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 [Pos36, Pos44], Kleene [Kle43], Church [Chu36b], Turing [Tur37], Markov [Mar60]) 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 [Pé67] and Kalmár [Kal57], have argued that it is conceivable that the intuitive concept of

Intro to (un)Computability and Unprovability via URMs and Church's Thesis © by **George Tourlakis, 2011 and 2019**

By the way, Church proved ([Chu36a, Chu36b]) 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. A leap of faith: Church's Thesis

The aim of Computability is to *mathematically capture* (for example, via URMs) the *informal* notions of "algorithm" and "computable function" (or "computable relation").

As announced in class on Jan. 23, we will not do any more programming with URMs in class (one or two simple cases may appear in the midterm and final).

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, Turing) and more recently by Shepherdson and Sturgis ([SS63]). 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 [Tou84].

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 (*made mathematically precise*, in other words) in each of the known models of computation. In particular, **it can be "programmed" as a URM**.

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 *formalise* 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 mathematical objects (URM, $\mathcal{P}$) with imprecise *informal* ones ("algorithm", "computable function").

---

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

[†]In the so-called relativised computability (with partial oracles) Church's Thesis fails [Tou86].

Intro to (un)Computability and Unprovability via URMs and Church's Thesis © by **George Tourlakis, 2011 and 2019**

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" details to URM-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 —to which will refer , for short, as "CT"—to "validate" our "high-level programs".

In the literature, Rogers ([Rog67], a very advanced book) heavily relies on CT. On the other hand, [Dav58, Tou84, Tou12] 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 **how** to use CT:

- We **completely** present —that is, no essential detail is missing— an algorithm in *pseudo-code*.
  ▶BTW, "pseudo-code" does not mean "sloppy-code"!◀
- We then say: By CT, there is a URM that implements our algorithm. Hence the function that our pseudo code computes is in $\mathcal{P}$.

1.2. The Universal and S-m-n Theorems

The following is a useful tool in the development of computability theory. It is Kleene's "*universal function theorem*".

1.2.1 Theorem. (Universal function theorem) *There is a* partial computable *two-variable function h with this property: For any one-variable function $f \in \mathcal{P}$, there is a number $i \in \mathbb{N}$ such that $h(i,x) = f(x)$ for all x. Equivalently, $\lambda x.h(i,x) = f$.*

Recall (Notes on diagonalisation) that "=" for partial function **calls**, $f(\vec{x})$ and $g(\vec{y})$, means the usual —equality of numbers— if both side are *defined*. $f(\vec{x}) = g(\vec{y})$ is also true if both sides are *undefined*. In symbols,

$$f(\vec{x}) = g(\vec{y}) \text{ iff } f(\vec{x}) \uparrow \wedge g(\vec{y}) \uparrow \vee (\exists z)\Big(f(\vec{x}) = z \wedge g(\vec{y}) = z\Big)$$

The "universality" of h lies in the fact that it (or the URM that computes it) acts like a "stored program" (i.e., general purpose or universal) "computer": To compute a function f we present both a "*program*" for f —**coded** as the number i— and the input *data* (the x) to h and then we let it crank along.

[†]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 URM, or, at least, as a "real" C-program or equivalent!

Proof. Each $\lambda x.f(x) \in \mathcal{P}$ is a $M^{\mathbf{x}}_{\mathbf{y}}$, by definition.

In the Notes (#3) about diagonalisation we proved that we can algorithmically enumerate **all** $\lambda x.f(x) \in \mathcal{P}$ by algorithmically enumerating all strings of the form $M^{\mathbf{x}}_{\mathbf{y}}$, where M runs over all URMs.

Thus every computable function f is some $M^{\mathbf{x}}_{\mathbf{y}}$ and thus occupies **at least one** position i in the listing.[†]

Therefore, for every $i \in \mathbb{N}$, the list —in location i— holds some function $f = M^{\mathbf{x}}_{\mathbf{y}}$.

Conversely, for any $M^{\mathbf{x}}_{\mathbf{y}}$ we can **find** it in the list in a finite number of computational steps like: "keep generating the list, comparing every item generated with $M^{\mathbf{x}}_{\mathbf{y}}$ until they match —and math they will, since NO $M^{\mathbf{x}}_{\mathbf{y}}$ is omitted from the list".

Here is how the universal h is computed

- Given input i and x.
- So, generate the listing of the $M^{\mathbf{x}}_{\mathbf{y}}$ long enough and stop as soon as the i-th entry was generated. Say, this entry is $M^{\mathbf{x}}_{\mathbf{y}}$.
- Now run program M with x inputed into the input program-variable $\mathbf{x}$. If and when M stops, then we return the value held in the program-variable $\mathbf{y}$ of M.

By CT, the three-bullet algorithm (pseudo-program) above can be implemented as a URM. So h is partial computable.

But is it *universal*? Well, let f, of one variable, be computable via URM N, as $N^{\mathbf{u}}_{\mathbf{v}}$. Locate $N^{\mathbf{u}}_{\mathbf{v}}$ in the algorithmic list of unary $\mathcal{P}$-functions. Say the location is i. Then $h(i,x) = f(x)$, for all x. □

We will next introduce a universally used notation due to Rogers ([Rog67]):

1.2.2 Definition. In all that follows in this, **and all** the following Notes that will be posted, ϕ_i will denote the i-ith unary function in *the algorithmic list of* ***all*** $M^{\mathbf{x}}_{\mathbf{y}}$. □

Equipped with the above definition we can rephrase the Universal Function Theorem 1.2.1 as

$$h(i,x) = \phi_i(x), \text{ for all } i \text{ and } x$$

or even (better)

$$\lambda x.f(x) \in \mathcal{P} \text{ iff, for some } i \in \mathbb{N}, \text{ we have } f = \phi_i$$

It is worth "parsing" this "iff" above:

[†]Why not exactly one? Because for every M we can add to the end, but before the **stop** instruction, one or more instructions $\mathbf{z} \leftarrow 1$ where $\mathbf{z}$ is *fresh* (new variable). Any one of the modified M, call it M', satisfies $M^{\mathbf{x}}_{\mathbf{y}} = M'^{\mathbf{x}}_{\mathbf{y}}$. Thus every function $f \in \mathcal{P}$ has infinitely many programs that compute it.

$\rightarrow$ direction: The hypothesis means $f = N^{\mathbf{u}}_{\mathbf{v}}$ for some N. If $N^{\mathbf{u}}_{\mathbf{v}}$ occupies location i in the list, then, by 1.2.2, $f = \phi_i$.

$\leftarrow$ direction: The hypothesis $f = \phi_i$ means that $f = N^{\mathbf{u}}_{\mathbf{v}}$, where $N^{\mathbf{u}}_{\mathbf{v}}$ occupies location i in the list. But, $f = N^{\mathbf{u}}_{\mathbf{v}}$ says that f is indeed computable; in $\mathcal{P}$.

Calling x the "program" for $\lambda y.\phi_x(y)$ is not exact, but **is eminently apt**: x is just a number, not a set of URM instructions; but this number is the *address* (location) of *a URM program for* $\lambda y.\phi_x(y)$. Given the address, we **can retrieve** the program from a list via a computational procedure, in a finite number of steps!

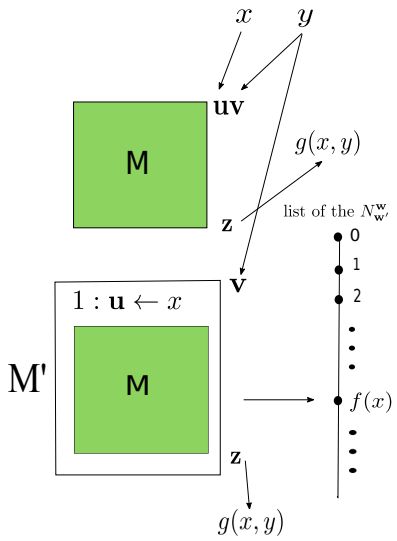
In the literature the address x in ϕ_x is called a ϕ-index. So, if $f = \phi_i$ then i is one of the infinitely many addresses where we can find how to program f.

Another fundamental theorem in computability is the *Parametrisation* or *Iteration* or also "*S-m-n*" theorem of Kleene. In fact, it and the universal function theorem along with a handful of initial computable functions are known to be *sufficient* to found computability axiomatically —but we will not get into this topic in this course.

1.2.3 Theorem. (Parametrisation theorem) *For every* $\lambda xy.g(x,y) \in \mathcal{P}$ *there is a function* $\lambda x.f(x) \in \mathcal{R}$ *such that*

$$g(x,y) = \phi_{f(x)}(y), \text{ for all } x, y \tag{1}$$

This says that given a program M that computes the function g as $M^{\mathbf{uv}}_{\mathbf{z}}$ with $\mathbf{u}$ receiving the input value x and $\mathbf{v}$ receiving the input value y —each via an "implicit" read statement— we can, for **any** fixed value x, *construct* a new program **located** in position $f(x)$ of *the algorithmic enumeration* of *all* $N^{\mathbf{w}}_{\mathbf{w}'}$ —the construction (of this address) effected by the *total* computable function f. The program at address $f(x)$ "knows" the value x, it is "hardwired" in its instructions, thus it does not receive the value x as a "read" *input*. This hardwiring is effected by **adding** to program M a new first instruction, namely, $1 : \mathbf{u} \leftarrow x$. The original first instruction of M is now the 2nd of the modified program. Indeed all instructions of M are pushed down (their addresses increase by 1).



The result is that the new program, at location $f(x)$ of the listing, and the original program for g —namely, $M^{\mathbf{uv}}_{\mathbf{z}}$— yield the same answer for the arbitrary fixed x, and all input values y "read" into the variable $\mathbf{v}$, as long as the the variable $\mathbf{u}$ —whether via a read statement (program M) or an "assignment" statement (program M') receives the same value x.

Proof. Of the S-m-n theorem. The proof is encapsulated by the preceding figure.

So, fix an input x for the variable $\mathbf{u}$ of program M. Then for any input y into $\mathbf{v}$ of M and the given x, M will output $g(x, y)$ (if $g(x, y)\downarrow$).

Now, the program M' depicted above is a trivial *algorithmic* modification of M —we can do this!— namely, we just added a new first instruction to M in order to "hardwire" the input value x in the resulting program M'. The new instruction is $1 : \mathbf{u} \leftarrow x$, and we have changed nothing else in M (except renumbering all its instructions $L : \ldots$ as $L + 1 : \ldots$[†]).

Thus, for the fixed x and the given I/O variables, trivially, both M and M' compute $g(x, y)$ for all y.

Since the program M' depends on the value x used in its first instruction, we call "$f(x)$" this program's address (location) in the algorithmic list of all $N^{\mathbf{w}}_{\mathbf{w}'}$.

All that remains to argue is that this address, $\lambda x.f(x)$ is **total computable**. Well,

[†]Of course, every L : **if x** $= 0$ **goto** P **else goto** R must change to $L + 1$: **if x** $= 0$ **goto** $P + 1$ **else goto** $R + 1$.

Intro to (un)Computability and Unprovability via URMs and Church's Thesis © by **George Tourlakis, 2011 and 2019**

- Given $M^{\mathbf{uv}}_{\mathbf{z}}$.
- Given x.
- build M' from M as outlined and indicated in the figure above.
- Go down the list of **all** $N^{\mathbf{w}}_{\mathbf{w}'}$ and keep comparing, until you find $M'^{\mathbf{v}}_{\mathbf{z}}$.
- Output the location, $f(x)$, of $M'^{\mathbf{v}}_{\mathbf{z}}$. You **WILL** find said location due to the underlined "all" above. So f is total.

By Church's thesis all informal computations here (building M' from M and the process for finding $f(x)$ for the given x) can be done by URMs. Thus, $f \in \mathcal{R}$. □

1.3. Unsolvable "Problems" The Halting Problem

Some of the comments below (and Definition 1.3.1) occurred already in earlier posted Notes. We revisit and introduce some additional terminology (e.g., "decidable").

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 (0 for **f** and non-zero for **t**).

1.3.1 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, we call the function $\lambda \vec{x}_n.c_Q(\vec{x}_n)$ —which does the re-coding of the outputs— the *characteristic function* of the relation Q ("c" for "characteristic"). □

Thus, "a relation $Q(\vec{x}_n)$ is computable or decidable" means that some URM computes c_Q. But that means that some URM 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 URM that *decides* membership of *any* tuple $\vec{x}_n$ in the relation.

1.3.2 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***. □

1.3.3 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. □

Here is the most famous undecidable problem:

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

A different formulation uses the set

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

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

K we shall call the "**halting set**", and (1) we shall the "**halting problem**".

Clearly, (1) is equivalent to

$$x \in K$$

1.3.4 Theorem. *The halting problem is unsolvable.*

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

Thus we start by assuming the opposite.

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

that is, we can *decide membership* in K via a URM, or, what is the same, we can *decide truth or falsehood* of $\phi_x(x)\downarrow$ for any x:

Consider then the infinite matrix below, each row of which denotes a function in $\mathcal{P}$ as an array of outputs, the outputs being numerical, or the special symbol "↑" for any undefined entry $\phi_x(y)$.

[†] All three [Rog67, Tou84, Tou12] use K for this set, but this notation is by no means standard. It is unfortunate that this notation clashes with that for the first projection K of a pairing function J. However the context will manage to fend for itself!

By 1.2.1 and the comments following it, **each** one argument function of $\mathcal{P}$ are in some row (as an array of outputs).

$$\begin{array}{cccccc}
\phi_0(0) & \phi_0(1) & \phi_0(2) & \dots & \phi_0(i) & \dots \\
\phi_1(0) & \phi_1(1) & \phi_1(2) & \dots & \phi_1(i) & \dots \\
\phi_2(0) & \phi_2(1) & \phi_2(2) & \dots & \phi_2(i) & \dots \\
\vdots & & & & & \\
\phi_i(0) & \phi_i(1) & \phi_i(2) & \dots & \phi_i(i) & \dots \\
\vdots & & & & &
\end{array}$$

We will show that under the assumption (3) that we hope to contradict the flipped diagonal —flipping all $\uparrow$ red entries to $\downarrow$ and vice versa; (3) says we can tell via a URM decider whether $\phi_x(x)\downarrow$ or not— represents a *partial recursive function* and hence **must** fit the matrix along some row i since we have all ϕ_i captured in the matrix.

On the other hand, flipping the diagonal is *diagonalising*, and thus the diagonal function constructed cannot fit. Contradiction! So, we blame (3) and thus have its negation proved: $K \notin \mathcal{R}_*$

In more detail, or as most texts present this, we have defined the flipped diagonal for all x as

$$d(x) = \begin{cases} \downarrow & \text{if } \phi_x(x)\uparrow \\ \uparrow & \text{if } \phi_x(x)\downarrow \end{cases}$$

Strictly speaking, the above does not **define** d since the "$\downarrow$" in the top case is not a value; it is ambiguous. Easy to fix:

Say,

$$d(x) = \begin{cases} 42 & \text{if } \phi_x(x)\uparrow \\ \uparrow & \text{if } \phi_x(x)\downarrow \end{cases} \tag{4}$$

Here is why the function in (4) is partial computable:

Given x, do:

- Use the decider for K (for $\phi_x(x)\downarrow$, that is) —assumed to exist by (3)— to test which condition obtains in (4); top or bottom.
- If the top condition is true, then we return 42 and stop.
- If the bottom condition holds, then transfer to an infinite loop:

while $1 = 1$ **do**
end

By CT, the 2-bullet program has a URM realisation, so d is computable.

Say now

$$d = \phi_i \tag{5}$$

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

Case 1. $\phi_i(i) \downarrow$. Then we are in the bottom case of (4). Thus $d(i) \uparrow$. But we also have $d(i) = \phi_i(i)$ by (5), and our case assumes $\phi_i(i) \downarrow$, that is, $d(i) \downarrow$; a contradiction.

Case 2. $\phi_i(i) \uparrow$. This leads to a contradiction too, since $d(i) = 42$ in this case, thus, $d(i) \downarrow$. But by (5) $d(i) = \phi_i(i)$, so we must also have $d(i) \uparrow$; contradiction once more.

So we reject (3). □

In terms of *theoretical significance*, the above is the most significant unsolvable problem that enables the process of finding more! How?

As an Example we illustrate the "program correctness problem" (see below).

But how does "$x \in K$" 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 URM. We build a *reduction* that goes like this:

(1) Suppose that we have a URM 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* $x \in K$*, for any* x*.*

(3) Since the latter is unsolvable*, no such URM 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 URMs, and hence the above asks whether two given URMs compute the same function —program equivalence.

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

1.3.5 Theorem. (Equivalence problem) *The equivalence problem of URMs 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 URM that solves it, "yes"/"no", then we have a URM that solves the halting problem too!

So assume we have an algorithm (URM) E for the *equivalence problem*.

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

So, fix an a (2)

Consider these two computable functions given by:
For all x:

$$Z(x) = 0$$

and

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

Both functions are intuitively computable: For Z we already have shown a URM M that computes it (in class). For $\widetilde{Z}$ and input x compute as follows:

- Print 0 and stop if $x \neq 0$.
- On the other hand, if $x = 0$ then, using the universal function h start computing $h(a, a)$, which is the same as $\phi_a(a)$ (cf. 1.2.1). If this ever halts just print 0 and halt; otherwise let it loop forever.

By CT, $\widetilde{Z}$ is in $\mathcal{P}$, that is, it has a URM program, say $\widetilde{M}$.

We can *compute* the locations i and j of M and $\widetilde{M}$ respectively by going down the list of all $N^{\mathbf{w}}_{\mathbf{w}'}$. Thus $Z = \phi_i$ and $\widetilde{Z} = \phi_j$.

By the "blue" assumption, we proceed to feed i and j to E. 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! A contradiction to the existence of URM E. □

1.4. 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

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

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

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:

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

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

[‡] 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 from contradiction —that is, it **cannot prove *all* formulas**. In plain English: Arithmetic cannot prove its own freedom from 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 [HB68]; the second, different complete proof, was published in [Tou03].

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

Indeed, the alphabet of Peano arithmetic is finite

$$x, \prime, (,), =, \neg, \vee, \forall, 0, S, +, \times, <$$

where x and $\prime$ are used to build the infinite supply of object variables

$$x, x', x'', \ldots$$

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

$$x, \prime, (,), =, \neg, \vee, \forall, 0, S, +, \times, <, \# \tag{1}$$

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

$$F_1, \ldots, F_n$$

namely,

$$\#F_1\#F_2\#\ldots\#F_n\#$$

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 URM 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

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

We take on faith (cf. [Tou08, Tou03]) that $\phi_{\widetilde{a}}(\widetilde{a})\downarrow$ and $\phi_{\widetilde{a}}(\widetilde{a})\uparrow$ are expressible as formulae of arithmetic —where $\widetilde{a}$ is the number a written in the language of Peano arithmetic, (1), as

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

OK, here it goes:

- Start computing $h(a,a)$ —i.e., $\phi_a(a)$— where h is the universal function of 1.2.1.
- Simultaneously run also the enumerator E that lists *all* theorems of Peano arithmetic (List3).
- For h, keep an eye for whether it halts on input (a,a); if so, halt everything and proclaim $a \in K$.
- For E, keep an eye for whether it ever prints *the formula* "$\phi_{\widetilde{a}}(\widetilde{a})\uparrow$"; if so, halt everything and proclaim $a \notin K$.

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), $\phi_{\widetilde{a}}(\widetilde{a})\uparrow$ *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 $h(a,a)$ 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, $h(a,a)$ halts, *and* $\phi_{\widetilde{a}}(\widetilde{a})\uparrow$ 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, $\phi_{\widetilde{a}}(\widetilde{a})\uparrow$, *cannot* appear as a theorem (all theorems are true in $\mathfrak{N}$).

□

Intro to (un)Computability and Unprovability via URMs and Church's Thesis © by
George Tourlakis, 2011 and 2019

Bibliography

[Chu36a] Alonzo Church, *A note on the Entscheidungsproblem*, J. Symbolic Logic **1** (1936), 40–41, 101–102.

[Chu36b] ———, *An unsolvable problem of elementary number theory*, Amer. Journal of Math. **58** (1936), 345–363, (Also in Davis [Dav65, 89–107]).

[Dav58] M. Davis, *Computability and Unsolvability*, McGraw-Hill, New York, 1958.

[Dav65] M. Davis, *The undecidable*, Raven Press, Hewlett, NY, 1965.

[HB68] D. Hilbert and P. Bernays, *Grundlagen der Mathematik I and II*, Springer-Verlag, New York, 1968.

[Kal57] L. Kalmár, *An argument against the plausibility of Church's thesis*, Constructivity in Mathematics, *Proc. of the Colloquium*, Amsterdam, 1957, pp. 72–80.

[Kle43] S.C. Kleene, *Recursive predicates and quantifiers*, Transactions of the Amer. Math. Soc. **53** (1943), 41–73, (Also in Davis [Dav65, 255–287]).

[Mar60] A. A. Markov, *Theory of algorithms*, Transl. Amer. Math. Soc. **2** (1960), no. 15.

[Pó67] Rózsa Péter, *Recursive Functions*, Academic Press, New York, 1967.

[Pos36] Emil L. Post, *Finite combinatory processes*, J. Symbolic Logic **1** (1936), 103–105.

[Pos44] ———, *Recursively enumerable sets of positive integers and their decision problems*, Bull. Amer. Math. Soc. **50** (1944), 284–316.

[Rog67] H. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[SS63] J. C. Shepherdson and H. E. Sturgis, *Computability of recursive functions*, Journal of the ACM **10** (1963), 217–255.

[Tou84] G. Tourlakis, *Computability*, Reston Publishing, Reston, VA, 1984.

Intro to (un)Computability and Unprovability via URMs and Church's Thesis © by
George Tourlakis, 2011 and 2019

[Tou86] G. Tourlakis, *Some reflections on the foundations of ordinary recursion theory, and a new proposal*, Zeitschrift für math. Logik **32** (1986), no. 6, 503–515.

[Tou03] G. Tourlakis, *Lectures in Logic and Set Theory, Volume 1: Mathematical Logic*, Cambridge University Press, Cambridge, 2003.

[Tou08] ———, *Mathematical Logic*, John Wiley & Sons, Hoboken, NJ, 2008.

[Tou12] ———, *Theory of Computation*, John Wiley & Sons, Hoboken, NJ, 2012.

[Tur37] Alan M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proc. London Math Soc. **2** (1936, 1937), no. 42, 43, 230–265, 544–546, (Also in Davis [Dav65, 115–154].).

Intro to (un)Computability and Unprovability via URMs and Church's Thesis © by
George Tourlakis, 2011 and 2019