Lecture #12, Oct. 26

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

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.

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.

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.

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

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").

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.[†]

[†]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!

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 we will refer, for short, as "CT"—to "validate" our "high-level (pseudo) 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. An Enumeration of all one-argument functions of $\mathcal{P}$

Recall:

1.2.1 Definition. The alphabet we use to construct the URMs is the following.

Consider the listing order of its members below **FIXED**.

$$\Gamma = \{X, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \leftarrow, :, +, \dot{-}, \textbf{if}, \textbf{goto}, \textbf{else}, \textbf{stop}, ;\}$$

where we added ";" to the alphabet as a *SEPARATOR*

to use when we write URMs horizontally, as STRINGS over the alphabet Γ. □

We repeat below the construction of the effective list (computable listing) of all loop programs (and one-argument primitive recursive functions) —almost as is— but this time for URMs.

(A) It is immediately believable that we can write a program that checks if a string over the alphabet Γ for URMs

is a syntactically correct URM program or not.

BTW, the symbols X and 1 above on p.10 generate *all* the variables,

$$X1, X11, X111, X1111, \ldots$$

As in the case of Loop-Programs we do not ever write variables down as what they really are —"$X\underbrace{1\ldots1}_{k\ 1s}$"—

but we will continue using *metasymbols* like

$$\mathbf{x}, \mathbf{y}''', z'_{123}, u''''_5, X, Y, Z, A, B, X'', Y'''_{23}$$

etc., for variables!

(B) We can algorithmically build the list, $List_1$, of ALL strings over Γ:

List by length; and in each length group **lexicographically** (alphabetically).

(C) Simultaneously to building $List_1$ build $List_2$ as follows:

For every string β generated in $List_1$, copy it into $List_2$ iff β checks to be a URM (which we can test by (A)).

(D) Simultaneously to building $List_2$ build $List_3$:

For every URM M (program) copied in $List_2$

copy *all the finitely many* strings M_Y^X (for all choices of X and Y in M) alphabetically (think of the string M_Y^X as "$M; X; Y$") into $List_3$.

Thus *ALL unary $\mathcal{P}$-functions* are listed by their aliases, the M_Y^X programs.

Let us call this list by its standard name in the literature ("Roger's Notation, [Rog67]"):

EFFECTIVE List of ALL one-argument $\mathcal{P}$ FUNCTIONS

$$\phi_0, \phi_1, \phi_2, \ldots, \phi_i, \ldots \tag{1}$$

where $\phi_i = M_Y^X$ iff M_Y^X is found in location i of $List_3$.

BTW "*EFFECTIVE List*" means "*algorithmically built List*".

1.2.1. A Universal function for unary $\mathcal{P}$ functions

We now have *universal* or *enumerating* function $U^{(P)}$ for *all* the unary functions in $\mathcal{P}$.

That is the function of TWO arguments

$$U^{(P)} = \lambda ix.\phi_i(x) \tag{2}$$

What do I mean by "Universal" here? The analogous concept as for $\mathcal{PR}$:

1.2.2 Definition. $U^{(P)}$ of (2) is *universal* or *enumerating* for all the unary functions of $\mathcal{P}$ meaning it has two properties:

1. If $g \in \mathcal{P}$ is unary, then there is an i such that
$$g = \lambda x.U^{(P)}(i, x)$$
because g is in the list (1) on the previous page.

   and

2. Conversely, for every $i \in \mathbb{N}$, $\lambda x.U^{(P)}(i, x) \in \mathcal{P}$ because it is an M_Y^X. □

We immediately obtain the Universal or *Enumeration Theorem* for *all* the unary functions in $\mathcal{P}$.

1.2.3 Theorem. (Enumeration theorem) *The Universal function of two arguments $\lambda ix.U^{(P)}(i,x)$ defined in (2) above and in 1.2.2 is partial recursive.*

Proof. Here is an informal algorithm to compute $U^{(P)}(i,x)$ for any i,x:

1. Given i,x

2. Develop the list $List_3$ of the preceding construction (part (D) on p.12) until the i-th URM M_Y^X has been listed

3. Take this M_Y^X and compute with input x.

If M terminates, then Y clearly holds the value $U^{(P)}(i,x) = \phi_i(x)$ —since $M_Y^X = \phi_i$.

Invoking CT we now declare that $\lambda ix.U^{(P)}(i,x)$ is more than informally algorithmic:

It CAN be implemented on a URM, hence is in $\mathcal{P}$. □

Hey, so *we* can *write a compiler for the* ϕ_i IN the URM Language!

How come we cannot repeat the argument we made for the case of $\mathcal{PR}$ *to show that* $U^{(P)} \notin \mathcal{P}$ *??!!*

Well, recall that "=" for partial function **calls**, $f(\vec{x})$ and $g(\vec{y})$, means the usual —equality of numbers— *if both sides* are *defined*.

However, $f(\vec{x}) = g(\vec{y})$ is also true *if both sides are* undefined. In symbols,

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

Thus, while

$$1 + U^{(PR)}(i,i) = U^{(PR)}(i,i)$$

was a contradiction in the case of $\mathcal{PR}$ because BOTH sides were defined, on the other hand, in the case of $\mathcal{P}$, something like

$$1 + U^{(P)}(i,i) = U^{(P)}(i,i)$$

simply implies that both sides are *undefined*! It is OK to be equal!

It is NOT a contradiction!

Thus the compiler for the M_Y^X CAN be programmed in the URM Programming Language.

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.

▶ NEXT let us establish the fact that we can *enumerate algorithmically* —or *we can effectively list*— also the set of *ALL partial recursive functions of TWO variables*.

Just *repeat* the construction (A)–(D) on pp.11–12, but modify (D):

Here you do instead:

▶ (D') Simultaneously to building $List_2$ build $List_3'$:

For every URM M (program) copied in $List_2$ copy *all* the finitely many strings M_Z^{XY} (for all choices of X, Y and Z —keeping X, Y distinct— in M) alphabetically (think of the string M_Z^{XY} as "$M; X; Y; Z$") into $List_3'$ ◀.

The obtained effective list $List_3'$ of M_Z^{XY} is denoted by

$$\phi_0^{(2)}, \phi_1^{(2)}, \phi_2^{(2)}, \phi_3^{(2)}, \ldots, \phi_i^{(2)}, \ldots \tag{2}$$

where $\phi_i^{(2)} = M_Z^{XY}$ iff M_Z^{XY} is found in location i.

The superscript "(2)" of ϕ indicates that we are effectively listing 2-argument $\mathcal{P}$-functions, $\lambda xy.\phi_i^{(2)}(x, y)$

1.2.4 Theorem. (Parametrisation theorem) *There is a 2-argument function S^1_1 in $\mathcal{R}$ such that*

$$\phi^{(2)}_i(x,y) = \phi_{S^1_1(i,x)}(y), \text{ for all } i,x,y \qquad (1)$$

Proof. This says that given a program M that computes a function $\phi^{(2)}_i$ as $M^{\mathbf{uv}}_{\mathbf{z}}$ with **u** receiving the input value x and **v** receiving the input value y —each via an "implicit" **read** statement— we can, for **any** fixed value x, *effectively*[†] *construct* a **new** program **located** in position $S^1_1(i,x)$ of *the algorithmic enumeration* of *all* unary $\mathcal{P}$-functions —(1) on p.13— that has the value x "hardwired" and only "reads" the y-value; YET GIVES THE SAME ANSWER in **z**.

▶ The "*effectively construct*" above is the requirement that S^1_1 is **recursive**: Indeed this requirement says that we can *OBTAIN* the program for the rhs of (1).

Each value x for **u** is "*hardwired*" —as $\mathbf{u} \leftarrow x$— in the program for $\phi_{S^1_1(i,x)}$ rather than being inputted via an implicit "**read u**".

[†] Algorithmically.

The program $N_{\mathbf{z}}^{\mathbf{v}}$ for $\phi_{S_1^1(i,x)}$ —for a given i and each x— is *almost* the same as $M_{\mathbf{z}}^{\mathbf{uv}}$, namely, it is

$$N_{\mathbf{z}}^{\mathbf{v}} = \left(\overbrace{1 : \mathbf{u} \leftarrow x;}^{\text{replaces } \mathbf{read\ u}} \ M \right)_{\mathbf{z}}^{\mathbf{v}} \tag{3}$$

where *all* instruction labels of M (even *inside* **if**-statments) are shifted by adding 1 to them.

Trivially, for all i (that is, all $M_{\mathbf{z}}^{\mathbf{uv}}$) and all x, we have (2) of the theorem.

Remains to argue that we *can compute* $S_1^1(i,x)$, *for all* i, x.

- Given i, x
- Develop the list (2) of the $\phi_i^{(2)}$ on p.19 (this is "$List_3'$" of (D')) on p.19) until we can obtain its i-th member, $M_{\mathbf{z}}^{\mathbf{uv}}$
- Now, Build the URM $N_{\mathbf{z}}^{\mathbf{v}}$ from $M_{\mathbf{z}}^{\mathbf{uv}}$ as in (3) above.
- Now, Develop $List_3$ —essentially the list of the ϕ_j— built by (D), p.12 and go down while you keep comparing, until you find $N_{\mathbf{z}}^{\mathbf{v}}$.
- Output the location of $N_{\mathbf{z}}^{\mathbf{v}}$ that you found —this is $S_1^1(i,x)$.

  You **WILL** find said location since $List_3$ contains *ALL* unary ϕ_j (listed as URM programs M_Y^X).

- So S_1^1 is total.

By Church's thesis the above informal process can be done by URMs. Thus, $S_1^1 \in \mathcal{R}$. □

LECTURE #13 Nov. 2

$\lambda x.S_1^1(i,x)$ is strictly increasing. Indeed, $S_1^1(i,x)$ is the *location* of

$$\left(\overbrace{1 : \mathbf{u} \leftarrow x;}^{\text{replaces } \mathbf{read}\ \mathbf{u}}\ M \right)_{\mathbf{z}}^{\mathbf{v}} \qquad (\dagger)$$

in the enumeration of the (unary) ϕ_j (p.13) while $S_1^1(i,x')$ is the *location* of

$$\left(\overbrace{1 : \mathbf{u} \leftarrow x';}^{\text{replaces } \mathbf{read}\ \mathbf{u}}\ M \right)_{\mathbf{z}}^{\mathbf{v}} \qquad (\ddagger)$$

Now if $x < x'$ then x is earlier than x' in the *alphabetic ordering* of x and x' viewed as strings of *decimal digits*.

But then —*all other things being equal*— program $(\dagger)$ appears earlier than program $(\ddagger)$ in the lexicographic ordering of programs N_Y^X.

Thus location $S_1^1(i,x)$ is before location $S_1^1(i,x')$.

In short, $S_1^1(i,x) < S_1^1(i,x')$.

1.3. Unsolvable "Problems" The Halting Problem

The following definition is repeated "for the record".

1.3.1 Definition. (Computable or Decidable relations) "*A relation* $Q(\vec{x}_n)$ *is* ***computable****, or* ***decidable*** *or* ***solvable***" means that *it is Recursive*;

That is, 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* $\mathcal{R}_*$. □

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 —by definition— 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*

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

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

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

Clearly, (1) is equivalent to

$$x \in K$$

[†]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!

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.

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

(3) says that 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 "$\uparrow$" for any undefined entry $\phi_x(y)$.

By 1.2.3 and the comments following it, **each** one-argument function of $\mathcal{P}$ is in some row (as an array of outputs).

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

We will use the assumed (3) above AND the main diagonal (red) of the above matrix to *define a function that is a 1-argument function of $\mathcal{P}$ that is NOT a row above*.

This will contradict "*each*" above.

So define the function d of one argument by

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

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

Given x, do:

- Use the *decider* for K (for $\phi_x(x)\downarrow$, that is) —assumed to exist by (3)— to test which condition is true 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:

$$k : X \leftarrow 1$$
$$k+1 : \textbf{goto } 1$$

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

Say now

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

Substitute ϕ_i for d in (4) to get

$$\phi_i(x) = \begin{cases} 42 & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases} \tag{6}$$

As *(6) is correct for all x*, it is correct for $x = i$.

So we get

$$\phi_i(i) = \begin{cases} 42 & \text{if } \phi_i(i) \uparrow \\ \uparrow & \text{if } \phi_i(i) \downarrow \end{cases}$$

You see the contradiction?

Cases.

- $\phi_i(i) = lhs = 42$. Then *we are in the top case*. But that implies $\phi_i(i) \uparrow$ No good!!!
- Well maybe the other case works? $\phi_i(i) = lhs = \uparrow$. *We are in the bottom case*. But this implies $\phi_i(i) \downarrow$ No good!!!

We have a contradiction no matter which case we pick.

So we reject (3). Done! □

In terms of *theoretical significance*, the above is a *very 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* 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*?

How so? *Well, we can view a specification as just another formalism to FINITELY express a function computation*.

By CT, all such formalisms, programs or specifications, boil down to URMs, and hence, at the end of the day, 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. We will show that if we have a URM that solves the *equivalence problem*, "yes"/"no", then we have a URM that *solves the halting problem too*!

So assume (URM) E solves the *equivalence problem*.

Let us use E 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 actually constructed a URM M* that computes it (in class/notes/text).

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, *call* $U^{(P)}(a, a)$, which is the same as $\phi_a(a)$ (cf. 1.2.3).

  If this ever halts just print 0 and halt; *otherwise let it loop forever*.

By CT, $\widetilde{Z}$ *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 $N^{\mathbf{w}}_{\mathbf{w}'}$ ($List_3$, p.12).

Thus $Z = \phi_i$ *and* $\widetilde{Z} = \phi_j$.

Since we ASSUMED that E solves the equivalence problem, feed it i and j and let it crank.

By definition of a decider, E will terminate with answer one of:

- **0**. Then $Z(x) = \widetilde{Z}(x)$, for all x. *But lhs is always defined*, thus so is rhs. *We conclude* $\phi_a(a)\downarrow$.

- **1**. $Z(x) = \widetilde{Z}(x)$, is *NOT* true for all x. The only possibility for that is $\neg\Big(Z(0) = \widetilde{Z}(0)\Big)$ (for all other x-values, lhs=rhs).

  This can only be because rhs is undefined. *We conclude* $\phi_a(a)\uparrow$.

We just solved the halting problem using E as a subroutine! IMPOSSIBLE. So E does not exist. □

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 [**?**, 89–107]).

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

[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 [**?**, 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.

[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.

[Tou12] G. Tourlakis, *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 [**?**, 115–154].).