simple sets

Every c.e. but non-computable set we have met thus far is creative. The final goal of this note is to show that there also exist noncomputable c.e. sets that are *not* creative; i.e., there exist noncomputable c.e. sets whose complements are not productive.

To achieve this, we first observe a nontrivial property of productive sets.

Theorem 9.9. *If $A \subseteq \mathbb{N}$ is productive then A contains an infinite c.e. subset.*

Proof. Suppose $g\colon \mathbb{N} \to \mathbb{N}$ witnesses the productivity of A.

We shall define a computable total function $f\colon \mathbb{N} \to \mathbb{N}$ satisfying:

$$\begin{aligned} f(0) &= e_0 \text{ s.t. } W_{e_o} = \emptyset \\ f(1) &= e_1 \text{ s.t. } W_{e_1} = \{g(e_0)\} \\ f(2) &= e_2 \text{ s.t. } W_{e_2} = \{g(e_0), g(e_1)\} \\ &\cdots \\ f(n+1) &= e_{n+1} \text{ s.t. } W_{e_{n+1}} = W_{e_n} \cup \{g(e_n)\} \, . \end{aligned}$$

By induction on n, each e_n satisfies $W_{e_n} \subseteq A$, so $g(e_n) \in A - W_{e_n}$. Therefore $g \circ f$ is a total computable function whose range is an infinite subset of A. By Theorem 7.9, its range is thus an infinite c.e. subset of A.

It remains to define a computable f as above. By the s-m-n theorem, there exists a computable total $h\colon \mathbb{N} \to \mathbb{N}$ such that

$$\phi_{h(x)}(y) \simeq \begin{cases} 17 & \text{if } y \in W_x \text{ or } y = g(x) \\ \uparrow & \text{otherwise}\,. \end{cases}$$

So $W_{h(e)} = W_e \cup g(e)$. Therefore, f is computable by primitive recursion.

$$\begin{aligned} f(0) &= \text{any chosen } e_0 \text{ such that } W_{e_0} = \emptyset \\ f(n+1) &= h(f(n))\ . \end{aligned}$$

□

Corollary 9.10. *Every productive set has an infinite computable subset.*

Proof. Combine Theorem 9.9 and Corollary 7.11. □

Theorem 9.9 suggests a way of identifying an interesting collection of sets that are neither productive nor c.e.

Definition 9.11 (Immune set)**.** A subset of $\mathbb{N}$ is *immune* if it is infinite but contains no infinite c.e. subset.

Obviously an immune set is not itself c.e. By Theorem 9.9, it is also not productive.

Definition 9.12 (Simple set)**.** A subset of $\mathbb{N}$ is *simple* if it is c.e. and its complement is immune.[6]

Clearly a simple set is a non-computable c.e. set that is also not creative. The final result today thus achieves the goal set at the start of the section.

Theorem 9.13 (Post's theorem)**.** *There exists a simple set.*

Proof. Consider the partial function f below, which is computable because it is definable from the universal function using minimisation.

$$f(e) \simeq \begin{cases} \phi_e(z) & \text{if } z \text{ is the smallest number such that} \\ & \quad \phi_e(0),\, \phi_e(1),\, \ldots\,,\, \phi_e(z) \\ & \text{are all defined and } \phi_e(z) \geq 2e\,, \\ \uparrow & \text{if no such } z \text{ exists.} \end{cases}$$

Define $A = \mathsf{range}(f)$. We show that A is simple.

As A is the range of a computable partial function, it is c.e by Theorem 7.9.

To show that $\overline{A}$ is infinite, note that $f(e) = n$ implies that $n \geq 2e$. So the numbers $\{0, 1, \ldots, 2n-1\}$ can only appear as $f(e)$ when $e \in \{0, \ldots, n-1\}$. So, for every $n \geq 0$, at least n distinct numbers from the set $\{0, 1, \ldots, 2n-1\}$ belong to $\overline{A}$. Therefore $\overline{A}$ is indeed infinite.

Finally, let B be an infinite c.e. set. We need to show that $B \not\subseteq \overline{A}$. Since B is c.e., we have, by Theorem 7.9, that $B = \mathsf{range}(\phi_e)$ for some e such that ϕ_e is total. But then $f(e)$ is defined, because ϕ_e is total and B is infinite so certainly contains some number $n \geq 2e$. We therefore have $f(e) = \phi_e(z)$, for some z; whence $f(e) \in \mathsf{range}(f) \cap \mathsf{range}(\phi_e) = A \cap B$. So indeed $B \not\subseteq \overline{A}$. □

[6] Although the terminology *simple* is clearly a misnomer, it is standard.

10 Computing with Infinite Words

Thus far in this course, we have been considering *discrete* data such as natural numbers and strings, and notions of computability for sets and functions of such data. In his 1936 paper *"On computable numbers, with an application to the Entscheidungsproblem"*, Turing's basic notion of computability concerned *continuous* data. The central definition in his paper was that of *computable real number*.

One way of defining what it means for a real number to be computable is as follows. We might say that a TM computes a real number if it generates the infinite decimal expansion of the number along its output tape, e.g.,

$$3 \cdot 1\,4\,1\,5\,9\,2\,6\,5\,3\,5\,8\,9\,7\,9\,3\,2\,3\,8\,4\,6\,2\,6\,4\,3\,3\,8\,3\,2\,7\,9\,5\,0\ \ldots$$

Such a TM clearly needs to compute forever. We also need to require that it continues to output data as it computes. Furthermore, we need to ask that the TM does not overwrite initial portions of the output sequence once written on the tape, since, at any point in the computational process, we would like to be able to identify the finite approximation to the output number that has been already calculated.

Using the above considerations, one can define a precise notion of *computable real number* in terms of ordinary TMs, as Turing did. However, the special status of the output sequence (once a finite portion of it has been calculated it never gets overwritten) suggests that we might *build this into* the model of computation. It is also natural (and in keeping with the practical experience of computation with *streams* of data) to consider infinite sequences as input.

For example, the following simple algorithm defines an intuitively computable function from $\{0,\ldots,9\}^\omega$ (the set of infinite sequences of decimal digits) to $\{0,\ldots,9\}^\omega$.

1. Read the next digit d from the input sequence.
2. Write $9-d$ to the output sequence.
3. Go back to step 1.

An example of a computation following the algorithm is:

$$\begin{aligned}\text{Input:}\ & 1\,4\,1\,5\,9\,2\,6\,5\,3\,5\,8\,9\,7\,9\,3\,2\,3\,8\,4\,6\,2\,6\,4\,3\,3\,8\,3\,2\,7\,9\,5\,0\ \ldots\\ \text{Output:}\ & 8\,5\,8\,4\,0\,7\,3\,4\,6\,4\,1\,0\,2\,0\,6\,7\,6\,1\,5\,3\,7\,3\,5\,6\,6\,1\,6\,7\,2\,0\,4\,9\ \ldots\end{aligned}$$

In the case of this algorithm, if we understand the input sequence as the decimal expansion of a real number $x \in [0,1]$ (e.g., $x = 0.1415926\ldots$) then the algorithm computes the decimal expansion of the number $y = 1-x$ (e.g., $y = 0.8584073\ldots$). That is, the algorithm computes the function $x \mapsto 1-x\colon [0,1] \to [0,1]$ on real numbers.

10.1 Type 2 Turing Machines

We now introduce a model of computation specifically designed to compute with infinite sequences/streams of data, as motivated above. We write Σ^ω for the set of infinite sequences of symbols from Σ. We variously refer to the elements of Σ^ω as *sequences*, *infinite words* or *ω-words*.

The notion of a *Type 2 Turing Machine* provides a simple mathematical model of an idealised machine that takes infinite words as input and generates an infinite word as output.

Definition 10.1 (Type 2 Turing machine)**.** A *(deterministic) Type 2 Turing Machine* (*T2M* for short), with $k \geq 0$ input tapes and $N \geq 0$ working tapes, is specified by:

- a finite tape alphabet Γ with $\Box \in \Gamma$;
- an input/output alphabet $\Sigma \subseteq \Gamma - \{\Box\}$;
- a finite set Q of states with $\mathsf{start} \in Q$;
- a transition (partial) function

$$\delta\colon\ Q \times \Sigma^k \times \Gamma^N \ \rightharpoonup\ Q \times \{0,+1\}^k \times \Gamma^N \times \{-1,0,+1\}^N \times (\Sigma \cup \{\Box\})\ .$$

Such a T2M has k *input tapes* which are infinite to the right (but not to the left). Associated with each input tape is a corresponding *read head* which is unidirectional: it can move only to the right. Each square of the input tape is required to contain a symbol from Σ.

As for an ordinary multi-tape TM, the N working tapes are infinite in both directions, and each tape has a bidirectional read/write head. The tape squares contain symbols from Γ, and only finitely many squares are non-blank.

Finally, there is a single *output tape*, which is again infinite to the right. Associated with this tape is a unidirectional write head; again, it can move only to the right. At any point in time, the output tape will start with finitely many (possibly 0) symbols from Σ, beyond which the remaining squares will all be blank. The write head is always positioned at the first blank square.

The computation of a T2M starts in the initial configuration in which:

- The state of the machine is start.
- The k read heads are at the leftmost squares of the k input tapes. And each input tape i contains an infinite sequence $p^i \in \Sigma^\omega$.
- The N working tapes are all blank.
- The write head is at the left end of the output tape and the output tape is blank.

A single step of computation then proceeds as follows. Suppose:

- the current machine state is $q \in Q$;
- the symbols at the current read positions on the k input tapes are $a_1, \ldots, a_k \in \Sigma$;
- the symbols at the current head positions on the N working tapes are $b_1, \ldots, b_N \in \Gamma$.

Suppose also that $\delta(q\,;\, a_1, \ldots, a_k\,;\, b_1, \ldots, b_N)$ is defined and equal to

$$(\,q'\,;\, e_1, \ldots, e_k\,;\, c_1, \ldots, c_N\,;\, d_1, \ldots, d_N\,;\, z\,)$$

then the following actions are performed.

- The state is changed to q'.
- Each read head i (where $1 \leq i \leq k$) is is moved according to e_i:

- if $e_i = 0$ then read head i is not moved;
- if $e_i = +1$ then read head i is moved one square to the right.

- On each working tape j (where $1 \leq j \leq N$) the following is done.
  - First symbol $c_j \in \Gamma$ is written on working tape j.
  - Then read/write head j is moved according to $d_j \in \{-1, 0, +1\}$.

- On the output tape the following is done.
  - If $z = \square$ then no action is taken.
  - Otherwise $z \in \Sigma$ is written to the output tape, after which the write head is moved one square to the right.

One could use the above description to formally define tape and machine configurations, and to formally define the step function on machine configurations, as in Lecture 1. However, at this stage in the course we leave this as a routine though cumbersome **exercise**.

10.2 Modi operandi of T2Ms

As with ordinary TMs, we consider various *modi operandi* of T2Ms, depending on their purpose. Specifically, we consider:

- computing infinite words;
- computing functions on infinite words;
- deciding/semideciding sets of infinite words.

10.2.1 Computing infinite words

Definition 10.2 (Computing a sequence)**.** A T2M M with k input tapes is said to *compute* an infinite word

$$p \;=\; p_0 p_1 p_2 p_3 \cdots \in \Sigma^\omega$$

given input $(p^1, \ldots, p^k) \in (\Sigma^\omega)^k$ if, for every $n \geq 0$, there exists t such that, after t steps of computation, M has written

$$p\restriction_n \;:=\; p_0\; p_1\; \cdots\; p_{n-1}$$

as the first n symbols on the output tape.

Definition 10.3 (Computable sequence)**.** A sequence $p \in \Sigma^\omega$ is said to be *computable* if there exists a T2M with no input tapes that computes it.

In the case that a T2M M has $k \geq 1$ input tapes, then it is possible for M to compute a non-computable $p \in \Sigma^\omega$ given a non-computable input $(p^1, \ldots, p^k)$.

10.2.2 Computing functions on infinite words

Definition 10.4 (Computing a partial function)**.** A k-input-tape T2M M is said to *compute* a k-ary partial function $f\colon (\Sigma^\omega)^k \rightharpoonup \Sigma^\omega$ if and only if:

- if $(p^1, \ldots, p^k) \in \mathsf{dom}(f)$ then M computes $f(p^1, \ldots, p^k)$ given input $(p^1, \ldots, p^k)$; and
- M given input $(p^1, \ldots, p^k)$ computes some $p \in \Sigma^\omega$ only if $(p^1, \ldots, p^k) \in \mathsf{dom}(f)$.

Definition 10.5 (Computable partial function)**.** A partial function $f\colon (\Sigma^\omega)^k \rightharpoonup \Sigma^\omega$ is said to be *computable* if there exists some T2M that computes it.

Note that, unlike with ordinary TMs, no halt state is used to define computability for functions on infinite words. Indeed, computations that produce infinite words cannot halt.

10.2.3 Deciding/semideciding sets of infinite words.

An *ω-language* is a subset $L \subseteq \Sigma^\omega$. In order to recognise ω-languages, we assume a T2M comes with:

- distinct halting states $\mathsf{accept}, \mathsf{reject} \in Q$.

Definition 10.6 (Acceptance/rejection)**.**

- A single-input-tape T2M *accepts* $p \in \Sigma^\omega$ if, when run with p on its input tape, it eventually terminates in the accept state.
- A single-input-tape T2M *rejects* $p \in \Sigma^\omega$ if, when run with p on its input tape, it eventually terminates in the reject state.

Definition 10.7 (Recognised ω-language)**.** The *language recognised* by a single-input-tape T2M M is

$$\mathcal{L}_\omega(M) \;:=\; \{p \in \Sigma^\omega \mid M \text{ accepts } p\} \;.$$

Definition 10.8 (Decidability/semidecidability)**.**

- $L \subseteq \Sigma^\omega$ is said to be *semidecidable* if there exists a T2M M such that $L = \mathcal{L}_\omega(M)$.

  (We also say that the machine M *semidecides* L.)
- $L \subseteq \Sigma^\omega$ is said to be *decidable* if there exists a T2M M such that $L = \mathcal{L}_\omega(M)$ and also M rejects every $p \in \Sigma^\omega - L$.

  (We also say that the machine M *decides* L.)

Proposition 10.9. *If an ω-language $L \subseteq \Sigma^\omega$ is decidable then it is semidecidable.*

10.3 Relation to TM computation

As already alluded to at the start of this note, there is no absolute need to consider the notion of type-2 Turing machine, in order to define the above notions of computability. It is in fact possible to define all the relevant computability notions concerning ω-words directly in terms of ordinary TMs. However, the definitions are more natural and compelling when formulated using type-2 Turing machines, which is why we have chosen this approach.

Nevertheless, to illustrate how the notions can be defined in terms of ordinary TMs, we consider one case: the computability of infinite words. Specifically, the result below characterises the computability of an infinite word $p \in \Sigma^\omega$ according to the T2M model (Definition 10.3) in terms of the (semi)decidabity of the language

$$\mathsf{Prefix}(p) \;:=\; \{p\restriction_n \mid n \geq 0\} \;\subseteq\; \Sigma^*$$

of finite prefixes of p, according to the TM model (Definition 1.7).

Proposition 10.10. *For any $p \in \Sigma^\omega$, the following are equivalent.*

1. *The infinite word p is T2M-computable.*
2. *The language $\mathsf{Prefix}(p)$ is TM-decidable.*
3. *The language $\mathsf{Prefix}(p)$ is TM-semidecidable.*

In particular, for languages of the form $\mathsf{Prefix}(p)$, semidecidability coincides with decidability. This is very much a peculiarity of the restricted class of prefix languages.

Proof.
1. ⇒ 2. Let M be a T2M that computes p. To decide if a given input word $w \in \Sigma^*$ belongs to $\mathsf{Prefix}(p)$, do the following.

- Simulate the execution of M until it produces as output symbols $b_0 \dots b_{|w|-1}$. (Since M computes $p \in \Sigma^\omega$ it is guaranteed to eventually produce $|w|$ symbols.) Check the equality $w = b_0 \dots b_{|w|-1}$. If this is true halt in state accept, otherwise halt in state reject.

  (The above informal algorithm can routinely be converted to an ordinary TM.)

2. ⇒ 3. Every decidable language is *a fortiori* semidecidable.

3. ⇒ 1. Suppose that M is an ordinary TM that semidecides $\mathsf{Prefix}(p)$. Let Σ be $\{a_1, \dots, a_m\}$, where all m symbols are distinct. Construct a T2M that behaves as follows.

- Once $w = w_0 \dots w_{n-1}$ has already been output, simulate the simultaneous execution of m copies of M running on the input words $wa_1, \dots, wa_m$ respectively. This is done by interleaving the computation steps of the m copies of M so that each copy completes t execution steps before any copy proceeds with step $t+1$. Because M semidecides $\mathsf{Prefix}(p)$, exactly one copy will accept its input, wa_j say. When this happens, output a_j and return to the start of this bullet point.

  Note that, at the start of the execution, $w = \varepsilon$ has "already been output", as this means that nothing has been output, so the above algorithm does indeed get started.

  (The above informal algorithm can be converted to a T2M.)

□

The notions of computability for partial functions on ω-words and (semi)decidability of ω-languages can be reformulated in terms of ordinary TMs in a similar way. However, we shall not go further into this.

11 Topological Aspects of Computing with Infinite Words

11.1 Semidecidability and open sets

Definition 11.1 (Cylinder set)**.** Every finite word $w \in \Sigma^*$ determines a *cylinder set* $\langle w \rangle \subseteq \Sigma^\omega$ defined by

$$\langle w \rangle \ := \ \{p \in \Sigma^\omega \mid p \restriction_{|w|} = w\} \ .$$

Definition 11.2 (Open set)**.** A subset $U \subseteq \Sigma^\omega$ is *open* if, for every $p \in U$ there exists $n \geq 0$ such that

$$\langle p \restriction_n \rangle \ \subseteq \ U \ .$$

Theorem 11.3. *If $L \subseteq \Sigma^\omega$ is semidecidable then it is open.*

Proof. Let M be a T2M that recognises L. Consider any $p \in L$. We must show that, for some $n \geq 0$, it holds that $\langle p \restriction_n \rangle \subseteq L$.

By the definition of recognising L, the T2M M terminates in the accept state when run on input p. Define:[7]

$$n \ := \ 1 + \text{position of input head when computation terminates.}$$

Note that the computation of M on input p is entirely determined by the first n symbols of the input tape, as the read head has not yet reached the other symbols. Accordingly, if p' is any sequence in Σ^ω for which $p' \restriction_n = p \restriction_n$ then the T2M also halts in the accept state when run on input p'. In other words, M accepts every ω-word in $\langle p \restriction_n \rangle$. Since M recognises L, this means that $\langle p \restriction_n \rangle \subseteq L$. □

11.2 Decidability and clopen sets.

Recall that a family $\mathcal{O}$ of subsets of a set X is the collection of *open sets* of a *topology* if it is closed under finite intersections (including the empty intersection, meaning that $X \in \mathcal{O}$) and arbitrary unions (including the empty union, meaning that $\emptyset \in \mathcal{O}$).

Proposition 11.4. *The open subsets, as defined above, form a topology on Σ^ω.*

We omit the proof, which follows directly from the definitions.

Given a topology $\mathcal{O}$ on a set X, recall that a subset $\mathcal{B} \subseteq \mathcal{O}$ is a *base* for the topology if every $U \in \mathcal{O}$ is a union sets in $\mathcal{B}$, i.e.,

$$U \ = \ \bigcup \{B \in \mathcal{B} \mid B \subseteq U\} \ .$$

Proposition 11.5. *The cylinder sets form a countable base for the topology on Σ^ω.*

Proof. Since each cylinder set is determined by a word $w \in \Sigma^*$, it is clear that there are countably many cylinder sets. The condition for them forming a base requires that, for every open set $U \subseteq \Sigma^\omega$, we have

$$U \ = \ \bigcup \{\langle w \rangle \mid w \in \Sigma^*, \ \langle w \rangle \subseteq U\} \ .$$

This is obvious from the definition of open set. □

[7] We consider the leftmost square of the input tape as being in position 0.

Given a topology $\mathcal{O}$ on a set X, recall that a subset $A \subseteq X$ is said to be *closed* if its complement

$$\overline{A} \;:=\; X - A$$

is open.

Proposition 11.6. *If a subset $L \subseteq \Sigma^\omega$ is decidable then it is both open and closed,*

Proof. If $L \subseteq \Sigma^\omega$ is decidable then both L and $\overline{L}$ are semidecidable. Hence, by Theorem 11.3, they are both open. □

Sets that are simultaneously open and closed are a rare phenomenon in the geometry-based spaces that are often used as the main examples in topology courses. For example, in *connected* spaces such as $\mathbb{R}^n$, nontrivial open sets are never closed. In contrast, the space Σ^ω has a rich collection of such closed open sets. By the very definition of the topology on Σ^ω, every cylinder set is open. As the next proposition observes, cylinder sets are also closed.

Proposition 11.7. *Every cylinder set is closed in Σ^ω.*

Proof. For any $w \in \Sigma^*$, consider the cylinder set $\langle w \rangle$. For any $p \in \overline{\langle w \rangle}$, we have that that w is not a prefix of p. Hence $\langle p\restriction_{|w|} \rangle \subseteq \overline{\langle w \rangle}$. This shows that $\overline{\langle w \rangle}$ is open. □

A set that is both open and closed in a topological space is called a *clopen* set. We have shown that cylinder sets are all clopen. Since cylinder sets form a base, the space Σ^ω has a base of clopen sets. (In topology, spaces with bases of clopen sets are called *zero-dimensional* spaces.)

Proposition 11.6 provides a connection between clopen sets and computability theory: every decidable set is clopen. In fact, something more holds: the decidable sets are exactly the clopen sets.

Theorem 11.8. *A subset $L \subseteq \Sigma^\omega$ is decidable if and only if it is clopen.*

This theorem is one of today's main results. The left-to-right implication has already been proved as Proposition 11.6. The converse implication is trickier, and depends on the topological notion of *compactness*. We shall return to this in Section 11.5 below.

11.3 Computability and continuity

Recall that a function $f\colon X \to Y$ between two topological spaces is defined to be *continuous* if, for every open subset V of Y, the preimage $f^{-1}V$ is an open subset of X. Equivalently, f is continuous if and only if the following property holds.

- For every $x \in X$ and for every open $V \subseteq Y$ with $f(x) \in V$, there exists an open subset $U \subseteq X$ with $x \in U$ such that, for all $x' \in U$, we have $f(x') \in V$.

Theorem 11.9 (Continuity theorem (total functions))**.** *If a total function $f\colon \Sigma^\omega \to \Sigma^\omega$ is computable then f is continuous.*

This continuity theorem for total functions is, in fact, a special case of a more general continuity theorem for partial functions. Recall that for any subset Z of a topological space X, the *subspace* (or *relative*) *topology* on Z has as its open sets:

$$\{U \cap Z \mid U \text{ is an open subset of } X\} \;.$$

Theorem 11.10 (Continuity theorem (partial functions))**.** *If a partial function $f\colon \Sigma^\omega \rightharpoonup \Sigma^\omega$ is computable then f is continuous as a total function from $\mathsf{dom}(f)$ (endowed with the subspace topology) to Σ^ω.*

In the special case of total functions $f\colon \Sigma^\omega \to \Sigma^\omega$, Theorem 11.10 asserts the same property as Theorem 11.9.

The next result characterises the property of continuity in more explicit terms. Continuity amounts to a computationally meaningful property: finite prefixes of the output sequence are determined by finite prefixes of the input sequence.

Proposition 11.11 (Characterisation of continuity)**.** *The following are equivalent for a partial function $f\colon \Sigma^\omega \rightharpoonup \Sigma^\omega$.*

1. *f is continuous as a total function from $\mathsf{dom}(f)$ (with the subspace topology) to Σ^ω.*
2. *For all $p \in \mathsf{dom}(f)$ and $n \geq 0$, there exists $m \geq 0$ such that, for all $p' \in \mathsf{dom}(f)$,*

$$p' \restriction_m = p \restriction_m \quad \Rightarrow \quad f(p') \restriction_n = f(p) \restriction_n \ .$$

The proof is left as an exercise.

Proof of Theorem 11.10. Let $f\colon \Sigma^\omega \rightharpoonup \Sigma^\omega$ be a computable partial function, and let M be a T2M that computes f. We show that f satisfies property 2 of Proposition 11.11. Accordingly, consider any $p \in \mathsf{dom}(f)$ and $n \geq 0$.

By the definition of computing f, there exists $t \geq 0$ such that, after t steps of computation, M outputs $f(p) \restriction_n$ when given input p. Define:

$$m \ := \ 1 + \text{position of input head after } t \text{ computation steps on input } p\,.$$

Note that the computation of M on input p for t computation steps is entirely determined by the first m symbols of the input tape. (The read head has not yet reached the other symbols.)

Let p' be any sequence in Σ^ω such that $p' \restriction_m = p \restriction_m$. Then the first t steps of the computation of M on input p' proceed identically to the computation of M on input p. Therefore, the computation of M on input p' writes $f(p) \restriction_n$ as the first n symbols on the output tape.

Since, in the case that $p' \in \mathsf{dom}(f)$, the computation of M on p' computes $f(p')$, this means that $f(p') \restriction_n = f(p) \restriction_n$. □

11.4 Domains of computable partial functions are G_δ sets

A subset Z of a topological space X is said to be G_δ (pronounced "gee-delta") if it can be expressed as a countable intersection of open sets; that is, if there exists a countable family $\mathcal{F}$ of open sets such that $X = \bigcap \mathcal{F}$. (Here *countable* means either finite or countably infinite.) It is trivial that every open set is G_δ. In general, however, there are G_δ subsets that are not open. It is easy to see that G_δ subsets are closed under finite unions and countable intersections. In the topological space Σ^ω, it further holds that every closed set is G_δ. (**Exercise:** prove this!)

Theorem 11.12. *If a partial function $f\colon \Sigma^\omega \rightharpoonup \Sigma^\omega$ is computable then $\mathsf{dom}(f)$ is a G_δ subset of Σ^ω.*

Proof. Let $f\colon \Sigma^\omega \rightharpoonup \Sigma^\omega$ be a computable partial function, and let M be a T2M that computes f. For every $n \geq 0$, consider the following subset of Σ^ω.

$$D_n \;:=\; \{p \in \Sigma^\omega \mid M \text{ produces} \geq n \text{ output characters when run on input } p\} \;.$$

The set D_n is semidecidable, because one can define a T2M that behaves like M but which also counts (on an additional working tape), as it computes, how many output characters have been given so far. As soon as this count number reaches the value n, the machine halts in the accept state.

Since D_n is semidecidable, it is open. Note that

$$\bigcap_n D_n \;=\; \{p \in \Sigma^\omega \mid M \text{ produces infinitely many output characters when run on input } p\} \;.$$

By the definition of what it means for M to compute f, it follows that $\bigcap_n D_n = \mathsf{dom}(f)$. As a countable intersection of open sets, $\mathsf{dom}(f)$ is thus indeed a G_δ set. □

11.5 Characterising decidability via compactness

Given a topology $\mathcal{O}$ on a set X, and a subset $K \subseteq X$, an *open cover of* K is a family $\mathcal{W} \subseteq \mathcal{O}$ such that $\bigcup \mathcal{W} \supseteq K$. If $\mathcal{W}$ is an open cover of K then $\mathcal{W}' \subseteq \mathcal{W}$ is said to be a *subcover* of K if $\bigcup \mathcal{W}' \supseteq K$.

Definition 11.13 (Compactness)**.** A subset $K \subseteq X$ is *compact* if every open cover of K has a finite subcover.

We now restrict attention to the space $X = \Sigma^\omega$, with its topology of open sets as defined in Section 11.1.

Theorem 11.14. *Σ^ω is compact.*

(It is crucial in this theorem that the alphabet Σ is finite.)

Proof. Let $\mathcal{W}$ be an open cover of Σ^ω. Assume for contradiction that $\mathcal{W}$ has no finite subcover of Σ^ω.

We shall define an infinite sequence $p_0, p_1, p_2, \ldots$ of elements of Σ, constructed by induction on $n \geq 0$ to satisfy, for every n:

- The n-element word $p_0 \ldots p_{n-1}$ is such that $\mathcal{W}$ contains no finite subcover of $\langle p_0 \ldots p_{n-1}\rangle$.

Base case: $n = 0$ The n-element word $p_0 \ldots p_{n-1}$ is the empty word ε, whence $\langle p_0 \ldots p_{n-1}\rangle = \langle \varepsilon \rangle = \Sigma^\omega$. Indeed $\mathcal{W}$ has no finite subcover of Σ^ω by assumption.

Step case: $n + 1$ We already have the symbols $p_0, \ldots, p_{n-1}$, and the induction hypothesis is that the n-element word $p_0 \ldots p_{n-1}$ is such that $\mathcal{W}$ contains no finite subcover of $\langle p_0 \ldots p_{n-1}\rangle$.

Let the finite alphabet be $\Sigma = \{a_1, a_2, \ldots, a_k\}$ (with k distinct symbols). Then

$$\langle p_0 \ldots p_{n-1}\rangle \;=\; \langle p_0 \ldots p_{n-1}a_1\rangle \cup \langle p_0 \ldots p_{n-1}a_2\rangle \cup \cdots \cup \langle p_0 \ldots p_{n-1}a_k\rangle \;.$$

If each of the k cylinder sets on the right had a finite subcover in $\mathcal{W}$, then the union of these k finite subcovers would give a finite subcover from $\mathcal{W}$ covering $\langle p_0 \ldots p_{n-1}\rangle$, contradicting the induction hypothesis. Accordingly, at least one cylinder set $\langle p_0 \ldots p_{n-1}a_i\rangle$ (where $1 \leq i \leq k$)

has no finite subcover in $\mathcal{W}$. Let i be the least such, and define $p_n := a_i$. We then indeed have that $\mathcal{W}$ contains no finite subover of $\langle p_0 \dots p_n \rangle$, as required.

The infinite sequence $p_0, p_1, p_2, \dots$ defines an ω-word $p \in \Sigma^\omega$. Because $\mathcal{W}$ is an open cover of Σ^ω, there exists some open set $U \in \mathcal{W}$ with $p \in U$. Since U is open, there exists $n \geq 0$ such that $\langle p_0 \dots p_{n-1} \rangle \subseteq U$. Thus the singleton set $\{U\}$ is a finite subset of $\mathcal{W}$ that covers $\langle p_0 \dots p_{n-1} \rangle$. This contradicts the property proved by induction, which states that $\mathcal{W}$ has no finite subcover of $\langle p_0 \dots p_{n-1} \rangle$. □

Corollary 11.15. *Every closed subset $A \subseteq \Sigma^\omega$ is compact.*

We omit the proof. On the one hand, it is a standard lemma from topology that every closed subset of a compact space is compact. On the other, it is anyway very easy to give a direct argument, and this is left as an exercise.

We can now complete the proof of Theorem 11.8. We need to prove that every clopen subset of Σ^ω is decidable.

Proof of Theorem 11.8: right-to-left implication. Suppose that L is clopen. Since L is open, we have $L = \bigcup \{ \langle w \rangle \mid w \in \Sigma^*,\ \langle w \rangle \subseteq L \}$; and so

$$\{ \langle w \rangle \mid w \in \Sigma^*,\ \langle w \rangle \subseteq L \}$$

is an open cover of L. Since L is closed, it is compact. Thus the open cover of L above has a finite subcover, and so

$$L \ = \ \langle w_1 \rangle \cup \dots \cup \langle w_m \rangle \ ,$$

for some $m \geq 0$ and words $w_1, \dots, w_m \in \Sigma^*$.

We now have the following algorithm for deciding the set L, presented as an informal description of a Type 2 Turing Machine. Given input $p \in \Sigma^\omega$, the machine looks at the first n input characters, where n is the length of the longest word among $w_1, \dots, w_m$. For each word w_i $(1 \leq i \leq m)$, the machine checks to see if the input prefix $p \restriction_{|w_i|}$ is the same as w_i. If this is true, for some i, the machine halts in the accept state. If not, a fact which is ascertained after finitely many steps of computation during which every $i = 1, \dots, m$ is considered, the machine halts in the reject state. □

12 Computing with Real Numbers

Type 2 Turing machines compute with ω-words. In Lecture 10, we used this to perform computation on real numbers, by representing real numbers as ω-words using decimal expansions. Alternatively, one might use binary representations or any other base b notation ($b \geq 2$). As it happens, this approach has serious limitations. For example, Proposition 12.1 below shows that the multiplication-by-three function is non-computable with respect to the decimal representation, and this is a very basic function!

Proposition 12.1 (Inadequacy of decimal representation)**.** *There is no computable partial function $f\colon \Sigma^\omega \rightharpoonup \Sigma^\omega$, where $\Sigma = \{0, \ldots, 9, \text{`.'}\}$ which maps every decimal representation of $x \in \mathbb{R}^+$ to a decimal representation of $3x$.*

Proof. We prove the stronger property that there is no such continuous f. By the continuity theorem (Theorem 11.10), this implies that there is no computable f.

Since $3 \times \frac{1}{3} = 1$, such a continuous f must give either:

$$f(0.333^\omega) = 1.000^\omega \quad \text{or} \quad f(0.333^\omega) = 0.999^\omega \ .$$

Suppose that the left-hand equality holds.

By continuity, there exists $m \geq 0$ such that, for any decimal representation $p' \in \Sigma^\omega$ with

$$p' \restriction_m \ = \overbrace{0.33\ldots3}^{m \text{ symbols}} \ ,$$

it holds that $f(p') \restriction_3 = 1.0$. In particular, if we consider the case

$$p'' \ := \ \overbrace{0.33\ldots30}^{m \text{ symbols}}0^\omega$$

then we have $f(p'') \restriction_3 = 1.0$. However, p'' is the decimal representation of a number $x < \frac{1}{3}$. Since $3x < 1$, no valid decimal representation for $3x$ can begin with the prefix 1.0. This contradicts that f computes decimal representations for the $3x$ function.

A similar argument gives a contradiction in the case that the right-hand equality holds. □

In this lecture, we find a better representation for real numbers, allowing T2Ms to successfully perform real-number computation. More generally, we might be interested in computing with elements of other interesting mathematical sets, for example:

$$\mathbb{C} \qquad \{(x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 + z^2 = 1\} \qquad \omega_1 \text{ (the first uncountable ordinal)}$$

To cater for such a variety of possibilities, we introduce a general notion of *(type 2) representation* of a set X, via which elements of X are represented by infinite words.

12.1 Type two representations

In Lecture 3, we studied *representations* that encode elements of a set using *names* given as words over an alphabet. *Type two representations* provide a similar encoding of elements of a set using *ω-words* as names instead of finite words. Because there are continuum-many ω-words, type two representations can be used to represent elements of uncountable sets such as $\mathbb{R}$. The definitions in this section parallel those of Note 3.

Definition 12.2. A *(type two) representation* of a set X by infinite words from an alphabet Σ is a surjective partial function $\gamma\colon \Sigma^\omega \twoheadrightarrow X$.

Here the notation $\twoheadrightarrow$ indicates a surjective partial function. Recall that surjectivity means that, for every $x \in X$, there exists $p \in \Sigma^\omega$ with $\gamma(p) = x$. We call any infinite word p such that $\gamma(p) = x$ a *name* (or *realiser*) for x.

Since the set Σ^ω has continuum cardinality (i.e., cardinality $2^{\aleph_0}$), any represented set X can have *at most* continuum cardinality.

Definition 12.3 (Computable element)**.** Given a representation $\gamma\colon \Sigma^\omega \twoheadrightarrow X$, we say that an element $x \in X$ is γ*-computable* if there exists a computable $p \in \Sigma^\omega$ such that $x = \gamma(p)$. That is, x is γ-computable if it has some computable name.

When γ can be inferred from the context, we simply say that x is *computable*.

Definition 12.4 (Computable partial function)**.** Given representations $\gamma\colon \Sigma^\omega \twoheadrightarrow X$ and $\gamma'\colon \Sigma^\omega \twoheadrightarrow X'$, we say that a partial function $f\colon X \rightharpoonup X'$ is $(\gamma \to \gamma')$*-computable* if there exists a computable partial function $g\colon \Sigma^\omega \rightharpoonup \Sigma^\omega$ such that, for every $x \in X$ and γ-name p for x,

- $g(p)\!\downarrow$ if and only if $f(x)\!\downarrow$, and
- $g(p)\!\downarrow$ implies $g(p)$ is a γ'-name for $f(x)$.

Equivalently, for any $p \in \mathsf{dom}(\gamma)$, both: (i) $\gamma'(g(p)) \simeq f(\gamma(p))$, and (ii) $g(p)\!\downarrow \Rightarrow \gamma'(g(p))\!\downarrow$.

Such a partial function g is said to be a *realiser* for the function f.

If f is $(\gamma \to \gamma')$-computable and g realises f then the following diagram of partial functions commutes, for all $p \in \mathsf{dom}(\gamma)$.

$$\begin{array}{ccc} \Sigma^\omega & \xrightarrow{\;g\;} & \Sigma^\omega \\ {\scriptstyle\gamma}\downarrow & & \downarrow{\scriptstyle\gamma'} \\ X & \xrightarrow[\;f\;]{} & X' \end{array}$$

For total functions f the definition of computability is equivalent to the commutativity of the above diagram for all $p \in \mathsf{dom}(\gamma)$. For a partial function f, the definition of computability is stronger than commutativity.

12.2 Representations of $\mathbb{R}$

In Lecture 10 we implicitly considered a decimal digit representation of $\mathbb{R}$. Now we have the notion of representation, we can give an explicit representation of $\mathbb{R}$. For the sake of variety, we consider a binary digit representation rather than a decimal one.

Example 12.5 (Binary digit representation of $\mathbb{R}$)**.** Consider the alphabet $\Sigma_b = \{0, 1, -, \text{`.'}\}$. We define a representation $\gamma_b\colon \Sigma_b^\omega \twoheadrightarrow \mathbb{R}$ as follows.

We define the domain $\mathsf{dom}(\gamma_b)$ to consist of those infinite words that have one of the two forms below

$$d_{m-1} \ldots d_0 \,.\, d_{-1}\, d_{-2}\, d_{-3}\, d_{-4} \ldots \qquad \text{or} \qquad -\, d_{m-1} \ldots d_0 \,.\, d_{-1}\, d_{-2}\, d_{-3}\, d_{-4} \ldots$$

where $m \geq 0$, and each $d_i \in \{0, 1\}$.

For $p \in \mathsf{dom}(\gamma_b)$, the value $\gamma_b(p)$ is defined by:

$$\gamma_b(p) \;=\; \begin{cases} \sum_{i=m-1}^{-\infty} d_i \,.\, 2^i & \text{if } p \text{ does not begin with } - \\ -\sum_{i=m-1}^{-\infty} d_i \,.\, 2^i & \text{if } p \text{ does begin with } -\,. \end{cases}$$

The next result shows that the binary representation of $\mathbb{R}$ suffers from the same defect as the decimal representation implicitly considered in Lecture 10, cf. Proposition 12.1.

Proposition 12.6. *The function $x \mapsto 3x \colon \mathbb{R} \to \mathbb{R}$ is not $(\gamma_b \to \gamma_b)$-computable*

This is proved similarly to Proposition 12.1.

We repair this unsatisfactory situation by defining an *improved* representation of $\mathbb{R}$, the *Cauchy representation*. A real number will be represented as a (fast converging) sequence of (dyadic) rational numbers. Recall that a rational number is *dyadic* if it can be expressed as an integer fraction in which the denominator is a power of 2. We write $\mathbb{Q}_d$ for the set of dyadic rationals. We can represent any such dyadic rational by a finite words $u \in \Sigma_b^*$ of the form

$$d_{m-1} \,\ldots\, d_0 \,\mathbf{.}\, d_{-1} \,\ldots\, d_{-n} \qquad \text{or} \qquad -\, d_{m-1} \,\ldots\, d_0 \,\mathbf{.}\, d_{-1} \,\ldots\, d_{-n}$$

where $m, n \geq 0$. Such a word u is interpreted as the dyadic rational $q_d(u)$ defined by:

$$q_d(u) \;=\; \begin{cases} \sum_{i=m-1}^{-n} d_i \,.\, 2^i & \text{if } u \text{ does not begin with } - \\ -\sum_{i=m-1}^{-n} d_i \,.\, 2^i & \text{if } u \text{ does begin with } -\,. \end{cases}$$

This just implements the ordinary binary notation for dyadic rational numbers. It defines a *type one representation* (i.e., a representation using finite words for names in the sense of Note 3) $q_d \colon \Sigma_b^* \rightharpoonup \mathbb{Q}_d$ of the dyadic rationals.

Example 12.7 (Cauchy representation of $\mathbb{R}$)**.** Using the above type one representation of dyadic rationals, we define the *Cauchy representation* $\gamma_c \colon \Sigma_c^\omega \rightarrow\!\!\!\!\!\rightarrow \mathbb{R}$, where $\Sigma_c := \Sigma_b \cup \{\,\mathbf{;}\,\}$ as follows. The domain $\mathsf{dom}(\gamma_c)$ consists of those infinite words of the form

$$u_0 \,\mathbf{;}\, u_1 \,\mathbf{;}\, u_2 \,\mathbf{;}\, u_3 \,\mathbf{;}\, u_4 \,\mathbf{;}\, u_5 \;\ldots$$

where each $u_i \in \Sigma^*$ is in $\mathsf{dom}(q_d)$ and the sequence of (dyadic) rationals:

$$q_d(u_0)\,,\; q_d(u_1)\,,\; q_d(u_2)\,,\; q_d(u_3)\,,\; \ldots$$

satisfies:

$$\text{for all } m \geq n \geq 0, \quad |q_d(u_m) - q_d(u_n)| \leq 2^{-n}\,. \tag{11}$$

(When condition (11) holds we say that $(q_d(u_i))_i$ is a *fast converging Cauchy sequence*.)

For such sequences $p \in \mathsf{dom}(\gamma_c)$, as defined above, define:

$$\gamma_c(p) \;=\; \lim_{n\to\infty} q_d(u_n)\,.$$

We make a couple of elementary observations about the above definition, using very basic analysis. The first point justifies that γ_c is indeed a representation.

Proposition 12.8.

1. *For every $x \in \mathbb{R}$ there exists $p \in {\Sigma_c}^\omega$ such that $x = \gamma_c(p)$.*
2. *If $p \in \mathsf{dom}(\gamma_c)$ is*

$$u_0 \,;\, u_1 \,;\, u_2 \,;\, u_3 \,;\, u_4 \,;\, u_5 \,\ldots$$

then for every $n \geq 0$, $|q_d(u_n) - \gamma_c(p)| \leq 2^{-n}$.

Proof.

1. Let $x \in \mathbb{R}$ be arbitrary. Since the dyadic rationals are dense in $\mathbb{R}$, for every $n \geq 0$, there exists some dyadic rational q_n such that $|x - q_n| \leq 2^{-(n+1)}$. Let $u_n \in \Sigma_b^*$ be such that $q_d(u_n) = q_n$. (It is easy to see that every dyadic rational is represented by some u_n.) Then

$$u_0 \,;\, u_1 \,;\, u_2 \,;\, u_3 \,;\, u_4 \,;\, u_5 \,\ldots$$

is a name for x.

2. $|q_d(u_n) - \gamma_c(p)| = |q_d(u_n) - \lim_{m\to\infty} q_d(u_m)| = \lim_{m\to\infty} |q_d(u_n) - q_d(u_m)| \leq 2^{-n}$, using (11) for the last equality.

□

The Cauchy representation γ_c is finally a *good* representation of real numbers. It gives rise to the same notion of computable real number as the binary (and decimal) representations. However, it supports a far larger collection of computable functions on real numbers.

Proposition 12.9. *A real number is γ_c-computable if and only if it is γ_b-computable.*

We shall not give the proof of this result. In one direction, seeing that γ_b-computability implies γ_c-computability is easy. The converse is quite messy, and is left as an **exercise** for the enthusiastic.

Proposition 12.10. *The function $x \mapsto 3x \colon \mathbb{R} \to \mathbb{R}$ is $(\gamma_c \to \gamma_c)$-computable.*

Proof. The required realiser $g \colon \Sigma^\omega \rightharpoonup \Sigma^\omega$ maps an infinite word

$$u_0 \,;\, u_1 \,;\, u_2 \,;\, u_3 \,;\, u_4 \,;\, u_5 \,\ldots$$

to

$$3u_2 \,;\, 3u_3 \,;\, 3u_4 \,;\, 3u_5 \,;\, 3u_6 \,\ldots$$

where $3u$ means an easily computed word in Σ_b^* representing the dyadic rational $3 \,.\, q_d(u)$. The one subtle point is that the output sequence is shifted twice to the left (it starts with $3u_2$ rather than $3u_0$). This is done to ensure that the fast Cauchy property (11) is satisfied. □

The computability of a very simple function such as $x \mapsto 3x$ is not a spectacular result. But it is the tip of the iceberg. All the major functions on $\mathbb{R}$ arising in mathematical analysis are computable relative to the γ_c representation. Because of its good behaviour, we shall take γ_c as the *standard representation* of $\mathbb{R}$. So if we talk about computability notions (semidecidable set, computable function, etc.) related to real numbers, by default we mean computability with respect to the γ_c representation. For example, Proposition 12.10 can be restated concisely as:

Proposition 12.11. *The function $x \mapsto 3x \colon \mathbb{R} \to \mathbb{R}$ is computable.*

.

12.3 The continuity theorem

Computable partial functions on $\mathbb{R}$ enjoy a continuity property analogous to Theorem 11.10. The major difference is that the notion of continuity in question is now the standard one for (partial) functions from $\mathbb{R}$ to $\mathbb{R}$, defined using the familiar ϵ-δ property.

Theorem 12.12 (Continuity theorem for $\mathbb{R}$)**.** *If $f : \mathbb{R} \rightharpoonup \mathbb{R}$ is computable then f is continuous on its domain (i.e., f is a continuous function from $\mathsf{dom}(f)$ to $\mathbb{R}$ with respect to the subspace topology on $\mathsf{dom}(f)$).*

One way of looking at this theorem is as a limitation on what is computable. Any partial function on $\mathbb{R}$ that is not continuous on its domain is necessarily non-computable.

To prepare for the proof, we give a simple definition and lemma that strengthen Proposition 12.8. We say that p of the form

$$u_0 \,;\, u_1 \,;\, u_2 \,;\, u_3 \,;\, u_4 \,;\, u_5 \,\ldots$$

with $\gamma_c(p) = x$ is a *close name* for x if for every $n \geq 0$, $|q_d(u_n) - x| \leq 2^{-(n+1)}$. (Note that the inequality is an improvement upon the $\leq 2^{-n}$ in Proposition 12.8.)

Lemma 12.13.

1. *Every x in $\mathbb{R}$ has a close name.*
2. *If*
$$u_0 \,;\, u_1 \,;\, u_2 \,;\, u_3 \,;\, u_4 \,;\, u_5 \,\ldots$$
is a close name for x then, for every $n \geq 0$, every x' with $|x' - x| < 2^{-(n+1)}$ has a name of the form
$$u_0 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, u'_{n+1} \,;\, u'_{n+2} \,;\, u'_{n+3} \,\ldots$$

Proof.

1. Let
$$u_0 \,;\, u_1 \,;\, u_2 \,;\, u_3 \,;\, u_4 \,;\, u_5 \,\ldots$$
be any name for x then
$$u_1 \,;\, u_2 \,;\, u_3 \,;\, u_4 \,;\, u_5 \,;\, u_6 \,\ldots$$
is a close name for x, by Proposition 12.8.
2. In the case that $x' < x$ then, for every $i \geq n+1$, let q_i be a dyadic rational such that $x' < q_i < x$ and $q_i - x' < 2^{-(i+1)}$, and let u'_i be a word such that $q_d(u'_i) = q_i$. The case that $x < x'$ is similar. The case that $x = x'$ is trivial.

□

Proof of Theorem 12.12. Suppose f is computable with realiser g. Suppose $f(x)$ is defined and consider any $\epsilon > 0$. We must find $\delta > 0$ such that:

$$\text{for all } x' \in \mathsf{dom}(f) \text{ with } |x' - x| < \delta, \text{ it holds that } |f(x') - f(x)| < \epsilon\,.$$

Let p be a close name for x (as given by Lemma 12.13) of the form

$$p \;=\; u_0 \,;\, u_1 \,;\, u_2 \,;\, u_3 \,;\, u_4 \,;\, u_5 \,\ldots$$

Define $r := g(p)$, which is defined because $x \in \mathsf{dom}(f)$. Then r is a name for $f(x)$ because g is a realiser for f,

Since $r \in \mathsf{dom}(\gamma_c)$, it is of the form

$$r \;=\; v_0 \,;\, v_1 \,;\, v_2 \,;\, v_3 \,;\, v_4 \,;\, v_5 \;\ldots$$

Let N be such that $2^{-N} < \epsilon/2$. Then $|q_d(v_N) - f(x)| \leq 2^{-N} < \epsilon/2$, by Proposition 12.8.

Let n be the length (number of characters) of the prefix

$$v_0 \,;\, v_1 \;\ldots\; ;\, v_N \,;$$

of r. By the continuity theorem for computable functions on infinite words (Theorem 11.10), there exists $m \geq 0$ such that, for all $p' \in \mathsf{dom}(g)$,

$$p' \restriction_m = p \restriction_m \quad \Rightarrow \quad g(p') \restriction_n = v_0 \,;\, v_1 \;\ldots\; ;\, v_N \,;$$

Let $M \geq 0$ be such that the prefix

$$u_0 \,;\, u_1 \;\ldots\; ;\, u_M \,;$$

of p has length $m' \geq m$. Then, for all $p' \in \mathsf{dom}(g)$,

$$p' \restriction_{m'} = u_0 \,;\, u_1 \;\ldots\; ;\, u_M \,; \quad \Rightarrow \quad g(p') \restriction_n = v_0 \,;\, v_1 \;\ldots\; ;\, v_N \,; \tag{12}$$

Define $\delta = 2^{-(M+1)}$. Suppose $x' \in \mathsf{dom}(f)$ is such that $|x' - x| < \delta$. By Lemma 12.13, x' has a name p' of the form

$$p' \;=\; u_0 \,;\, u_1 \;\ldots\; ;\, u_M \,;\, u'_{M+1} \,;\, u'_{M+2} \,;\, u'_{M+3} \;\ldots$$

Since $x' \in \mathsf{dom}(f)$, it holds that $g(p')\!\downarrow$ and, by (12),

$$g(p') \restriction_n = v_0 \,;\, v_1 \;\ldots\; ;\, v_N \,;$$

As g realises f, we have that $g(p')$ is a name for $f(x')$. By Proposition 12.8, we have that $|q_d(v_N) - f(x')| \leq 2^{-N} < \epsilon/2$. Since also $|q_d(v_N) - f(x)| < \epsilon/2$, we have that $|f(x') - f(x)| < \epsilon$, as required. $\square$

13 Algorithmic Information Theory

13.1 Kolmogorov complexity

Definition 13.1 (Universal computable function $\{0,1\}^* \rightharpoonup \{0,1\}^*$)**.** A computable partial function $u\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ is *universal* if, for every computable partial function $f\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$, there exists a word $v_f \in \{0,1\}^*$ such that

$$f(w) \;\simeq\; u(v_f\, w)\,, \quad \text{for all } w \in \{0,1\}^* \;.$$

Proposition 13.2. *There exists a universal computable function $u\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$.*

Definition 13.3 (Kolmogorov complexity)**.** Let $u\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ be a universal computable function. The *Kolmogorov complexity* relative to u of a word $w \in \{0,1\}^*$ is:

$$K_u(w) \;:=\; \min\{|v| \mid u(v) = w\} \;.$$

The definition is dependent on the choice of universal function u, but any two universal functions give rise to notions of Kolmogorov complexity that differ only by an additive constant.

Proposition 13.4. *Suppose $u, u'\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ are universal computable functions. There exists $c \geq 0$ such that, for all $w \in \{0,1\}^*$, we have $|K_{u'}(w) - K_u(w)| \leq c$; more briefly: $K_{u'}(w) = K_u(w) + O(1)$.*

Proof. Because u is universal and u' is computable, there exists $v_{u'}$ such that $u'(w) \simeq u(v_{u'}\, w)$ for all w. Similarly, there exists v'_u such that $u(w) \simeq u'(v'_u\, w)$ for all w. So $c = \max(|v_{u'}|, |v'_u|)$ has the required property. □

We henceforth fix a preferred universal computable function u and write $K(w)$ for $K_u(w)$.

The length of a string provides an upper bound for its Kolmogorov complexity (modulo an additive constant).

Proposition 13.5. *There exists c such that, for any $w \in \{0,1\}^*$, we have $K(w) \leq |w| + c$; more briefly: $K(w) \leq |w| + O(1)$.*

Proof. Since the identity function $\mathsf{id}(w) = w$ is computable, $u(v_{\mathsf{id}}\, w) = w$. So $c = |v_{\mathsf{id}}|$ satisfies the required property. □

By the above result, we can think of any word w for which $K(w) \geq |w|$ as a word with high Kolmogorov complexity (within a constant c of the highest possible value it can take). The next result observes that at least one string of any length has high complexity in this sense.

Proposition 13.6. *For any $n \geq 0$, there exists $w \in \{0,1\}^*$ with $|w| = n$ for which $K(w) \geq n$.*

Proof. There is at most one word with Kolmogorov complexity 0 (namely $u(\varepsilon)$ if this is defined), at most two words with complexity 1 (namely $u(0)$ and $u(1)$ if defined and different from $u(\varepsilon)$), and in general at most 2^n with complexity n. Thus the number of words with complexity $< n$ is $\sum_{i=0}^{n-1} 2^i = 2^n - 1$. Since there are 2^n words of length n, at least one of them has complexity $\geq n$. □

A similar argument show that at least $2^{n-1}+1$ words of length n have complexity $\geq n-1$, that is the majority of words of length n have such close-to-maximum complexity.

The next result observes that when a computable function on words maps an input word to an output word, the Kolmogorov complexity increases by at most an additive constant.

Proposition 13.7. *If $f\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ is computable, then there exists c such that, for all $w \in \mathsf{dom}(f)$, it holds that $K(f(w)) \leq K(w) + c$; more briefly $K(f(w)) \leq K(w) + O(1)$.*

Proof. Because $f(u(w)) \simeq u(v_{f \circ u}\, w)$ for all w, the constant $c = |v_{f\circ u}|$ has the required property. □

Komolgorov complexity provides one (theoretical) perspective on the (practical) problem of data compression. We might view a word w such that $K(w) \ll |w|$ as *compressible* in the sense that there exist words v much shorter than w from which w can be recovered as $w = u(v)$. If instead $K(w) \approx |w|$ then no such short word exists, so w is *incompressible*. (As usual, $\ll$ means much smaller than and $\approx$ means approximately equal to.) Kolmogorov wondered if it is possible to find infinite bit sequences that are of such high complexity throughout that every prefix is incompressible. This is formulated precisely by the definition below.

Definition 13.8 (Kolmogorov incompressibility)**.** An ω-word $p \in \{0,1\}^\omega$ is *Kolmogorov incompressible* if there exists c such that, for all $n \geq 0$, it holds that $K(p\restriction_n) \geq n - c$; more briefly $K(p\restriction_n) \geq n - O(1)$.

It turns out, however, that the above property is impossible to satisfy. This was proved by Martin-Löf.

Proposition 13.9 (Martin-Löf)**.** *There are no Kolmogorov incompressible sequences.*

It turns out that the problem with Definition 13.8 is that the notion of Kolmogorov complexity is too crude to allow the definition to work. We next look at a more subtle notion of complexity, called *prefix-free* complexity

13.2 Prefix-free complexity

Prefix-free complexity is a variation on the idea of Kolmogorov complexity introduced by Chaitin. The idea of Kolmogorov complexity is to measure the amount of information in a word w, in terms of the number of bits needed to encode w via a word v that is decoded by the universal function $u(v) = w$. However, there is a valid viewpoint that giving the word v as input provides more information than just $|v|$ bits. For example, if we were to input v on a keyboard, we would type in $|v|$ bits and press the enter key. So essentially we have communicated v bits together with the additional information that our input has finished. Chaitin's idea was to vary the model of a universal function so that all the information is encoded in the input word. That is, an input word $w \in \{0,1\}^*$ should be *self-delimiting*: when we type in a legitimate input w such that $u(w)$ is defined, the universal function u knows purely on the basis of the word w itself that we have completed the input. It follows that no proper prefix of w' of w can also be a legitimate input of u, otherwise u would not allow us to complete the entry of the input w. It also follows that no word w'' that has w as a proper prefix can be an input. These points together say that the domain of u must be a *prefix-free* set in the sense of the definition below.

A set $W \subseteq \{0,1\}^*$ of words is said to be *prefix free* if, for every pair of distinct $w, w' \in W$, it holds that neither w nor w' is a prefix of the other. A partial function $f\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ is said to be *prefix free* if its domain is prefix free. Note that for any prefix-free $u\{0,1\}^* \rightharpoonup \{0,1\}^*$ and word v, the partial function $w \mapsto u(v\,w)$ is also prefix free.

Definition 13.10 (Universal computable prefix-free function $\{0,1\}^* \rightharpoonup \{0,1\}^*$)**.** A computable prefix free partial function $u\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ is *universal* if, for every computable prefix free partial function $f\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$, there exists a word $v_f \in \{0,1\}^*$ such that

$$f(w) \;\simeq\; u(v_f\, w)\,, \quad \text{for all } w \in \{0,1\}^* \;.$$

Proposition 13.11. *A universal computable prefix free function $u\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ exists.*

Definition 13.12 (Prefix-free complexity)**.** Let $u\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ be a universal computable prefix-free function. The *prefix-free complexity* relative to u of a word $w \in \{0,1\}^*$ is:

$$C_u(w) \;:=\; \min\{|v| \mid u(v) = w\} \;.$$

As before, any two universal functions give rise to complexity measures that differ only by an additive constant. We omit the proof, which is identical to that of Proposition 13.4.

Proposition 13.13. *Suppose $u, u'\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ are universal computable prefix-free functions. There exists $c \geq 0$ such that, for all $w \in \{0,1\}^*$, we have $|C_{u'}(w) - C_u(w)| \leq c$; more briefly: $C_{u'}(w) = C_u(w) + O(1)$.*

We henceforth fix a preferred universal computable prefix-free function $u\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ and write $C(w)$ for $C_u(w)$.

The upper bound for prefix-free complexity is more subtle than that for Kolmogorov complexity (Proposition 13.5). For this lecture, we satisfy ourselves with the following statement, which is weaker than the optimal bound.

Proposition 13.14. *For any real $d > 1$, there exists c such that, for all $w \in \{0,1\}^*$, we have $C(w) \leq d\,|w| + c$; more briefly: $C(w) \leq o(|w|)$.*

Proof. Given a real $d > 1$, let N be such that $N < d \log_2(2^N - 1)$. Use blocks of N bits to encode a $2^N - 1$-character alphabet Σ, with every word in $\{0,1\}^N$ that contains a 1 representing a character. The word 0^N is reserved as an end-of-input symbol. Encode words $w \in \{0,1\}^*$ as words $w' \in \Sigma^*$ in a sensible way so that $|w'| \leq \lceil |w| / \log_2(2^N - 1) \rceil$, and so that the decoding function $\Sigma^* \to \{0,1\}^*$ is computable. We now define a prefix-free function $f\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ whose domain consists of words of length $(k+1)N$ for some $k \geq 0$, in which each of the first k blocks of N bits contains a 1 and whose last N bits are all 0. Such a $(k+1)N$-bit word thus represents a word $a_0 \ldots a_{k-1} \in \Sigma^k$ followed by the end-of-input symbol. The function f returns the word in $\{0,1\}^*$ of length $\geq (k-1)\log_2(2^N - 1)$ encoded by $a_0 \ldots a_{k-1} \in \Sigma^*$. Writing v for the $(k+1)N$-bit input word , we see that

$$|v| \;=\; (k+1)N \;\leq\; d(k-1)\log_2(2^N - 1) + 2N \;\leq\; d\,|f(v)| + 2N \;.$$

Since the above function f is prefix free, we there exists v_f such that $u(v_f\, w) = f(w)$. So the constant $2N + |v_f|$ satisfies the required property. □

Once again, we view strings w whose complexity is close to $|w|$ as being complex, and such strings exist with an identical proof to that of Proposition 13.6.

Proposition 13.15. *For any $n \geq 0$, there exists $w \in \{0,1\}^*$ with $|w| = n$ for which $C(w) \geq n$.*

As for Kolmogorov complexity, computable functions increase prefix-free complexity by at most an additive constant.

Proposition 13.16. *If $f\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ is computable, then there exists c such that, for all $w \in \mathsf{dom}(f)$, it holds that $C(f(w)) \leq C(w) + c$; more briefly $C(f(w)) \leq C(w) + O(1)$.*

The proof is the same as that of Proposition 13.6. It works because the function $f \circ u$ is prefix free. (For any partial function f and prefix-free function g, the composite function $f \circ g$ is prefix free.)

One of Chaitin's motivations for introducing prefix-free complexity was to define a meaningful notion of incompressibility for infinite sequences, adapting Definition 13.8 in the obvious way.

Definition 13.17 (Prefix-free incompressibility)**.** An ω-word $p \in \{0,1\}^\omega$ is *prefix-free incompressible* if there exists c such that, for all $n \geq 0$, it holds that $C(p\restriction_n) \geq n - c$; more briefly $C(p\restriction_n) \geq n - O(1)$.

In contrast to the situation for Kolmogorov incompressibility, prefix-free incompressible ω-words do exist.

A natural example of an incompressible ω-word is given by the following construction due to Chaitin. We define a real number in the interval $[0,1]$:

$$\Omega \;:=\; \sum_{w \in \mathsf{dom}(u)} 2^{-|w|} \;. \tag{13}$$

This indeed defines a number ≤ 1 because the set $\mathsf{dom}(u)$ is prefix free. This is known as Kraft's inequality. One way of seeing that it is true is as follows. The prefix free property tells us that for distinct $w, w' \in \mathsf{dom}(u)$, we have $\langle w\rangle \cap \langle w'\rangle = \emptyset$; i.e., the words in $\mathsf{dom}(u)$ have disjoint cylinder sets. The formula (13) thus calculates the probability that a randomly generated (by fair coin tosses) $p \in \{0,1\}^\omega$ lands in $\mathsf{dom}(u)$. Since the sum is a probability it is ≤ 1.

The interpretation as a probability also leads to a nice conceptual description of Ω: it is the probability that if we feed a randomly generated $r \in \{0,1\}^\omega$ bit-by-bit to the universal prefix-free function u, we shall eventually, after n bits say, arrive at an input word $r\restriction_n$ in the domain of u, resulting in a terminating computation $u(r\restriction_n)$. Accordingly, Ω is often referred to as Chaitin's *halting probability*.

It can be proven that Ω as defined above is an irrational (indeed non-computable) number. Accordingly, it can be given a unique binary expansion $p^\Omega \in \{0,1\}^\omega$; i.e., we have:

$$\Omega \;=\; \sum_{i=0}^{\infty} p^\Omega_i \, 2^{-(i+1)} \;.$$

Theorem 13.18 (Chaitin)**.** *p^Ω is a prefix-free-incompressible sequence.*

Proof. We define a computable partial function $f\colon \{0,1\}^* \rightharpoonup \{0,1\}^*$ with the following behaviour. For any $n \geq 0$, it holds that $f(p^\Omega\restriction_n)$ is defined and $C(f(p^\Omega\restriction_n)) > n$.

In outline, f runs the following algorithm. Let $w_0, w_1, w_2, \ldots$ be a computable enumeration of $\mathsf{dom}(u)$ without repetitions. (Since $\mathsf{dom}(u)$ is infinite and the domain of a partial

computable function, it has such an enumeration.) Given an input word $p_0 \ldots p_{n-1}$ of length n in $\{0,1\}^*$, the algorithm finds the smallest $N \geq 0$ such that

$$\sum_{i=0}^{N-1} 2^{-|w_i|} \;\geq\; \sum_{i=0}^{n-1} p_i \cdot 2^{-(i+1)} \;\;,$$

if such N exists. The algorithm then returns the first $v \in \{0,1\}^*$, according to a standard enumeration of the set $\{0,1\}^*$, with the property that $v \notin \{u(w_0), \ldots, u(w_{N-1})\}$.

To show that this algorithm enjoys the claimed property, suppose $p_0 \ldots p_{n-1} = p^\Omega \restriction_n$. Then N as above exists by the definition of Ω, and indeed

$$\sum_{i=0}^{n-1} p_i \cdot 2^{-(i+1)} \;\leq\; \sum_{i=0}^{N-1} 2^{-|w_i|} \;<\; \Omega \;<\; \sum_{i=0}^{n-1} p_i \cdot 2^{-(i+1)} + 2^{-n} \;\;.$$

(The strict inequalities here are because Ω is irrational.) Since (by 13) any $w \in \mathsf{dom}(u)$ contributes a weight of $2^{-|w|}$ to Ω, and since $\Omega - \sum_{i=0}^{N-1} 2^{-|w_i|} < 2^{-n}$, it follows that every w_i with $i \geq N$ is such that $|w_i| > n$. Thus, for any $v \in \{0,1\}^*$ with the property that $v \notin \{u(w_0), \ldots, u(w_{N-1})\}$, we have $C(v) > n$, because such a v can only arise as $v = u(w)$ for w with $|w| > n$.

Since the algorithm for f returns $v \in \{0,1\}^*$ with the property that $v \notin \{u(w_0), \ldots, u(w_{N-1})\}$, it does indeed hold that

$$C(f(p^\Omega \restriction_n)) > n \;\;. \tag{14}$$

By Proposition 13.16, there exists c such that, for all $n \geq 0$,

$$C(f(p^\Omega \restriction_n)) \;\leq\; C(p^\Omega \restriction_n) + c \;\;. \tag{15}$$

Putting (14) and (15) together, for all $n \geq 0$,

$$C(p^\Omega \restriction_n) \;>\; n - c \;\;;$$

i.e., p^Ω is prefix-free incompressible. □

14 Algorithmic Randomness

14.1 Random sequences

Consider a sequence $p = p_0\, p_1\, p_2 \cdots \in \{0,1\}^\omega$ obtained by tossing a fair coin *ad infinitum*. Such a sequence will be called a *random sequence*. What properties should we expect such a sequence to satisfy?

One very simple example property is that we would expect such a random p to be different from the sequence 0^ω. One can justify this expectation as follows.

- For any $n \geq 0$, the probability that a randomly generated p satisfies $p\restriction_n = 0^n$ is 2^{-n}.
- So, the property $p = 0^\omega$ determines a *non-randomness test* as follows. Given a sequence $p \in \{0,1\}^\omega$, and any small $\epsilon > 0$, considered as the probability of making an error (an *error threshold*), we find n such that $2^{-n} \leq \epsilon$ and examine $p\restriction_n$. If $p\restriction_n = 0^n$ then we have detected that p is unlikely to be random, with probability of error $\leq \epsilon$.
- Since the sequence 0^ω passes this test for every $\epsilon > 0$, we can detect that 0^ω is non-random up to any error threshold.
- We take this as justification that 0^ω is non-random.

The mathematical structure of the above *non-randomness test* is as follows.

- We have a sequence $T_0, T_1, T_2, \ldots$ of subsets of $\{0,1\}^\omega$

$$T_n \;=\; \langle 0^n \rangle \;:=\; \{p \in \{0,1\}^\omega \mid p\restriction_n = 0^n\} \;.$$

- Given p and n with $p \in T_n$, we can see that $p \in T_n$ is true using a finite-time observation.
- It holds that $\lim_{n\to\infty} \lambda(T_n) = 0$, where $\lambda(T_n)$ is the probability that a randomly generated p belongs to T_n.
- $0^\omega \in \bigcap_{n=0}^{\infty} T_n$ (in fact $\{0^\omega\} = \bigcap_{n=0}^{\infty} T_n$).

We now consider a more sophisticated example of a property of random sequences. For $p \in \{0,1\}^\omega$ and $n > 0$ define

$$n\text{-}\mathsf{mean}(p) \;:=\; \frac{1}{n}\sum_{i=0}^{n-1} p_i \;.$$

We shall argue that every random p satisfies the property

$$\text{the sequence } (n\text{-}\mathsf{mean}(p))_n \text{ converges and } \lim_{n\to\infty} n\text{-}\mathsf{mean}(p) = \frac{1}{2}\,. \tag{16}$$

Property (16) is known as the *law of large numbers*. Accordingly, we write $\mathsf{LLN}(p)$ to say that p satisfies property (16).

Again we justify that $\mathsf{LLN}(p)$ holds for random p, by showing that we can detect the failure of $\mathsf{LLN}(p)$ using a *non-randomness test*.

Suppose that $\mathsf{LLN}(p)$ fails for a sequence $p \in \{0,1\}^\omega$. This means that there exists $\delta > 0$ such that

$$\left| n\text{-}\mathsf{mean}(p) - \frac{1}{2} \right| > \delta \;\text{ for infinitely many } n\,. \tag{17}$$

If a sequence p satisfies (17) then we say that it has δ-*bias*, and we write δ-$\mathsf{bias}(p)$ to say that this holds.

We construct a *non-randomness test* showing that every sequence with δ-bias is non-random.

- Define:

$$T_n \;=\; \left\{ p \in \{0,1\}^\omega \mid \text{for at least } n \text{ different values of } i,\ \left| i\text{-}\mathsf{mean}(p) - \frac{1}{2} \right| > \delta \right\} \quad .$$

- Given p and n such that $p \in T_n$, we can verify that $p \in T_n$ in finite time, by searching for n different values of i such that $\left|i\text{-}\mathsf{mean}(p) - \frac{1}{2}\right| > \delta$. If $p \in T_n$ then eventually this search will find such values $i_1 < i_2 < \cdots < i_n$. (Note that the search will not terminate in the case that $p \notin T_n$.)

- It holds that $\lim_{n\to\infty} \lambda(T_n) = 0$. We take this on trust, as its proof requires probability-theory calculations that are outside the scope of this course.

- $\{p \in \{0,1\}^\omega \mid \delta\text{-}\mathsf{bias}(p)\} \subseteq \bigcap_{n=0}^{\infty} T_n$. (Again this is actually an equality of sets.)

As in the first example, this test is applied as follows to a sequence p.

- Given any *error threshold* $\epsilon > 0$, find $n \geq 0$ such that $\lambda(T_n) \leq \epsilon$. Check to see if $p \in T_n$, which if true can be verified in finite time. If so, conclude that p is non-random with probability of error $\leq \epsilon$.

- Since every sequence with δ-bias passes the test for every $\epsilon > 0$, we can detect that every such sequence is non-random up to any error threshold.

- We take this as justification that every sequence with δ-bias is non-random.

By the contrapositive, random sequences do not have δ-bias for any $\delta > 0$. Therefore, every random sequence p satisfies $\mathsf{LLN}(p)$.

14.2 Naïve non-randomness tests

The above examples (and other similar ones) suggest the following natural notion of non-randomness test, which we give at this point even though some of the concepts involved (the Cantor topology, the probability function λ) have not been formally defined.[8] Full definitions will be given in Section 14.4.

Definition 14.1 (Non-randomness test, temporary definition)**.** A *non-randomness test* is given by a sequence $(T_n)_{n\geq 0}$ of subsets of $\{0,1\}^\omega$ such that:

1. Every T_n is open in the Cantor topology on $\{0,1\}^\omega$, and
2. $\lim_{n\to\infty} \lambda(T_n) = 0$, where $\lambda(T_n)$ is the probability that a randomly generated p lies in the set T_n.

A sequence $p \in \{0,1\}^\omega$ is said to *satisfy* the test if $p \in \bigcap_{n=0}^{\infty} T_n$.

[8] Actually, the Cantor topology was considered in exercises 4 and 5 of Tutorial 9.

The way in which such a general non-randomness test is applied follows the same pattern as before.

- Given any *error threshold* $\epsilon > 0$, find $n \geq 0$ such that $\lambda(T_n) \leq \epsilon$. Check to see if $p \in T_n$, which, *because* T_n *is open*, can be verified if true in finite time. If so, conclude that p is non-random with probability of error $\leq \epsilon$.
- If a sequence p passes the test for every $\epsilon > 0$, we can detect that p is non-random up to any error threshold.

This leads to the following *tentative* general definition of random sequence.

Definition 14.2 (Random sequence, temporary definition)**.** A sequence $p \in \{0,1\}^\omega$ is said to be:

- *non-random* if there exists a non-randomness test that p satisfies;
- *random* if it does not satisfy any non-randomness test.

The reason that the above definitions are labelled as temporary is that there is a fundamental problem with them.

Proposition 14.3. *According to Definitions 14.1 and 14.2, no sequence is random.*

Proof. Let $q \in \{0,1\}^\omega$ be arbitrary. Define

$$T_n \;=\; \langle q\restriction_n \rangle \;:=\; \{p \in \{0,1\}^\omega \mid p\restriction_n = q\restriction_n\} \;.$$

Then $(T_n)_n$ is a non-randomness test that q satisfies. □

14.3 Martin-Löf tests

In spite of the negative result of Proposition 14.3, the above approach to characterising randomness using non-randomness tests is not fundamentally wrong. However, we need to be a little more careful about how we define non-randomness tests.

If a non-randomness test is to be applicable in practice to test a sequence $p \in \{0,1\}^\omega$, then the following additional properties should hold.

(A) Given any *error threshold* $\epsilon > 0$, it should be possible to calculate from ϵ an appropriate $n \geq 0$ such that $\lambda(T_n) \leq \epsilon$.

(B) There should be an algorithm for testing (semideciding), given $p \in \{0,1\}^\omega$ and n, whether $p \in T_n$.

Note that the second property does not apply to the tests used in the proof of Proposition 14.3 in the case that q is a non-computable sequence.

Thus we shall build in notions from computability theory to define a practicable notion of non-randomness test.

Definition 14.4 (Martin-Löf test)**.** A *Martin-Löf (non-randomness) test* (*M-L test* for short) is a sequence $(T_n)_{n\geq 0}$ of subsets of $\{0,1\}^\omega$ such that:

- $(T_n)_n$ is a *computable sequence* of *computable open sets*.
- $\lim_{n\to\infty} \lambda(T_n) = 0$ with a *computable rate of convergence*.

We remark that the first bullet point in the definition implements property (B) above, and the second implements property (A).

14.4 Supporting definitions

14.4.1 The probability function λ

The aim of this section is to define, for every open set T, the probability $\lambda(T)$ that a randomly generated p belongs to T.

For cylinder sets, the definition is simple:

$$\lambda(\langle w\rangle) \;=\; 2^{-|w|} \;.$$

Lemma 14.5. *Every open set T is a countable disjoint union of cylinder sets; i.e., there exists $G \subseteq \{0,1\}^*$ such that:*

- $T = \bigcup\{\langle w\rangle \mid w \in G\}$, *and*
- *for all $w, w' \in G$ with $w \neq w'$, it holds that $\langle w\rangle \cap \langle w'\rangle = \emptyset$.*

Note that any such G is automatically countable because $\{0,1\}^*$ is a countable set.

Proof. If T is open then $T = \bigcup\{\langle w\rangle \mid w \in G'\}$ holds for some set G', for which the disjointness property may fail. Define:

$$G \;=\; \{w \in G' \mid \text{no proper prefix of } w \text{ is in } G'\} \;.$$

The correctness of this definition is left as an **exercise**. □

To define the probability of an arbitrary open set T, we find G such that

$$T \;=\; \bigcup\{\langle w\rangle \mid w \in G\} \;\text{ a disjoint union}$$

and we then define

$$\lambda(T) \;=\; \sum_{w\in G} \lambda(\langle w\rangle) \;.$$

It will be proved in the exercise class that this is a good definition; i.e., if

$$\bigcup\{\langle w\rangle \mid w \in G\} \;=\; \bigcup\{\langle w\rangle \mid w \in G'\} \;\text{ both disjoint unions}$$

then

$$\sum_{w\in G} \lambda(\langle w\rangle) \;=\; \sum_{w\in G'} \lambda(\langle w\rangle) \;.$$

(A remark for the measure-theoretically informed reader. The above definition coincides with defining $\lambda(T)$ as the measure assigned to the open set T by the uniform probability measure on the Borel sets of $\{0,1\}^\omega$.)

14.4.2 Computable open sets

We write $\mathcal{O}(\{0,1\}^\omega)$ for the set of open subsets of $\{0,1\}^\omega$. Consider the alphabet

$$\Sigma_o := \{0, 1, \mathbf{;}, \emptyset\} \;.$$

We define a representation

$$\gamma_o \colon {\Sigma_o}^\omega \rightarrow\!\!\!\!\!\rightarrow \mathcal{O}(\{0,1\}^\omega) \;.$$

$\mathsf{dom}(\gamma_o)$ consists of those sequences in ${\Sigma_o}^\omega$ of the form

$$p \;=\; w_0\,;w_1\,;w_2\,;w_3\,;w_4\,;\ldots$$

where each w_i is either the symbol $\emptyset$ or a word in $\{0,1\}^*$.

For such a sequence $p \in \mathsf{dom}(\gamma_o)$, we define:

$$\gamma_o(p) \;=\; \bigcup_{i\geq 0}\langle w_i\rangle \;,$$

where by convention we define $\langle\emptyset\rangle := \emptyset$.

An open set $T \in \mathcal{O}(\{0,1\}^\omega)$ is defined to be *computable* if it is γ_o-computable in the sense of Definition 12.3.

It is an interesting fact that the computable open sets coincide with the semidecidable sets in the sense of Lecture 11.

14.4.3 Computable sequence of open sets

We define a representation

$$\gamma_o^\omega \colon {\Sigma_o}^\omega \twoheadrightarrow (\mathcal{O}(\{0,1\}^\omega))^\omega \;.$$

Recall the pairing function $p\colon \mathbb{N}^2 \to \mathbb{N}$ defined in Lecture 4. Using this, we consider a sequence $s \in {\Sigma_o}^\omega$ as representing an infinite sequences

$$\pi_0(s)\;\pi_1(s)\;\pi_2(s)\;\pi_3(s)\;\ldots$$

of sequences in ${\Sigma_o}^\omega$, by defining

$$\pi_i(s) \;:=\; s_{p(i,0)}\,s_{p(i,1)}\,s_{p(i,2)}\,s_{p(i,3)}\;\cdots$$

The representation γ_0^ω is defined on those sequences s satisfying:

$$\text{for every } i,\;\; \pi_i(s) \in \mathsf{dom}(\gamma_o)\,.$$

For such a sequence s, define

$$\gamma_0^\omega(s) \;:=\; \gamma_o(\pi_0(s))\;\gamma_o(\pi_1(s))\;\gamma_o(\pi_2(s))\;\gamma_o(\pi_3(s))\;\ldots$$

A sequence $(T_n)_{n\geq 0}$ is said to be a *computable sequence of open sets* if it is γ_0^ω-computable.

Lemma 14.6. *If $(T_n)_{n\geq 0}$ is a computable sequence of open sets then every T_n is a computable open set.*

The proof, which is left as an **exercise**, is a consequence of the computability of the pairing function p.

14.4.4 Computable rate of convergence

Rather than giving the general definition of computable rate of convergence for an arbitrary Cauchy sequence, we consider only the case that the limit of the Cauchy sequence is 0, which is the only case we need.

Definition 14.7. A Cauchy sequence $(x_n)_n$ in $[0,1]$ with limit 0 has a *computable rate of convergence* if there exists a computable increasing function $r\colon \mathbb{N} \to \mathbb{N}$ satisfying: for any $n \geq 0$ and $m \geq r(n)$, we have $x_m \leq 2^{-n}$.

14.5 Martin-Löf Randomness

Definition 14.8 (Martin-Löf random)**.** A sequence is *Martin-Löf random* (*M-L random* for short) if it satisfies no M-L test.

Proposition 14.9. *If $q \in \{0,1\}^\omega$ is computable then q is not M-L random.*

Proof. Let $q \in \{0,1\}^\omega$ be computable. Define

$$T_n \;=\; \langle q\restriction_n \rangle \;:=\; \{p \in \{0,1\}^\omega \mid p\restriction_n = q\restriction_n\} \;.$$

Because q is computable, it holds that $(T_n)_n$ is a computable sequence of open sets. Also $\lambda(T_n) = 2^{-n}$, so $\lim_{n\to\infty} \lambda(T_n) = 0$ with $r(n) = n$ giving a computable rate of convergence. Finally, $q \in \bigcap_{n\geq 0} T_n$. □

A fundamental property of Martin-Löf randomness is that there is a single M-L test that is satisfied by every non-M-L-random sequence. Such a test is called a *universal test*.

Definition 14.10 (Universal test)**.** An M-L test $(T_n)_n$ is *universal* if it is satisfied by every sequence that is not M-L random.

Theorem 14.11 (Martin-Löf 1966)**.** *There exists a universal M-L test $(T^U_n)_n$.*

The universal test is thus a single test that can always be used as a statistical test of non-randomness.

The proof of Theorem 14.11 is not terribly difficult. However, it would need an additional lecture to address it. In the absence of this, we content ourselves with considering some consequences of the existence of the universal test.

Corollary 14.12.

1. *The set*
$$R_{ML} \;:=\; \{p \in \{0,1\}^\omega \mid p \text{ is M-L random}\}$$
is a countable union of closed subsets of $\{0,1\}^\omega$.
2. *$\lambda(R_{ML}) = 1$, where λ is the uniform (Borel) probability measure on $\{0,1\}^\omega$.*
3. *There exists an M-L-random sequence.*

Proof.

1. By the definition of the universal test, the set of non-random sequences $\{0,1\}^\omega - R_{\mathrm{ML}}$ satisfies
$$\{0,1\}^\omega - R_{\mathrm{ML}} \;=\; \bigcap_{n\geq 0} T^U_n \;.$$
Since $\{0,1\}^\omega - R_{\mathrm{ML}}$ is a countable intersection of open sets, its complement R_{ML} is a countable union of closed sets.[9]

[9] Subsets that arise as counbtable unions of closed sets are known as F_σ *sets.*

2. By the monotonicity of Borel measure, for all $m \geq 0$,

$$\lambda\left(\bigcap_{n\geq 0} T_n^U\right) \;\leq\; \lambda(T_m^U) \;.$$

As $(T_n^U)_n$ is an M-L test, we have $\lim_{n\to\infty} \lambda(T_n^U) = 0$ hence

$$\lambda(\{0,1\}^\omega - R_{\mathrm{ML}}) \;=\; \lambda\left(\bigcap_{n\geq 0} T_n^U\right) \;=\; 0 \;.$$

Hence, by finite additivity of Borel measure,

$$\lambda(R_{\mathrm{ML}}) \;=\; 1 - \lambda(\{0,1\}^\omega - R_{\mathrm{ML}}) \;=\; 1 \;.$$

3. Since R_{ML} has positive measure it cannot be empty.[10]

□

We now know that Martin-Löf-random sequences exist and are almost surely (i.e., with probability 1) produced by randomly tossing a fair coin *ad infinitum*. Is it possible to give an explicit definition of an ML-random sequence? In fact it is. The sequence p^Ω defined from Chaitin's halting probability Ω in Note 13 is one such sequence. This follows from the following deep and beautiful connection between ML-randomness and incompressibility, due to Schnorr.

Theorem 14.13 (Schnorr)**.** *The following are equivalent for any $p \in \{0,1\}^\omega$.*

1. *p is ML-random.*
2. *p is prefix-free incompressible.*

So randomness defined in terms of failing any statistical tests for non-randomness (ML-randomness) coincides with randomness in terms of not exhibiting any patterns that can be exploited for compression purposes (prefix-free incompressibility). The proof of this theorem is quite involved, and beyond the scope of this course. Nevertheless, as a bridge between the last two topics we have covered, and as a result that connects computability theory with topics of wider interest, the statement of this theorem seems a fitting end for the course.

[10] In fact, it is a standard fact that every set of positive measure, under any non-atomic probability measure on a complete separable metric space, has cardinality $2^{\aleph_0}$. Thus there are $2^{\aleph_0}$-many ML-random sequences.