

A user-friendly Introduction to (un)Computability and Unprovability via “Church’s Thesis”

Computability is the part of logic that gives a mathematically precise formulation to the concepts *algorithm*, *mechanical procedure*, and *calculable function* (or relation). Its advent was strongly motivated, in the 1930s, by Hilbert’s program, in particular by his belief that the *Entscheidungsproblem*, or *decision problem*, for axiomatic theories, that is, the problem “Is this formula a theorem of that theory?” was solvable by a mechanical procedure that was yet to be discovered.

Now, since antiquity, mathematicians have invented “mechanical procedures”, e.g., Euclid’s algorithm for the “greatest common divisor”,[†] and had no problem recognising such procedures when they encountered them. But how do you mathematically *prove* the *nonexistence* of such a mechanical procedure for a particular problem? You need a *mathematical formulation* of what *is* a “mechanical procedure” in order to do that!

Intensive activity by many (Post [11, 12], Kleene [8], Church [2], Turing [17], Markov [9]) led in the 1930s to several alternative formulations, each purporting to mathematically characterise the concepts *algorithm*, *mechanical procedure*, and *calculable function*. All these formulations were quickly proved to be equivalent; that is, the calculable functions admitted by any one of them were the same as those that were admitted by any other. This led Alonzo Church to formulate his conjecture, famously known as “Church’s Thesis”, that any *intuitively* calculable function is also calculable within any of these mathematical frameworks of calculability or computability.[‡]

[†]That is, the largest positive integer that is a common divisor of two given integers.

[‡]I stress that even if this sounds like a “completeness *theorem*” in the realm of computability, it is not. It is just an empirical belief, rather than a provable result. For example, Péter [10] and Kalmár [7], have argued that it is conceivable that the intuitive concept of calcu-

By the way, Church proved ([1, 2]) that Hilbert’s *Entscheidungsproblem* admits no solution by functions that are calculable within any of the known mathematical frameworks of computability. Thus, if we accept his “thesis”, the Entscheidungsproblem admits no algorithmic solution, period!

The eventual introduction of computers further fueled the study of and research on the various mathematical frameworks of computation, “models of computation” as we often say, and “computability” is nowadays a vibrant and very extensive field.

1.1. Turing machines

We will very briefly describe a formal (not “real” as in “commercially available”) programming language, the so-called Turing machine. It is one of the earliest formalisations of the concept of “algorithm” and “computation” and is due to Alan Turing [17].



One way to think of a *Turing machine*, or “TM” for short, is as an abstract model of a “computer program”; indeed the Turing formalism is a *programming language*, and any *specific* TM is a program written in said language—a very *primitive* one, more about which shortly.



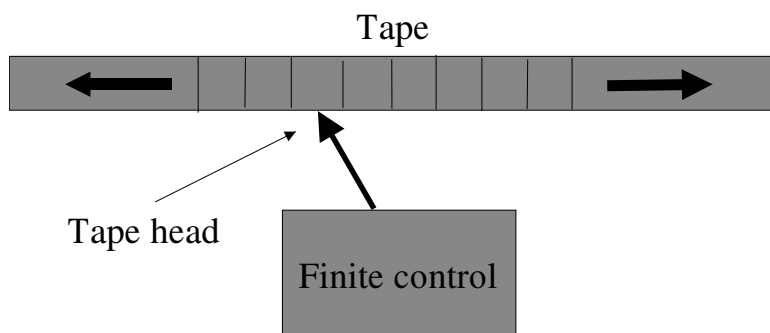
Like a computer, this “machine” can faithfully carry out simple instructions. To avoid technology-dependent limitations, it is defined so that it has unbounded “memory” (or storage space). Thus, it never runs out of storage during a computation.

OK. Temporarily thinking of a TM, M , as a “machine” we can describe it—informally at first—as follows:

M consists of

- (a) An infinite two-way tape.
- (b) A read/write tape-head.
- (c) A “black-box”, the finite control, which can be at any one of a (fixed) *finite* set of internal states (or just, *states*). Pictorially, a TM is often represented as in the figure below:

ability may in the future be extended so much as to transcend the power of the various mathematical models of computation that we currently know.



The tape is subdivided into squares, each of which can hold a *single symbol* out of a finite set of admissible symbols associated with the TM—the *tape-alphabet*, Γ .

The tape-head can scan only *one square* at a time. The TM tape corresponds, as we will see, to a variable of *string type* in an actual programming language. In *one move*, the tape-head can move to scan the next square, to the left or to the right of the present square, but it has the option to stay on the current square.

The tape-head can read or write a symbol on the scanned square. Writing a symbol is assumed to erase first what was on the square previously.

There is a distinguished alphabet symbol, the blank—denoted by B —which appears everywhere except in a finite set of tape squares. This symbol is not used as “an input symbol”.

The machine *moves*, at each computation step, as follows:

Depending on

(a) The currently scanned symbol

and

(b) The current state

the machine will:

- (i) Write a symbol or leave the symbol unchanged on the scanned square
- (ii) Enter a (possibly) new state
- (iii) Move the head to the left or right or leave it stationary.

We shall require our TMs in this Note to be *deterministic*, i.e., that they behave as follows: Given the current symbol/state pair, they have a *uniquely defined* response.

A TM computation *begins* by positioning the tape-head on the left-most non blank symbol on the tape (that is, the first symbol of the input string), “initialising” the machine (by putting it in a distinguished state, q_0), and then letting it go.

Our convention for *stopping the machine* is: The machine will *stop* (or “*halt*”, as we prefer to say) *iff* at some instance it is not specified how to proceed, given the current symbol/state pair. At that time (when the machine has halted), whatever is on the tape (that is, the largest string of symbols that starts and ends with some non blank symbol), is the result or output of the TM computation for the given input.

A question might now naturally arise in the reader’s mind: How will the “TM operator” (a human) ever be sure that she/he has *seen* “the largest string on tape that starts and ends with some non blank symbol”, when the tape is infinite? Is she or he doomed to search the tape forever and never be sure of the output?

This “problem” is not real. It arises here (now that I asked) due to the *informality* of our discussion above, which included talk about an “infinitely long tape medium”. Mathematically—as we will shortly see—the “tape” is just a *string-type variable* and the TM, which is just a program, manipulates the contents of its single variable, allowed to do so *one symbol per step*.

All the “operator” has to do is to “print” the contents of the variable once (and if) the computation of the “machine” halted!

1.2. Definitions

Now the formalities:

1.2.1 Definition. (TM—static description) A Turing Machine (TM), M , is a 4-tuple $M = (\Gamma, Q, q_0, I)$, where $\Gamma = \{s_0, s_1, \dots, s_n\}$ is a finite set of *tape symbols* called the *tape-alphabet*, and $s_0 = B$ (the *distinguished blank* symbol).

$Q = \{q_0, q_1, \dots, q_r\}$ is a finite set of states of which q_0 is *distinguished*. It is the *start-state*. States are really program labels.

I is the program. Its *instructions* are of three types:

- $q : abq'S$, meaning that, when performing instruction labelled q , the program modifies the currently scanned symbol a into b (these are just *names*; the symbol might have stayed the same!). The symbol in the *same position of the string* (“tape”) will be considered next—will become “current”—and the next instruction is the one labelled q' . The labels q and q' may well be the same!
- $q : abq'L$, meaning that, when performing instruction labelled q , the program modifies the currently scanned symbol a into b . The symbol *imme-*

diately to the left of b will be considered next, and the next instruction is the one labelled q' . If the contents of the string variable—the “tape” as we say—immediately before performing this instruction was au , where u is a string, then it will be Bbu after the instruction was executed. A blank symbol just appeared so that we never overshoot the string at the *left* end.

⚠ This makes mathematically precise the earlier barbarism according to which we have infinitely many B s all over the “tape” at all times. We don’t.

- $q : abq'R$, meaning that, when performing instruction labelled q , the program modifies the currently scanned symbol a into b . The symbol *immediately to the right* of b will be considered next, and the next instruction is the one labelled q' . If the contents of the string variable immediately before performing this instruction was ua , where u is a string, then it will be ubB after the instruction. A blank symbol just appeared so that we never overshoot the string at the *right* end either.

In using a TM one sometimes chooses a *subset*, Σ , of Γ that never includes the blank symbol. Σ is the alphabet used to form input strings; the *input alphabet*.

The very first instruction that will execute must be labelled q_0 . *By definition*, the execution of the program halts while at label q iff a is the current symbol and the program has no instruction that starts with this pair $q : a$.

If the pair $q : a$ always uniquely determines the instruction, then the TM is *deterministic*; otherwise it is *nondeterministic*.

We will only deal with deterministic TMs in this Note. □

⚠ It is clear now that a TM is a program that processes a single string variable. It is also clear that we can say “a TM is a finite set of quintuples $q : abq'T$ ”, where $T \in \{S, L, R\}$ as described above. The “active” alphabets Γ and Q —that is, the set of the corresponding symbols that are *referenced* by instructions—can be recovered from the set of quintuples by collecting all the “ a, b ” and all the “ q, q' ”, respectively, that occur in the instructions. It is clear that “set” rather than “ordered sequence” is fine since the TM programs are goto-driven; as long as we remember to start computing at label q_0 .

So how do we compute with a TM? By following the snapshots or *instantaneous descriptions* (ID) of its computation. A snapshot at any given computation step is determined if we know the string variable contents (“tape” contents), the position of the current symbol, and the instruction label of the instruction the program is about to perform. This snapshot we can capture by a string $tqau$, where $a \in \Gamma$, t and u are strings over Γ and q is the current state, while a is the current symbol.

⚠ There is always an a -part since the TM never overshoots the string it is manipulating!

Thus, a *computation* is a *finite* sequence of IDs, $\alpha_0, \dots, \alpha_n$.

1. α_0 is the *initial* ID, of the form q_0x —where x is the input string; the initial contents of the string variable. *Note that this is a theoretical model so we do not have a read instruction!*
2. α_n is a *terminal* or *halting* ID. That is, it has the form $tqau$ such that no instruction starts with “ $q : a$ ”. Thus tau is the *result* or *output* of the computation. *Note that this is a theoretical model so we do not have a write/print instruction!*
3. For all $i = 0, \dots, n - 1$, ID α_i *yields*, as we say, ID α_{i+1} —in symbols $\alpha_i \vdash \alpha_{i+1}$ —as we explained in 1.2.1.

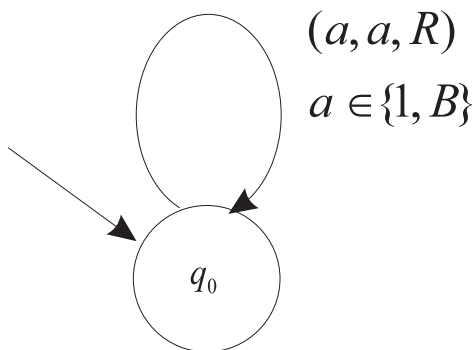


A “computation” is *by definition* halting or terminating. But what if the TM is in an “infinite loop” as in the example below? Well, if we have an infinite sequence of IDs $\alpha_0, \alpha_1, \dots$ that satisfies 1 above and also $\alpha_i \vdash \alpha_{i+1}$ for all i , then we say that we have an *infinite* or *non terminating* computation. Lack of a qualifier *always defaults to a terminating computation*.



Here are some examples.

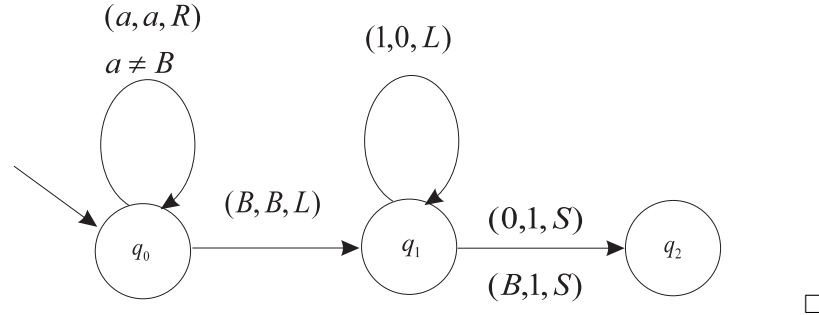
1.2.2 Example. The following is a TM over $\Gamma = \{1, B\}$ that never halts, regardless of what string in $\{1\}^+$ is presented as input.



The above is a “flowchart” or a “flow diagram” that depicts the two-quintuple TM

$$\{q_0 : aaq_0R\}, \text{ for } a \in \{1, B\} \quad \square$$

1.2.3 Example. The following is a TM over $\Gamma = \{0, 1, B\}$ that computes $x + 1$ if the number x is presented in *binary notation*.



In Computability one studies exclusively *number theoretic functions* and relations.

“Number theoretic” *means* that both the inputs and the outputs are members of the set of natural numbers, $\mathbb{N} = \{0, 1, 2, 3, \dots\}$.[†]

This choice of “what to attempt computing” presents no loss of generality, because all finite objects with which we may want to compute—e.g., negative integers, rational numbers, strings, matrices, graphs, etc.—can be coded by natural numbers (indeed, by “binary strings”[‡] which in turn we may think of as natural number representations, “base-2”).

Having fixed the “game” to be a theory of number theoretic (computable) functions and relations, the next issue is:

What is a convenient input/output convention for Turing machines, when these are used to “do Computability”?

The custom is to adopt the following input/output (“I/O”) conventions for TMs (see [5, 13, 14]):

1.2.4 Definition. (I/O) A number x is presented as input in *unary notation*, that is, it is represented as a string of length $x + 1$ over $\{1\}$.

Note the length which allows to code 0 as “1”!

More generally, a “vector input” x_1, x_2, \dots, x_n , often abbreviated as \vec{x}_n —or simply \vec{x} if the length n is unimportant, or understood—is represented as

$$1^{x_1+1}01^{x_2+1}0 \dots 01^{x_n+1}$$

where, as always, for a string v and a positive integer i

$$v^i \stackrel{\text{def}}{=} \underbrace{vv \dots v}_{i \text{ copies}}$$

[†]A relation returns “yes” or “no”, or equivalently, “true” or “false”. Traditionally, we code “yes” by 0 and “no” by 1—this is opposite to the convention of the C-language!—thus making relations a special case of (number theoretic) functions.

[‡]That is, strings over $\{0, 1\}$. Such strings we may also call *bit strings*.

If, on a given input, the TM halts, and β is its (unique) terminal ID, then the *integer-valued output* is *the total number of occurrences of the symbol “1” in the string β* .

Since in case of termination, the initial ID q_0t uniquely determines β , we write $Res(q_0t)$ —or $Res_M(q_0t)$ if we must be clear which TM M we are talking about—for the count of 1s in the terminal ID of the computation that starts with q_0t . \square



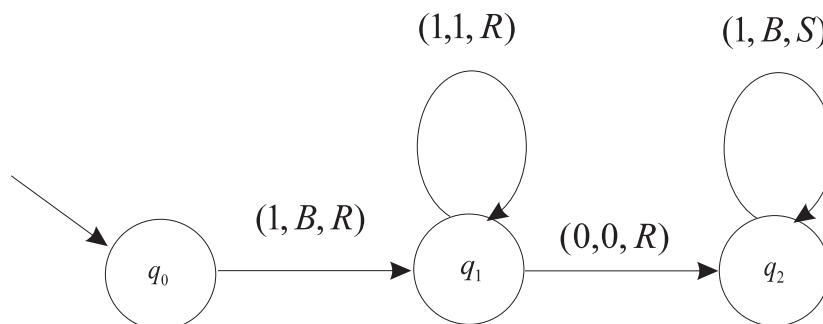
We have no guarantee that the output $Res_M(q_0t)$ is *defined* for all inputs t since some inputs may cause a *nonterminating* computation. E.g., no q_01^{x+1} leads to a computation in the machine of Example 1.2.2.



1.2.5 Example. The following TM computes the function.

input: x, y

output: $x + y$



We indulged above in a bit of “obscure programming” to avoid a 4th state: The last “loop” is not a loop at all. Once a 1 is replaced by a B and the head does not move, q_2 has no moves hence the machine halts.

It is easy to check that the computations (for any choices of values for x, y) here are

$$q_01^{x+1}01^{y+1} \vdash \dots \vdash B1^x0q_2B1^y$$

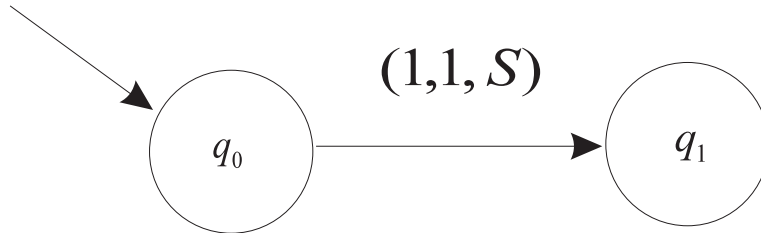
It is customary to write “ \vdash^* ” rather than “ $\vdash \dots \vdash$ ”. Thus the number theoretic output is as claimed: $x + y$ \square

1.2.6 Example. The following TM computes the function.

input: x

output: $x + 1$

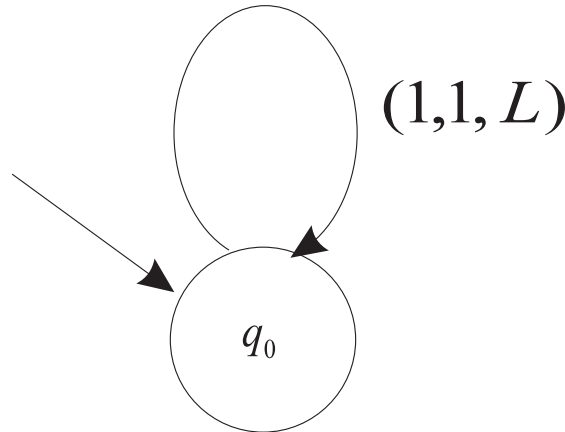
This time we do it in unary, as our “Computability conventions” dictate.



Here

$$q_0 1^{x+1} \vdash q_1 1^{x+1}$$

Another solution is



Here

$$q_0 1^{x+1} \vdash q_0 B 1^{x+1}$$

□

⚠ **1.2.7 Example.** A TM does *not* determine its number of arguments (1-vector, 2-vector, etc.), unlike “practical” programming languages where “read” statements and/or procedure parameters leave no doubt as to *what is the correct number of inputs*.

► *There is no “correct” or predetermined number of inputs for a TM.*

As long as we stick to the conventions of Definition 1.2.4, we can supply vectors of any length whatsoever, as input, to any given TM. ◀

For example, since the last TM above also has the computation

$$q_0 1^{x+1} 0 1^{y+1} 0 1^{x+1} \vdash^* q_0 B 1^{x+1} 0 1^{y+1} 0 1^{x+1}$$

it also computes the function

input: x, y, z

output: $x + y + z + 3$

As another example, looking back to Example 1.2.2 we have, for all x

$$Res(q_0 1^{x+1}) \uparrow$$

that is, that TM computes the (unary, or one-argument) *empty function*, denoted by \emptyset just like the empty set.



Incidentally, “ $\dots \uparrow$ ” means “ \dots is *undefined*” while “ $\dots \downarrow$ ” means “ \dots is *defined*”



Thus, the TM of Example 1.2.2 satisfies:

input: x

output: UNDEFINED

However, give it two (or more) inputs, and it computes something altogether different!

For example, it computes

input: x, y

output: $x + y + 2$

since

$$q_0 1^{x+1} 0 1^{y+1} \vdash^* 1^{x+1} q_0 0 1^{y+1}$$

□



We witnessed some “neat” things above. There is one thing that (purposely) stuck out throughout, to motivate the following definition. That was the cumbersome designation of functions by writing

input: bla-bla

output: bla-bla

1.2.8 Definition. (λ notation) There is a compact notation due to Church, called λ *notation*, that denotes a function given as

input: x_1, x_2, \dots, x_n

output: E

where “ E ” is an expression, or a “rule” on how to obtain a value.

We write simply “ $\lambda x_1 x_2 \dots x_n. E$ ”.

Thus “ λ –“.” is a “**begin**”–“**end**” block that delimits the arguments, and immediately after the “.” follows the “output rule, or expression”. □



1.2.9 Example. Thus, the first function discussed in Example 1.2.7 is

$$\lambda xyz. x + y + z + 3$$

the 2nd is

$$\lambda x. \uparrow$$

NOTE. “ \uparrow ” is *not* a value or number so, strictly speaking, writing $\lambda x. \uparrow$ is *abuse of notation*, treating “ \uparrow ” as an “undefined number”! We will (reluctantly) allow such (ab)uses of \uparrow .

Since we gave a name to the empty function (or *totally undefined function*) in 1.2.7, we may write

$$\emptyset = \lambda x. \uparrow$$

Careful! Do *not* write

$$\emptyset(x) = \lambda x. \uparrow$$

The left hand side is (an undefined) *value*, the right hand side is a *function*. The types of these two objects don’t match; “value” vs. “table”. They cannot possibly be equal!

You *may* write

$$\emptyset(x) = \uparrow$$

but, *better* still,

$$\emptyset(x) \uparrow$$

The final function in 1.2.7 is

$$\lambda xy. x + y + 2$$



1.2.10 Definition. (Partial functions) A *partial (number theoretic) function* f is one that *perhaps* is not defined on all values of the input(s). Thus, **all** functions that our theory studies are *partial*.

A function is *total* iff it is defined for *all possible values* of the input(s).

In the opposite case we say we have a *nontotal* function. □



Thus, if we put all total and all nontotal functions together, we obtain all the partial functions.

We see that “partial” is just a wishy-washy term (unlike the terms “total”/“nontotal”), that does *not* in itself tell us whether a function is *total* or *nontotal*.

It just says, “Caution! This function *may*, for some inputs, be undefined”.

At long last! ◇

1.2.11 Definition. (Computable (partial) function) A function $\lambda \vec{x}_n. f(\vec{x}_n)$ is a “*Turing computable partial function*”—but (following the literature) we rather say ***partial (Turing) computable function***—iff there is a TM M such that

$$\text{For all } a_i (i = 1, \dots, n) \text{ in } \mathbb{N}, \quad f(\vec{a}_n) \simeq \text{Res}_M(q_0 1^{a_1+1} 0 1^{a_2+1} 0 \dots 0 1^{a_n+1})$$



where the symbol “ \simeq ” here extends “ $=$ ”. It means that either *both* sides are *undefined* (i.e., “equal” each to “ \uparrow ”) or *both are defined* and equal in the *standard sense of equality of numbers*. ◇

A partial computable function is also called *partial recursive*. □



Why being so fancy? We said: “A function $\lambda \vec{x}_n. f(\vec{x}_n)$ ”.

Well, I cannot say “A function $f(\vec{x}_n)$ ”, because this object is **not** a function—*not* a *table* of input output pairs—it is rather a *number* (which I do not happen to know) or is undefined.

The alternatives

“A function f of arguments \vec{x}_n ”

or

“A function f with

input: \vec{x}_n

output: $f(\vec{x}_n)$ ”

are *correct*, but rather ugly. □



1.2.12 Definition. (\mathcal{P} and \mathcal{R}) The set of all *partial computable* (*partial recursive*) functions is denoted by the (calligraphic) letter \mathcal{P} .

The set of all *total computable* (*total recursive*) functions is denoted by the (calligraphic) letter \mathcal{R} .

Indeed, people say just *recursive* (or *computable*) function, and they *mean* total (computable). [**We have to get used to this!**] □

1.2.13 Theorem. $\mathcal{R} \subset \mathcal{P}$.

Proof. That $\mathcal{R} \subseteq \mathcal{P}$ is a direct consequence of definition 1.2.12. That the subset relation is *proper* follows from examples 1.2.2 and 1.2.7: The function \emptyset is in \mathcal{P} , but it is not in \mathcal{R} . □

1.3. A leap of faith: Church’s Thesis

The aim of Computability is to formalise (for example, via Turing Machines) the *informal* notions of “algorithm” and “computable function” (or “computable relation”).

We will not do any more programming with TMs.

A lot of models of computation, that were very different in their syntactic details and semantics, have been proposed in the 1930s by many people (Post, Church, Kleene, Markov, Turing). They were all *proved to compute exactly the same number theoretic functions*—those in the set \mathcal{P} . The various models, and the gory details of why they all do the same job precisely, can be found in [14].

This prompted Church to state his *belief*, famously known as “*Church’s Thesis*”, that

Every *informal* algorithm (pseudo-program) that we propose for the computation of a function can be implemented (*formalised*, in other words) in each of the known models of computation. In particular, it can be “programmed” as a Turing machine.

⚠ We note that at the present state of our understanding the concept of “algorithm” or “algorithmic process”, *there is no known way* to define an “intuitively computable” function—via a pseudo-program of sorts—*which is outside of \mathcal{P}* .

Thus, as far as we know, \mathcal{P} appears to be the *largest*—i.e., most inclusive—set of “intuitively computable” functions known.

This “empirical” evidence supports Church's Thesis. ⚠

Church's Thesis is not a theorem. It can never be, as it “connects” precise concepts (TM, \mathcal{P}) with imprecise ones (“algorithm”, “computable function”).

It is simply a belief that has overwhelming empirical backing, and should be only read as an *encouragement to present algorithms in “pseudo-code”*—*that is, informally*. Thus, Church's Thesis (indirectly) suggests that we concentrate in the essence of things, that is, perform only the high-level design of algorithms, and leave the actual “coding” to TM-programmers.[†]

Since we are interested in the essence of things in this note, and also promised to make it user-friendly, we will heavily rely on Church's Thesis here—which will refer to for short as “CT”—to “validate” our “high-level programs”.

In the literature, Rogers ([13], a very advanced book) heavily relies on CT. On the other hand, [3, 14] never use CT, and give all the necessary constructions (implementations) in their full gory details—*that is the price to pay, if you avoid CT*.

⚠ Here is the template of our use of CT: We present an algorithm in pseudo-code.

▶BTW, “pseudo-code” does not mean “sloppy-code”!◀

We then say: By CT, there is a TM that implements our algorithm. ⚠

It turns out, as we will observe in the next section, that the development of Computability theory benefits from an “arithmetisation” of Turing machines.

⚠ By the term *arithmetisation* we simply understand an *algorithmic process* by which we can

- (a) assign a natural number to a Turing machine, and conversely,
- (b) recover, from any given number, a *unique* Turing machine that the number represents or “names”—in other words, assign a TM to a number. ⚠

To this end,

- First, we will argue that we can algorithmically list all Turing machines. To this end, let us have a finite alphabet that can generate all possible “tape”-symbols (for Γ —see 1.2.1) and all possible state symbols (for Q —see 1.2.1):

$$\{s, q, '\}$$

Thus, we have an infinite supply of tape symbols

$$\{s, s', s'', s''', s''', \dots\}$$

[†]If ever in doubt about the legitimacy of a piece of “high-level pseudo-code”, then you ought to try to implement it in detail, as a TM, or, at least, as a “real” C-program!

that we denote—metatheoretically—by writing s_0 for s and s_n for s^n primes. We will always have $s_0 = B$, $s_1 = 0$ and $s_2 = 1$. We also have an infinite supply of instruction labels (states)

$$\{q, q', q'', q''', q'''' , \dots\}$$

that we denote, metatheoretically, as q_0, q_1, q_2, q_3, q_4 , etc. All our TMs will have their start state actually called “ q ” (that is, q_0).

We can now code any TM, over the alphabet

$$\{s, q, ', \$, \#, L, R, S\} \quad (1)$$

starting with the TM's instructions. An instruction $q_i : s_j s_k q_m T$, where $T \in \{S, L, R\}$, is coded as

$$\#q_i\#s_j\#s_k\#q_m\#T\# \quad (2)$$

A TM is coded by gluing into a string all its instructions, in any order, using $\$$ as inter-instruction glue.

Thus, a TM of n instructions can be coded according to the above scheme in $n!$ ways.

- Next, let us note that given a string over the alphabet (1) we can “parse it” to see if it is a TM or not: We look for
 - (a) It is a string that is a single string of type (2), or several such interglued by the symbol $\$$.
 - (b) q_0 occurs as the first symbol, after the leading $\#$, *in at least one participating instruction* (2).
 - (c) No two distinct participating instructions have identical starts: $\#q_i\#s_j\#$
- It is now easy to enumerate algorithmically all TMs: Since, as coded, they are strings over the finite alphabet (1), enumerate all TMs into a “List2” *simultaneously* with enumerating *all* strings over (1) in a list “List1”. We do the latter by enumerating by string length, and within the set of all strings of the same length we enumerate lexicographically, taking the order of symbols as displayed in (1) as fixed. Now, every time we put a string in List1 *we parse it* to see if it is a TM (code). We place it in List2 iff it is.
- By virtue of the previous bullet, the algorithm that lists the TMs also assigns one or more *position-numbers* to each, namely, the machine's position in List2. *Note that since we can have several ways to glue instructions together for a given TM, it will appear in several places.*



Thus we have obtained an algorithmic enumeration of all TMs as

$$M_0, M_1, M_2, \dots \quad (3)$$



It is immediate that given a TM N we can find it in the enumeration (3), and we can compute an i such that M_i is N : Indeed, as a first step, code N as described above. Now generate List2 as described, looking for N (as coded). There is a *first* (and only) time at which we will match N with a TM, M_i , that was listed. But this M_i is N .

We found N ’s location in the list: It is i .[†]

Now recall the comment that TMs do not determine the number of inputs of the functions they compute (1.2.7). Thus, for every $n > 0$, we can define

Important: $\phi_i^{(n)}$ is the function of n inputs computed by M_i (4)

that is, for all \vec{x}_n , we have $\phi_i^{(n)}(\vec{x}_n) \simeq Res_{M_i}(q_01^{x_1}01^{x_2}0 \dots 01^{x_n})$.



We write ϕ_i rather than $\phi_i^{(1)}$ in the one-argument case. The ϕ -notation is due to Rogers [13].



We say that $\phi_i^{(n)}$ is computed by “TM i ” rather than “TM M_i ” since if we know i then we know M_i and vice versa.

1.3.1 Theorem. A function f of $n > 0$ arguments is in \mathcal{P} iff, for some $i \in \mathbb{N}$, it is $\phi_i^{(n)} = f$.

Proof. The *if* is just (4) above. For the *only if* we are given that f is computable, say, by a TM N . Let i be such that $N = M_i$ in the manner we showed above (actually we can compute this i). Then $f = \phi_i^{(n)}$ by (4) above. \square

1.4. Unsolvable “Problems” The Halting Problem

A number-theoretic *relation* is some *set of n -tuples* from \mathbb{N} . A relation’s outputs are **t** or **f** (or “yes” and “no”). However, a number-theoretic relation *must* have values (“outputs”) also in \mathbb{N} .



Thus we re-code **t** and **f** as 0 and 1 respectively. This convention is preferred by Recursion Theorists (as people who do research in Computability like to call themselves) and is the opposite of the re-coding that, say, the C language employs.



We often write

$$R(\vec{a}_n)$$

as “short” for

$$\langle a_1, \dots, a_n \rangle \in R$$

[†]Since N can be coded in $n!$ different ways if it has n quintuples, it is clear the while N appears $n!$ times in the List2, every given coding of it appears just once.

Relations with $n = 2$ are called binary, and rather than, say,

$$< (a, b)$$

we write, in “infix”,

$$a < b \tag{1}$$

1.4.1 Definition. (Computable or Decidable relations) “A relation $Q(\vec{x}_n)$ is *computable*, or *decidable*” means that the function

$$c_Q = \lambda \vec{x}_n. \begin{cases} 0 & \text{if } Q(\vec{x}_n) \\ 1 & \text{otherwise} \end{cases}$$

is in \mathcal{R} .


The collection (set) of *all* computable relations we denote by \mathcal{R}_* . Computable relations are also called *recursive*.

By the way, the function $\lambda \vec{x}_n. c_Q(\vec{x}_n)$ we call the *characteristic function* of the relation Q (“c” for “characteristic”). \square



Thus, “a relation $Q(\vec{x}_n)$ is computable or decidable” means that some TM computes c_Q . But that means that some TM behaves as follows:

On input \vec{x}_n , it halts and outputs 0 iff \vec{x}_n satisfies Q (i.e., iff $Q(\vec{x}_n)$), it halts and outputs 1 iff \vec{x}_n does **not** satisfy Q (i.e., iff $\neg Q(\vec{x}_n)$).

We say that the relation has a *decider*, i.e., the TM that *decides* membership of *any* tuple \vec{x}_n in the relation. 

1.4.2 Definition. (Problems) A “*Problem*” is a formula of the type “ $\vec{x}_n \in Q$ ” or, equivalently, “ $Q(\vec{x}_n)$ ”.

Thus, *by definition*, a “problem” is a *membership question*. \square

1.4.3 Definition. (Unsolvable Problems) A problem “ $\vec{x}_n \in Q$ ” is called any of the following:

Undecidable

Recursively unsolvable

or just

Unsolvable

iff $Q \notin \mathcal{R}_*$ —in words, iff Q is **not** a computable relation. \square

Here is the most famous undecidable problem:

$$\phi_x(x) \downarrow \tag{1}$$

A different formulation uses the set

$$H = \{x : \phi_x(x) \downarrow\}^\dagger \tag{2}$$

[†]Both [13, 14] use K instead of H , but this notation is by no means standard. Thus, I felt free to use “ H ” here for *Halting*.

that is, *the set of all numbers x , such that machine M_x on input x has a (halting!) computation.*

H we shall call the “**halting set**”, and (1) we shall the “**halting problem**”. Clearly, (1) is equivalent to

$$x \in H$$

1.4.4 Theorem. *The halting problem is unsolvable.*

Proof. We show, **by contradiction**, that $H \notin \mathcal{R}_*$.

Thus we start by assuming the opposite.

$$\text{Let } H \in \mathcal{R}_* \tag{3}$$

that is, we can *decide membership* in H via a TM:

$$c_H \in \mathcal{R} \tag{4}$$

Define the function d below:

$$d(x) = \begin{cases} \phi_x(x) + 1 & \text{if } \phi_x(x) \downarrow \\ 0 & \text{if } \phi_x(x) \uparrow \end{cases} \tag{5}$$

Here is why it is computable:

Given x , do:

- Use the decider of H to test in which condition we are in (5); top or bottom.
- If we are in the top condition, then we fetch M_x and call it on input x . We add 1 to its output and halt everything. ***Because the top condition is true, the call will terminate!***
- If the bottom condition holds, then print 0 and exit.

By CT, the 3-bullet program has a TM realisation, so d is computable. Say,

$$d = \phi_i \tag{6}$$

What can we say about $\phi_i(i)$? Well, we have two cases:

Case 1. $\phi_i(i) \downarrow$. Then we are in the top case of (5). Thus, $d(i) = \phi_i(i) + 1$. But we also have $d(i) = \phi_i(i)$ by (6). This yields $\phi_i(i) + 1 = \phi_i(i)$. Since $\phi_i(i)$ is a number due to the case we are in, we got $1 = 0$; a contradiction.

Case 2. $\phi_i(i) \uparrow$. This leads to a contradiction too, since $d(i) = 0$ in this case, but is *also* $d(i) \uparrow$ by virtue of $d(i) = \phi_i(i)$.

Thus, (4) (and hence (3)) is false. We are done. \square

In terms of *theoretical significance*, the above is the most significant unsolvable problem.

Its import lies in the fact that we can use it to discover more unsolvable problems, some of which have great application interest. Example: The “program correctness problem” (see below).

But how does “ $x \in H$ ” help? Through the following technique of *reduction*:



Let P be a new *problem* (relation!) for which we want to see whether $\vec{y} \in P$ can be solved by a TM. We build a *reduction* that goes like this:

(1) Suppose that we have a TM M that decides $\vec{y} \in P$, for any \vec{y} . (2) Then we show how to use M as a subroutine to also solve $x \in H$, for any x . (3) Since the latter is unsolvable, no such TM M exists!



The *equivalence problem* is

Given two programs M and N can we test to see whether they compute the same function?



Of course, “testing” for such a question *cannot be done by experiment*: We cannot just run M and N for all inputs to see if they get the same output, because, for one thing, “all inputs” are infinitely many, and, for another, there may be inputs that cause one or the other program to run forever (infinite loop).



By the way, the equivalence problem is the general case of the “*program correctness*” problem which asks

Given a program P and a *program specification* S , does the program fit the specification for all inputs?

since we can view a specification as just another formalism to express a function-computation. By CT, all such formalisms, programs or specifications, boil down to TMs, and hence the above asks whether two given TMs compute the same function—program equivalence.

Let us show now that the program equivalence problem cannot be solved by any TM.

1.4.5 Theorem. (Equivalence problem) *The equivalence problem of TMs is the problem “given i and j ; is $\phi_i = \phi_j$?”[†]*

This problem is undecidable.

Proof. The proof is by a reduction (see above), hence by contradiction. We will show that if we have a TM that solves it, “yes”/“no”, then we have a TM that solves the halting problem too!

So assume we have an algorithm (TM) M for the equivalence problem (1)

[†]If we set $P = \{\langle i, j \rangle : \phi_i = \phi_j\}$, then this problem is the question “ $\langle i, j \rangle \in P$?” or “ $P(\langle i, j \rangle)$?”.

Let us use it to answer the question “ $a \in H$ ”—that is, “ $\phi_a(a) \downarrow$ ”, for any a .

So, fix an a (2)

Consider these two functions, for all x :

$$Z(x) = 0$$

and

$$\tilde{Z}(x) \simeq \begin{cases} 0 & \text{if } x = 0 \wedge \phi_a(a) \downarrow \\ 0 & \text{if } x \neq 0 \end{cases}$$

Both functions are intuitively computable: For Z and input x just print 0 and stop. For \tilde{Z} and input x , just print 0 and stop provided $x \neq 0$. On the other hand, if $x = 0$ then fetch M_a —where a is that in (2) above—from list (3) on p.14, and run it on input a . If this ever halts just print 0 and halt; otherwise let it loop forever.

By CT, both have TM programs, N and \tilde{N} . We can *compute* i and j by going down the aforementioned list (3), such that $N = M_i$ and $\tilde{N} = M_j$. Thus $Z = \phi_i$ and $\tilde{Z} = \phi_j$.

On the assumption (1), we feed i and j to M . This machine will halt and answer “yes” (0) precisely when $\phi_i = \phi_j$; will halt and answer “no” (1) otherwise. But note that $\phi_i = \phi_j$ iff $\phi_a(a) \downarrow$. We have thus solved the halting problem! □

1.5. Gödel's Incompleteness Theorem

It is rather surprising that Unprovability and Uncomputability are intimately connected. Gödel's original proof of his Incompleteness theorem did not use methods of Computability—indeed Computability theory was not yet developed. He used instead a variant of the *liar's paradox*,[†] namely, he devised within Peano arithmetic a formula D with no free variables, which said: “I am not a theorem.” He then proceeded to prove (essentially) that this formula is true, but has no syntactic proof within Peano arithmetic—it is not a theorem!

Gödel's Incompleteness theorem speaks to the inability of formal mathematical theories, such as Peano arithmetic and set theory, to totally capture the concept of truth. This does not contradict Gödel's own Completeness theorem that says “if $\models A$, then $\vdash A$ ”.

You see, Completeness talks about *absolute truth*,

that is, $\models_{\mathfrak{D}} A$, for *all* interpretations \mathfrak{D}

while Incompleteness speaks about *truth relative to the “standard” model only*. For Peano arithmetic, the standard model, $\mathfrak{N} = (\mathbb{N}, M)$ is the one that assigns to the nonlogical symbols—via M —the *expected*, or “standard”, interpretations as in the table below

[†] “I am lying”. Is this true? Is it false?

Abstract (language) symbol	Concrete interpretation
0	0 (zero)
S	$\lambda x.x + 1$
+	$\lambda xy.x + y$
\times	$\lambda xy.x \times y$
<	$\lambda xy.x < y$

Before we turn to a Computability-based proof of Gödel's Incompleteness, here, in outline, is how he did it: Suppose D at the top of this section is provable (a theorem) in Peano arithmetic. Then, since the rules of inference preserve truth and the axioms are true in \mathfrak{N} , we have that D is true in this interpretation. But note what it says! "I am not a theorem". This makes it also false (since we assumed it *is* a theorem!)

So, it is not a theorem after all. This automatically makes it true, for this is precisely what it says!

His proof was quite complicated, in particular in exhibiting a *formula* D that says what it says.

Here is a "modern" proof of Incompleteness, via a simple reduction proof within Computability:

1.5.1 Theorem. (Gödel's First Incompleteness Theorem) [†] *There is a true but not (syntactically) provable formula of Peano arithmetic.*

Proof. This all hinges on the fact that the set of theorems of Peano arithmetic can be algorithmically listed—by a *TM*.

Indeed, the alphabet of Peano arithmetic is finite

$$\perp, \top, p, x, ', (,), =, \neg, \wedge, \vee, \rightarrow, \equiv, \forall, 0, S, +, \times, <$$

where p and $'$ are used to build the infinite supply of Boolean variables

$$p, p', p'', \dots$$

and x and $'$ are used to build the infinite supply of object variables

$$x, x', x'', \dots$$

But then we can add a new symbol $\#$ to the alphabet to form

$$\perp, \top, p, x, ', (,), =, \neg, \wedge, \vee, \rightarrow, \equiv, \forall, 0, S, +, \times, <, \# \quad (1)$$

We use $\#$ to make a single string out of a proof

$$F_1, \dots, F_n$$

[†]The Second Incompleteness Theorem of Gödel shows that another true but *unprovable formula of arithmetic* is rather startling and significant: It says that "Peano arithmetic is free of contradiction—that is, it cannot prove \perp ". In plain English: Arithmetic cannot prove its own freedom of contradiction; such a proof must come from the "outside".

The 2nd Incompleteness Theorem is much harder to prove, and actually Gödel never gave a complete proof. The first complete proof was published in [6]; the second, different complete proof, was published in [15].

namely,

$$\#F_1\#F_2\#\dots\#F_n\#$$

Here's our listing algorithm:

Form *three* lists of strings over the alphabet (1).

- The first list, *List1*, contains all strings over (1), generated by size, and within each size-group generated lexicographically.
- The second, *List2*, is a list of *all proofs*—coded as above into single strings: Add a string to *List2* every time that we place a string in *List1* and find that it *is* a proof: We can check algorithmically for proof status, since we can recognise the axioms, and also can recognise when MP was used.
- Every time we place a proof in *List2*, we place its *last* formula in *List3*.

By CT, we have a TM enumerator, *E*, for *List3*, i.e., a machine that will have no input but will keep generating all of Peano arithmetic's theorems (with repetitions, to be sure, since every theorem appears in many proofs; how "many"?)

Let now an *a* be given, and let us show that I can solve " $\phi_a(a) \downarrow$?" provided Gödel's theorem is *false*, and therefore

$$\text{Every true formula of Peano arithmetic has a proof.} \quad (2)$$

We take on faith (cf. [16, 15]) that $\phi_{\tilde{a}}(\tilde{a}) \downarrow$ and $\phi_{\tilde{a}}(\tilde{a}) \uparrow$ are expressible as formulae of arithmetic—where you will recall from class that \tilde{a} is the number *a* written in the language of Peano arithmetic, (1), as

$$\overbrace{SS \dots S}^{a \text{ S's}} 0$$

OK, here it goes:

- Fetch machine M_a from the list (3) on p.14.
- Simultaneously run two machines: M_a on input *a* and also the enumerator *E* that lists *all* theorems of Peano arithmetic (*List3*).
- For M_a , keep an eye for whether it halts on input *a*; if so, halt everything and proclaim $a \in H$.
- For *E*, keep an eye for whether it ever prints *the formula* " $\phi_{\tilde{a}}(\tilde{a}) \uparrow$ "; if so, halt everything and proclaim $a \notin H$.

We solved the halting problem!

Hold on: Let me explain. What the assumption of *falsehood* of Gödel's theorem—(2) above—gives us is a means to verify $\phi_a(a) \uparrow$:

1. If $\phi_a(a) \uparrow$ is true, then by (2), $\phi_{\tilde{a}}(\tilde{a}) \uparrow$ *is a theorem*, thus it *appears* in the enumeration that *E* cranks out.

On the other hand,

2. If $\phi_a(a) \downarrow$ is true, then M_a will verify so for us, *by halting*.

So we will have computed the answer to $a \in H$ either way, having solved the halting problem, which is impossible!

Hence (2) is *false*!



Wait a minute! What if *both* things happen? That is, M_a halts, and $\phi_{\tilde{a}}(\tilde{a}) \uparrow$ shows up in the enumeration of theorems?

This would be disastrous because, depending on what happens *first*, we may end up with the wrong answer.

But it *cannot* happen, for if $\phi_a(a) \downarrow$ is true, then $\phi_a(a) \uparrow$ is *false*, hence its formal version, $\phi_{\tilde{a}}(\tilde{a}) \uparrow$, *cannot* appear as a theorem (all theorems are true in \mathfrak{N}).



□

Bibliography

- [1] Alonzo Church. A note on the Entscheidungsproblem. *J. Symbolic Logic*, 1:40–41, 101–102, 1936.
- [2] Alonzo Church. An unsolvable problem of elementary number theory. *Amer. Journal of Math.*, 58:345–363, 1936. (Also in Davis [4, 89–107]).
- [3] M. Davis. *Computability and Unsolvability*. McGraw-Hill, New York, 1958.
- [4] M. Davis. *The Undecidable*. Raven Press, Hewlett, NY, 1965.
- [5] Martin Davis. *Computability and Unsolvability*. McGraw-Hill, New York, 1958.
- [6] D. Hilbert and P. Bernays. *Grundlagen der Mathematik I and II*. Springer-Verlag, New York, 1968.
- [7] L. Kalmár. An argument against the plausibility of Church’s thesis. In *Constructivity in Mathematics*, pages 72–80. *Proc. of the Colloquium*, Amsterdam, 1957.
- [8] S.C. Kleene. Recursive predicates and quantifiers. *Transactions of the Amer. Math. Soc.*, 53:41–73, 1943. (Also in Davis [4, 255–287]).
- [9] A. A. Markov. Theory of algorithms. *Transl. Amer. Math. Soc.*, 2(15), 1960.
- [10] Rózsa Péter. *Recursive Functions*. Academic Press, New York, 1967.
- [11] Emil L. Post. Finite combinatory processes. *J. Symbolic Logic*, 1:103–105, 1936.
- [12] Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bull. Amer. Math. Soc.*, 50:284–316, 1944.
- [13] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.
- [14] G. Tourlakis. *Computability*. Reston Publishing, Reston, VA, 1984.

- [15] G. Tourlakis. *Lectures in Logic and Set Theory, Volume 1: Mathematical Logic*. Cambridge University Press, Cambridge, 2003.
- [16] G. Tourlakis. *Mathematical Logic*. John Wiley & Sons, Hoboken, NJ, 2008.
- [17] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math Soc.*, 2(42, 43):230–265, 544–546, 1936, 1937. (Also in Davis [4, 115–154].).