A user-friendly Introduction to (un)Computability and Unprovability via "Church's Thesis" Part II

This is Part II of our Uncomputability notes. We introduce "half-computable" relations $Q(\vec{x})$ here. These play a central role in Computability. The term "half-computable" describes them well: For each of these relations there is a URM M that will halt precisely for the inputs $\vec{a}$ that make the relation true: i.e., $\vec{a} \in Q$ or equivalently $Q(\vec{a})$ is true. For the inputs that make the relation false —$\vec{b} \notin Q$— M loops forever. That is, M *verifies* membership but does not yes/no-*decide* it by halting and "printing" the appropriate 0 (yes) or 1 (no).

Can't we tweak M into M' that is a decider of such a Q? No, not in general! For example, our halting set K has a verifier but no decider! (The latter we know: having a decider means $K \in \mathcal{R}_*$ and we know that this NOT the case.

Since the "yes" of a verifier M is signaled by halting but the "no" is signaled by looping forever, the definition below does not require the verifier to print 0 for "yes". Here "yes" equals "halting".

0.1. Semi-decidable relations (or sets)

0.1.1 Definition. (Semi-recursive or semi-decidable sets)

A relation $Q(\vec{x}_n)$ is *semi-decidable* or *semi-recursive* —what we called suggestively "half-computable" above— iff there is a URM, M, which on input $\vec{x}_n$ ***has a (halting!) computation iff*** $\vec{x}_n \in Q$. ***The output of*** M ***is unim-***

portant!

A less civilized, but more mathematically precise way to say the above is:

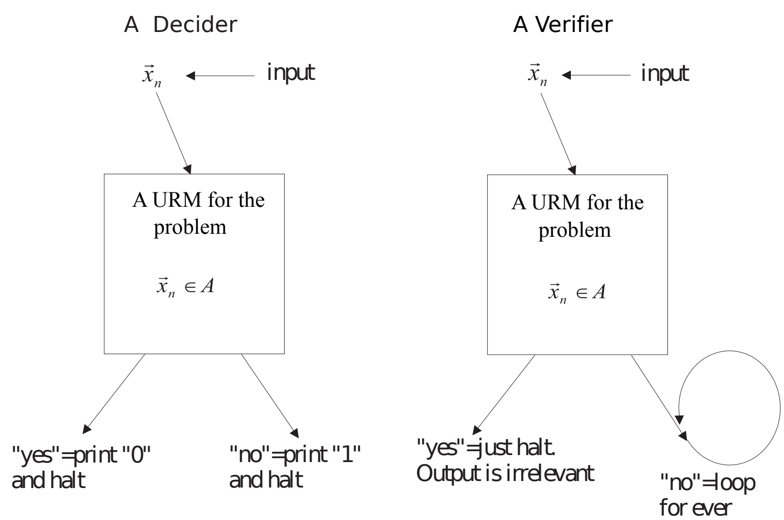
A relation $Q(\vec{x}_n)$ is *semi-decidable* or *semi-recursive* iff there is an $f \in \mathcal{P}$ such that

$$Q(\vec{x}_n) \equiv f(\vec{x}_n) \downarrow \tag{1}$$

Clearly, an $f \in \mathcal{P}$ is some $M_{\mathbf{y}}^{\vec{\mathbf{x}}_n}$. Thus, M is a verifier for Q.

The set of *all* semi-decidable relations we will denote by $\mathcal{P}_*$.[†] □

The following figure shows the two modes of handling a query, "$\vec{x}_n \in A$", by a URM.



Here is an important semi-decidable set.

0.1.2 Example. K is semi-decidable. To work within the formal definition (0.1.1) we note that the function $\lambda x.\phi_x(x)$ is in $\mathcal{P}$ via the universal function theorem of Part I: $\lambda x.\phi_x(x) = \lambda x.h(x, x)$ and we know $h \in \mathcal{P}$.

Thus $x \in K \equiv \phi_x(x) \downarrow$ settles it. By Definition 0.1.1 (statement labeled (1)) we are done. □

0.1.3 Example. Any recursive relation A is also semi-recursive.

That is,

$$\mathcal{R}_* \subseteq \mathcal{P}_*$$

[†]This is not a standard symbol in the literature. Most of the time the set of all semi-recursive relations has *no* symbolic name! We are using this symbol in analogy to $\mathcal{R}_*$—the latter being fairly "standard".

Indeed, intuitively, all we need to do to convert a decider for $\vec{x}_n \in A$ into a verifier is to "intercept" the "print 1"-step and convert it into an "infinite loop",

while(1)
  {
  }

By CT we can certainly do that via a URM implementation.

A more elegant way (which still invokes CT) is to say, OK: Since $A \in \mathcal{R}_*$, it follows that c_A, its characteristic function, is in $\mathcal{R}$.

Define a new function f as follows:

$$f(\vec{x}_n) = \begin{cases} 0 & \text{if } c_A(\vec{x}_n) = 0 \\ \uparrow & \text{if } c_A(\vec{x}_n) = 1 \end{cases}$$

This is intuitively computable (the "↑" is implemented by the same **while** as above).

Hence, by CT, $f \in \mathcal{P}$. But

$$\vec{x}_n \in A \equiv f(\vec{x}_n) \downarrow$$

because of the way f was defined. Definition 0.1.1 rests the case.

One more way to do this: Totally mathematical ("formal", as people say incorrectly[†]) this time!

OK,

$$f(\vec{x}_n) = \text{if } c_A(\vec{x}_n) = 0 \text{ then } 0 \text{ else } \emptyset(\vec{x}_n)$$

That is using the sw function that is in $\mathcal{PR}$ and hence in $\mathcal{P}$, as in

$$f(\vec{x}_n) = \text{if } \overset{\overset{c_A(\vec{x}_n)}{\downarrow}}{z} = 0 \text{ then } \overset{\overset{0}{\downarrow}}{u} \text{ else } \overset{\overset{\emptyset(\vec{x}_n)}{\downarrow}}{w}$$

$\emptyset$ is, of course, the empty function which by Grz-Ops can have any number of arguments we please! For example, we may take

$$\emptyset = \lambda \vec{x}_n.(\mu y) g(y, \vec{x}_n)$$

where $g = \lambda y \vec{x}_n.SZ(y) = \lambda y \vec{x}_n.1$.

In what follows we will prefer the informal way (proofs by Church's Thesis) of doing things, most of the time. □

[†] "Formal" refers to syntactic proofs based on axioms. Our "*mathematical*" proofs are mostly *semantic*, depend on meaning, not just syntax. That is how it is in the majority of MATH publications.

An important observation following from the above examples deserves theorem status:

0.1.4 Theorem. $\mathcal{R}_* \subset \mathcal{P}_*$

Proof. The $\subseteq$ part of "$\subset$" is Example 0.1.3 above.

The $\neq$ part is due to $K \in \mathcal{P}_*$ (0.1.2) and the fact that the halting problem is unsolvable ($K \notin \mathcal{R}_*$).

So, there are sets in $\mathcal{P}_*$ (e.g., K) that are not in $\mathcal{R}_*$. □

What about $\overline{K}$, that is, the *complement*

$$\overline{K} = \mathbb{N} - K = \{x : \phi_x(x) \uparrow\}$$

of K?

The following general result helps us handle this question.

0.1.5 Theorem. *A relation $Q(\vec{x}_n)$ is recursive if **both** $Q(\vec{x}_n)$ and $\neg Q(\vec{x}_n)$ are semi-recursive.*

Before we proceed with the proof, a remark on notation is in order.

In "set notation" we write the complement of a set, A, of n-tuples as $\overline{A}$. This means, of course, $\mathbb{N}^n - A$, where

$$\mathbb{N}^n = \underbrace{\mathbb{N} \times \cdots \times \mathbb{N}}_{n \text{ copies of } \mathbb{N}}$$

In "relational notation" we write the same thing (complement) as

$$\neg A(\vec{x}_n)$$

Similarly,

"set notation": $A \cup B$, $A \cap B$

"relational notation": $A(\vec{x}_n) \vee B(\vec{y}_m)$, $A(\vec{x}_n) \wedge B(\vec{y}_m)$

Back to the proof.

Proof. We want to prove that some URM, N, **decides**

$$\vec{x}_n \in Q$$

We take two verifiers, M for "$\vec{x}_n \in Q$" and M' for "$\vec{x}_n \in \overline{Q}$",[†] and run them —on input $\vec{x}_n$— as "co-routines" (i.e., we crank them simultaneously).

If M halts, then we stop everything and print "0" (i.e., "yes").

If M' halts, then we stop everything and print "1" (i.e., "no").

CT tells us that we can put the above —if we want to— into a single URM, N. □

[†] We can do that, i.e., M and M' exist, since both Q and $\overline{Q}$ are semi-recursive.

0.1.6 Remark. The above is really an "iff"-result, because $\mathcal{R}_*$ is *closed under complement* as we showed in an earlier Note.

Thus, if Q is in $\mathcal{R}_*$, then so is $\overline{Q}$, by closure under $\neg$. By Theorem 0.1.4, both of Q and $\overline{Q}$ are in $\mathcal{P}_*$. □

0.1.7 Example. $\overline{K} \notin \mathcal{P}_*$.

Now, **this** ($\overline{K}$) is a horrendously unsolvable problem! This problem is so hard it is not even ***semi***-decidable!

Why? Well, if instead it were $\overline{K} \in \mathcal{P}_*$, then combining this with Example 0.1.2 and Theorem 0.1.5 we would get $K \in \mathcal{R}_*$, which we know is not true. □

0.2. Unsolvability via Reducibility

We turn our attention now to a **methodology** towards discovering new undecidable problems, and also new non semi-recursive problems, beyond the ones we learnt about so far, which are just, $x \in K$, $\phi_i = \phi_j$ (equivalence problem) and $x \in \overline{K}$. In fact, we will learn shortly that $\phi_i = \phi_j$ is worse than undecidable; just like $\overline{K}$ it is not even semi-decidable.

The tool we will use for such discoveries is the concept of *reducibility* of one set to another:

0.2.1 Definition. (Strong reducibility) For any two subsets of $\mathbb{N}$, A and B, we write

$$A \leq_m B^{\dagger}$$

or more simply

$$A \leq B \tag{1}$$

pronounced *A is strongly reducible to B*, meaning that there is a (total) *recursive* function f such that

$$x \in A \equiv f(x) \in B \tag{2}$$

We say that "*the reduction is effected by f*". □

In words, $A \leq_m B$ says that we can *algorithmically* solve the problem $x \in A$ if we know how to solve $z \in B$. The algorithm is:

1. Given x.
2. Given the "subroutine" $z \in B$.
3. Compute $f(x)$.

[†] The subscript m stands for "many one", and refers to f. We do not require it to be 1-1, that is; *many* (inputs) to *one* (output) will be fine.

4. Give the same answer for $x \in A$ (true or false) as you did for $f(x) \in B$.

When (1) holds, then, intuitively, "A is easier than B to either decide or verify" since if we know how to decide or (only) verify membership in B then we can use this to decide or (only) verify membership in A. This observation has a very precise counterpart (Theorem 0.2.3 below). But first,

0.2.2 Lemma. *If $Q(y, \vec{x}) \in \mathcal{P}_*$ and $\lambda \vec{z}.f(\vec{z}) \in \mathcal{R}$, then $Q(f(\vec{z}), \vec{x}) \in \mathcal{P}_*$.*

Proof. By Definition 0.1.1 there is a $g \in \mathcal{P}$ such that

$$Q(y, \vec{x}) \equiv g(y, \vec{x}) \downarrow \tag{1}$$

Now, for any $\vec{z}$, $f(\vec{z})$ is some number which if we plug into y in (1) throughout we get an equivalence:

$$Q(f(\vec{z}), \vec{x}) \equiv g(f(\vec{z}), \vec{x}) \downarrow \tag{2}$$

But $\lambda \vec{z}\vec{x}.g(f(\vec{z}), \vec{x}) \in \mathcal{P}$ by Grz-Ops. Thus, (2) and Definition 0.1.1 yield $Q(f(\vec{z}), \vec{x}) \in \mathcal{P}_*$. □

0.2.3 Theorem. *If $A \leq B$ in the sense of 0.2.1, then*

(i) if $B \in \mathcal{R}_*$, then also $A \in \mathcal{R}_*$

(ii) if $B \in \mathcal{P}_*$, then also $A \in \mathcal{P}_*$

Proof.

Let $f \in \mathcal{R}$ effect the reduction.

(i) Let $z \in B$ be in $\mathcal{R}_*$.
Then for some $g \in \mathcal{R}$ we have

$$z \in B \equiv g(z) = 0$$

and thus

$$f(x) \in B \equiv g(f(x)) = 0 \tag{1}$$

But $\lambda x.g(f(x)) \in \mathcal{R}$ by composition, so (1) says that "$f(x) \in B$" is in $\mathcal{R}_*$. But that is the same as "$x \in A$".

(ii) Let $z \in B$ be in $\mathcal{P}_*$. By 0.2.2, so is $f(x) \in B$. But this says $x \in A$. □

Taking the "contrapositive", we have at once:

0.2.4 Corollary. *If $A \leq B$ in the sense of 0.2.1, then* (*i*) *if* $A \notin \mathcal{R}_*$*, then also* $B \notin \mathcal{R}_*$

(*ii*) *if* $A \notin \mathcal{P}_*$*, then also* $B \notin \mathcal{P}_*$

We can now use K and $\overline{K}$ as a "yardsticks" —or reference "problems"— and discover more undecidable and also *non semi-decidable* problems.

The idea of the corollary is applicable to the so-called "complete index sets".

0.2.5 Definition. (Complete Index Sets) Let $\mathcal{C} \subseteq \mathcal{P}$ and $A = \{x : \phi_x \in \mathcal{C}\}$. A is thus the set of **ALL** programs (known by their addresses) x that compute any *unary* $f \in \mathcal{C}$: Indeed, let $\lambda x.f(x) \in \mathcal{C}$. Thus $f = \phi_i$ for some i. Then $i \in A$. But this is true of **all** ϕ_m that equal f.

We call A a *complete* (all) *index* (programs) set. □

0.2.6 Example. The set $A = \{x : \mathrm{ran}(\phi_x) = \emptyset\}$ is not semi-recursive.

Recall that "range" for $\lambda x.f(x)$, denoted by $\mathrm{ran}(f)$, is defined by

$$\{x : (\exists y) f(y) = x\}$$

We will try to show that

$$\overline{K} \leq A \tag{1}$$

If we can do that much, then Corollary 0.2.4, part ii, will do the rest.
Well, define

$$\psi(x, y) = \begin{cases} 0 & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{if } \phi_x(x) \uparrow \end{cases} \tag{2}$$

Here is how to compute ψ:

Given x, y, ignore y. Fetch machine M at address x from the standard listing, and call it on input x. If it ever halts, then print "0" and halt everything. If it never halts, then you will never return from the call, which is the correct specified in (2) behaviour for $\psi(x, y)$.

By CT, ψ is in $\mathcal{P}$, so, by the S-m-n Theorem, there is a recursive h such that

$$\psi(x, y) = \phi_{h(x)}(y), \text{ for all } x, y$$

You may NOT use S-m-n UNTIL after you have proved that your "$\lambda xy.\psi(x, y)$" <u>is</u> in $\mathcal{P}$.

We can rewrite this as,

$$\phi_{h(x)}(y) = \begin{cases} 0 & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{if } \phi_x(x) \uparrow \end{cases} \tag{3}$$

or, rewriting (3) without arguments (as equality of functions, not equality of function calls)

$$\phi_{h(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x) \downarrow \\ \emptyset & \text{if } \phi_x(x) \uparrow \end{cases} \tag{3'}$$

In (3′), $\emptyset$ stands for $\lambda y. \uparrow$, the empty function.

Thus,

$$h(x) \in A \text{ iff } \operatorname{ran}(\phi_{h(x)}) = \emptyset \overbrace{\text{ iff }}^{\text{bottom case in } 3'} \phi_x(x) \uparrow$$

The above says $x \in \overline{K} \equiv h(x) \in A$, hence $\overline{K} \leq A$, and thus $A \notin \mathcal{P}_*$ by Corollary 0.2.4, part ii. □

0.2.7 Example. The set $B = \{x : \phi_x \text{ has finite domain}\}$ is not semi-recursive.

This is really easy (once we have done the previous example)! All we have to do is "talk about" our findings, above, differently!

We use the same ψ as in the previous example, as well as the same h as above, obtained by S-m-n.

Looking at (3′) above we see that the top case has infinite domain, while the bottom one has finite domain (indeed, empty). Thus,

$$h(x) \in B \text{ iff } \phi_{h(x)} \text{ has finite domain } \overbrace{\text{ iff }}^{\text{bottom case in } 3'} \phi_x(x) \uparrow$$

The above says $x \in \overline{K} \equiv h(x) \in B$, hence $B \notin \mathcal{P}_*$ by Corollary 0.2.4, part ii. □

0.2.8 Example. Let us mine twice more (3′) to obtain two more important undecidability results.

1. Show that $G = \{x : \phi_x \text{ is a constant function}\}$ is undecidable.

   We (re-)use (3′) of 0.2.6. Note that in (3′) the top case defines a constant function, but the bottom case defines a non-constant. Thus

   $$h(x) \in G \equiv \phi_x = \lambda y.0 \equiv x \in K$$

   Hence $K \leq G$, therefore $G \notin \mathcal{R}_*$.

2. Show that $I = \{x : \phi_x \in \mathcal{R}\}$ is undecidable. Again, we retell what we can read from (3′) in words that are relevant to the set I:

   $$h(x) \in I \overset{\emptyset \notin \mathcal{R}!}{\equiv} \phi_x = \lambda y.0 \equiv x \in K$$

   Thus $K \leq I$, therefore $I \notin \mathcal{R}_*$. □

In Notes #8 we will sharpen the result 2 of the previous example.

0.2.9 Example. (The Equivalence Problem, again) We now revisit the equivalence problem and show it is more unsolvable than we originally thought (cf. Notes #6): The relation $\phi_x = \phi_y$ is not semi-decidable.

By 0.2.2, if the 2-variable predicate above is in $\mathcal{P}_*$ then so is $\lambda x.\phi_x = \phi_y$, i.e., taking a constant for y. Choose then for y a ϕ-index for the *empty function*.

So, if $\lambda xy.\phi_x = \phi_y$ is in $\mathcal{P}_*$ then so is

$$\phi_x = \emptyset$$

which is equivalent to

$$\mathrm{ran}(\phi_x) = \emptyset$$

and thus not in $\mathcal{P}_*$ by 0.2.6. □

0.2.10 Example. The set $C = \{x : \mathrm{ran}(\phi_x) \text{ is finite}\}$ is not semi-decidable.

Here we cannot reuse $(3')$ above, because **both** cases —in the definition by cases— have functions of ***finite range***. We want one case to have a function of finite range, but the other to have i*nfinite range*.

Aha! This motivates us to choose a different "ψ" (hence a different "h"), and retrace the steps we took above.

OK, define

$$g(x, y) = \begin{cases} y & \text{if } \phi_x(x) \downarrow \\ \uparrow & \text{if } \phi_x(x) \uparrow \end{cases} \qquad (ii)$$

Here is an algorithm for g:

Given x, y.

Use the universal program M for unary partial computable functions (computes the $\lambda xy.h(x, y)$ of Notes #6) and start computing $h(x, x)$, that is, $\phi_x(x)$.

If this ever halts, then print "y" and halt everything. If it never halts then you will never return from the call, which is the correct behaviour for $g(x, y)$: namely, we want $g(x, y) \uparrow$ if $x \in \overline{K}$.

By CT, g is partial recursive, thus by S-m-n, for some recursive unary k we have

$$g(x, y) = \phi_{k(x)}(y), \text{ for all } x, y$$

Thus, by (ii)

$$\phi_{k(x)} = \begin{cases} \lambda y.y & \text{if } x \in K \\ \emptyset & \text{othw} \end{cases} \qquad (iii)$$

Hence,

$$k(x) \in C \text{ iff } \phi_{h(x)} \text{ has finite range} \quad \overbrace{\text{iff}}^{\text{bottom case in } iii} \quad x \in \overline{K}$$

That is, $\overline{K} \leq C$ and we are done. □

0.2.11 Exercise. Show that $D = \{x : \mathrm{ran}(\phi_x) \text{ is infinite}\}$ is undecidable. □

0.2.12 Exercise. Show that $F = \{x : \mathrm{dom}(\phi_x) \text{ is infinite}\}$ is undecidable. □

Enough "negativity"! Here is an important "positive result" that helps to prove that certain relations *are* semi-decidable:

0.2.13 Theorem. (Projection theorem) *A relation $Q(\vec{x}_n)$ is semi-recursive **iff** there is a recursive (*decidable*) relation $S(y, \vec{x}_n)$ such that*

$$Q(\vec{x}_n) \equiv (\exists y) S(y, \vec{x}_n) \tag{1}$$

Q is obtained by "projecting" S along the y-co-ordinate, hence the name of the theorem.

Proof. **If**-part. Let $S \in \mathcal{R}_*$, and Q be given by (1) of the theorem.

We show that some M semi-decides

$$\vec{x}_n \in Q \tag{2}$$

Here is how:

proc $Q(\vec{x}_n)$
  $y \leftarrow 0$ /* Initialize "search" */
  while $(c_S(y, \vec{x}_n) = 1)$ /* This call always terminates since $S \in \mathcal{R}_*$ */
    {
      $y \leftarrow y + 1$
    }

By CT, there is a URM N that implements the above pseudo-code. Clearly, this URM semi-decides (2).

Did I say "search"? But of course! Trivially,

$$(\exists y) S(y, \vec{x}_n) \equiv (\mu y) S(y, \vec{x}_n) \downarrow \tag{$*$}$$

But $\lambda \vec{x}_n . (\mu y) S(y, \vec{x}_n) \in \mathcal{P}$.[†] Hence $Q(\vec{x}_n)$ is semi-recursive by Definition 0.1.1 since, by $(*)$,

$$Q(\vec{x}_n) \equiv (\mu y) S(y, \vec{x}_n) \downarrow$$

Only if-part. This is more interesting because it introduces a new proof-technique:

So, we now know that $Q \in \mathcal{P}_*$, and want to show that *there is an $S \in \mathcal{R}_*$ for which (1) above holds*:

Well, let M semi-decide $\vec{x}_n \in Q$.

[†] You recall, of course, that $(\mu y) S(y, \vec{x}_n)$ is *defined* to *mean* $(\mu y) c_S(y, \vec{x}_n)$.

Define $S(y, \vec{x}_n)$ as follows:

$$S(y, \vec{x}_n) \stackrel{\text{by Def}}{\equiv} \begin{cases} \textbf{true} & \text{if } M \text{ on input } \vec{x}_n \text{ halts in } \textbf{exactly } y \text{ computation steps} \\ \textbf{false} & \text{otherwise} \end{cases}$$

We argue that $S(y, \vec{x}_n)$ is decidable. Indeed, here is how to decide it:

1. Enlist the help of *someone* who keeps track of computing **time** for M from the time the URM's (program's) computation starts and onwards.

   In intuitive (non mathematical) terms, this "someone" could be the Operating System under which the program M is compiled and executed.

2. Given an input $y, \vec{x}_n$, the *System* keeps track of **elapsed computation time** during M's computation. This "time" could be in *time units*, like *seconds, nanoseconds*, etc., **or** in *instruction-execution units*, that is, the *number of instructions executed* —*with* repetitions, of course: instruction, say, $L: \ldots$, if embedded in a loop, may be executed *several* times. Each counts!

   The system will halt the entire process (including exiting M even if M did not hit its stop instruction yet) as soon as y time units have elapsed.

   It is *absolutely important* to remember at this point that any URM M will continue computing *in a trivial manner* once it hits **stop**: This "trivial manner" is that M will go on "computing", specifically "executing" **stop** ad infinitum, and doing so by **changing nothing in any variable**. See Definition 0.1.1.2, case (iv), in Notes #2.

3. **Output Decisions at time y.**

   Output will be as follows:

   - **true** (0) if M was executing **stop**, but **not** doing so at step $y - 1$.

     Comment. The above is the case where M hit its **stop** instruction **exactly** in y steps.

   - **false** (1) if M was **not** executing **stop** at the time the System halted everything.

     Comment. The above is the case where M needed MORE than y steps to finish its computation (if at all).

   - **false** (1) if M was executing **stop**, **and** doing so at step $y-1$ as well.

     Comment. The above is the case where M hit its **stop** *before* y steps.

By CT, the above algorithm, M plus Operating System plus decisions on what to output, can be formalized into a URM, N, which **decides** (true/false) S, i.e., $S \in \mathcal{R}_*$.

Now it is trivial that (1) holds, for we have the equivalences

$$Q(\vec{x}_n) \equiv \text{For some } y,\ M, \text{ on input } \vec{x}_n, \text{ halts in exactly } y \text{ steps}$$

That is

$$Q(\vec{x}_n) \equiv \text{For some } y,\ S(y, \vec{x}) \text{ is true}$$

□

0.2.14 Example. The set $A = \{(x, y, z) : \phi_x(y) = z\}$ is semi-recursive.
Here is a verifier for the above predicate:

Given input x, y, z. **Comment**. Note that $\phi_x(y) = z$ is true iff two things happen: (1) $\phi_x(y)\downarrow$ **and** (2) the computed value is z.

1. Call the universal function h on input x, y.
2. If the Universal program H for h halts, then
   - If the output of H equals z then halt everything (the “yes” output).
   - If the output is not equal to z, then enter an infinite loop (say “no”, by looping).

By CT the above informal verifier can be formalised as a URM M.
But is it correct? Does it verify $\phi_x(y) = z$?
Yes. See **Comment** above. □