# A user-friendly Introduction to Computability via Turing Machines (and “Church’s Thesis”) Part II

We have stopped short our exposition—in Part I of these notes—due to a “publication deadline”.[†]

This is Part II. Section numbering resumes from where we stopped in Part I. In Section 6 we had hoped to include material on semi-decidable relations. We did not manage to include more than the definition and a remark. We reproduce both below—adding the notation “$\mathcal{P}_*$”—and take it from there.

## *7. Semi-decidable relations (or sets)*

**7.1 Definition. (Semi-recursive or semi-decidable sets)**

A relation $Q(\vec{x}_n)$ is *semi-decidable* or *semi-recursive* iff there is a TM, $M$, which on input $\vec{x}_n$ ***has a (halting!) computation iff*** $\vec{x}_n \in Q$. ***The output of*** $M$ ***is unimportant!***

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

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

Sipser [2] calls these relations ***recognizable***. Some others call them ***recursively enumerable***.[‡]

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

[†] I had set myself a deadline, the weekend of Nov. 11–12, to upload a substantial amount of notes.

[‡] To see why, visit Section 8.

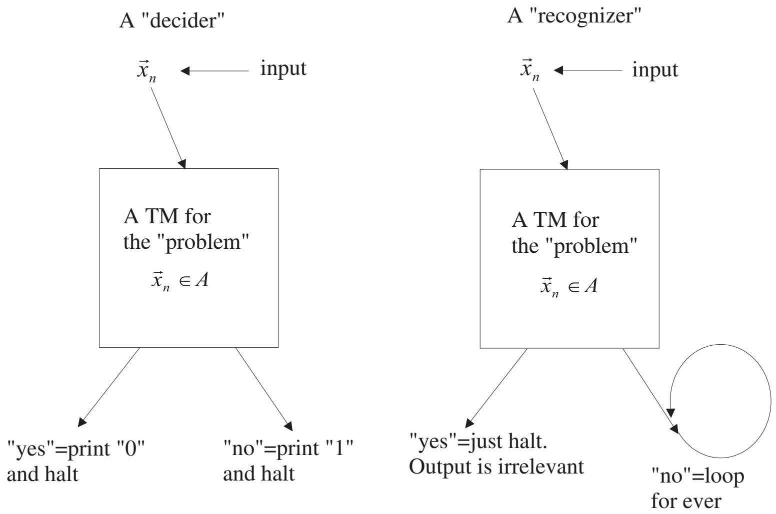
[§] This is not a standard symbol in the literature. Most of the time the set of all semi-recursive relations has *no* symbolic name! We are using this symbol in analogy to $\mathcal{R}_*$—the latter being fairly “standard”.

**7.2 Remark.** Yet another way to say (1) is:

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

$$Q(\vec{x}_n) \equiv \phi_e^{(n)}(\vec{x}_n) \downarrow \tag{2}$$

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




Here is an important semi-decidable set.

**7.3 Example.** $H$ is semi-decidable (recognizable, as our text ([2]) prefers to say). To work within the formal definition (7.1) we note that the function $\lambda x.\phi_x(x)$ is in $\mathcal{P}$.

Indeed, to compute it for any input value $x$ do:

(a) Go and fetch machine $M_x$

(b) Run $M_x$ on input $x$

(c) Output whatever output (if any!) is generated in step (b) above.

CT then validates our claim.

But

$$x \in H \text{ iff } \phi_x(x) \downarrow$$

By Definition 7.1 (statement labeled (1)) we are done.

Alternatively, we argue intuitively, using the figure for a recognizer, above: Our recognizer (for $x \in H$) does the following: Well, exactly (a)–(c) above!

If it halts, we take it to mean that $x \in H$ is true (and we are right too!). If it never halts (something we will never know), then this corresponds correctly to the "answer" $x \notin H$. $\square$

**7.4 Example.** Any recursive relation $A$ is also semi-recursive.

Indeed, intuitively, all we need to do to convert a decider for $\vec{x}_n \in A$ into a recognizer 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 TM 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 "$\uparrow$" 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 7.1 rests the case.

Clearly, the "elegant" way is just a wordy way to repeat the "informal" way. In what follows we will prefer the informal way of doing things, most of the time. $\square$

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

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

*Proof.* The $\subseteq$ part of "$\subset$" is Example 7.4 above.

The $\neq$ part is due to $H \in \mathcal{P}_*$ (7.3) and the fact that the halting problem is unsolvable ($H \notin \mathcal{R}_*$).

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

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

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

of $H$?

The following general result helps us handle this question.

**7.6 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": $A(\vec{x}_n) \vee B(\vec{x}_n)$, $A(\vec{x}_n) \wedge B(\vec{x}_n)$

Back to the proof.

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

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

We take two recognizers, $M$ 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 TM, $N$.
$\square$

**7.7 Remark.** The above is really an "iff"-result, because $\mathcal{R}_*$ is *closed under complement.*

If we believe the latter claim for a moment,[‡] we then see how the converse of Theorem 7.6 goes:

If $Q$ is in $\mathcal{R}_*$, then so is $\overline{Q}$, by closure. By Theorem 7.5, both of $Q$ and $\overline{Q}$ are in $\mathcal{P}_*$.

Now to see why $\mathcal{R}_*$ is closed under complement, let $Q$ be in $\mathcal{R}_*$. Let $M$ be a decider for $Q$. A decider $M'$ for $\overline{Q}$ is a trivial modification of $M$:

Simulate $M$ on input $\vec{x}_n$. If $M$ wants to print "0" and halt, print "1" instead and halt. If $M$ wants to print "1" and halt, print "0" instead and halt. "Mathematically", this is just the observation that

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

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

[‡] We have a closer look at closure properties in Section 9.

CT performs its obvious duty.

**7.8 Example.** $\overline{H} \notin \mathcal{P}_*$.

So, here is a horrendously unsolvable problem! This problem is so hard it is not even ***semi***-decidable!

Why? Well, if instead it were $\overline{H} \in \mathcal{P}_*$, then combining this with Example 7.3 and Theorem 7.6 we would get $H \in \mathcal{R}_*$, which we know is not true. □

Recall the concept of reducibility of one set to another:

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

$$A \leq_m B$$

or more simply

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

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

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

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

"**Small print**": $A \leq_m B$ is also pronounced $A$ *is many-one reducible to* $B$, hence the "$m$".† "Many-one" is a name due to the fact that (usually) many $x$ values (in $A$) are mapped into the same value $f(x)$ (in $B$)—see (2) above. If $f$ is a "1-1 function", that is, whenever $f(a) = f(b)$ then $a = b$, then we say that the reducibility is 1-1. This is so important that we have a symbol for it ([1]): $A \leq_1 B$.

We have seen in the previous section that, intuitively, "$B$ is more 'difficult' than $A$ (to decide)", in the special cases where $A = H$.

Indeed, when (1) holds, then "$B$ is more difficult than $A$ to either decide or recognize" in the following very precise sense:

**7.10 Theorem.** *If* $A \leq B$ *in the sense of 7.9, 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.*

† [2] attributes the "$m$" to the term "mapping-reducibility". We follow [1] in our choice of terminology.

$(i)$ Let $B \in \mathcal{R}_*$, and let $f \in \mathcal{R}$ effect the reduction.

Let $M$ be a *decider* for "$x \in B$".

Here is how to decide "$x \in A$":

(a) Compute $f(x)$ (for the given $x$). ***This computation terminates!***

(b) Use the result $f(x)$ as input to the decider $M$. That is, call $M$ with input $f(x)$.

If $M$ prints "0", then print "0" and halt. If $M$ prints "1", then print "1" and halt.

By CT we can build a single TM, $M'$, that carries out the steps (a)–(b).

Clearly, $M'$ behaves correctly: It halts with a "0" if $f(x) \in B$—that is, if $x \in A$ (by $A \leq B$—see 7.9). It halts with a "1" if $f(x) \notin B$—that is, if $x \in A$ is false (by $A \leq B$).

Thus, $A \in \mathcal{R}_*$

$(ii)$ Let $B \in \mathcal{P}_*$, and let $f \in \mathcal{R}$ effect the reduction.

Let $N$ be a *recognizer* for "$x \in B$".

Here is how to semi-decide "$x \in A$":

(1) Compute $f(x)$ (for the given $x$). ***This computation terminates!***

(2) Use the result $f(x)$ as input to the *recognizer* $N$. That is, call $N$ with input $f(x)$.

If $N$ ever halts, then halt.

By CT, there is a TM $N'$ that implements the algorithm (1)–(2).

This $N'$ behaves correctly: If $x \in A$, then $f(x) \in B$, thus $N'$ halts (the call to $N$ "returns").

If $x \notin A$, then $f(x) \notin B$, thus the call to $N$ (input $f(x)$) does ***not*** return. So $N'$ never returns either.

Thus, $A \in \mathcal{P}_*$

$\square$

Taking the "contrapositive", we have at once:

**7.11 Corollary.** *If $A \leq B$ in the sense of 7.9, 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 $\overline{H}$ as a "yardstick" and discover some more *unrecognizable* (= *NOT semi-recursive*) sets.

**7.12 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

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

As with the reducibility arguments of the previous section, we will assume that we can semi-decide

$$x \in A$$

and then try to use this to semi-decide

$$a \in \overline{H}$$

by finding a recursive $h$ such that

$$a \in \overline{H} \text{ iff } h(a) \in A$$

In short, we will try to show that

$$\overline{H} \le A \tag{1}$$

If we can do that much, then Corollary 7.11 will do the rest.
Well, define

$$f(a, x) = \begin{cases} 0 & \text{if } \phi_a(a) \downarrow \\ \uparrow & \text{if } \phi_a(a) \uparrow \end{cases}$$

Here is how to compute $f$:

Given $a, x$, ignore $x$. Fetch machine $M_a$ from the standard listing, and call it on input $a$. 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 behaviour for $f(a, x)$.

By CT, $f = \phi_e^{(2)}$ for some $e \in \mathbb{N}$. By the parametrization theorem, there is a recursive $h$ such that

$$f(a, x) = \phi_e^{(2)}(a, x) = \phi_{h(a)}(x), \text{ for all } a, x$$

Therefore,

$$\phi_{h(a)}(x) = \begin{cases} 0 & \text{for all } x \text{ if } \phi_a(a) \downarrow \\ \uparrow & \text{for all } x \text{ if } \phi_a(a) \uparrow \end{cases} \tag{2}$$

hence,

$$\phi_{h(a)} \text{ has empty range iff } \phi_a(a) \uparrow \ \text{(Otherwise the range is } \{0\})$$

Another way to say the above (bring in $A$!) is to say

$$a \in \overline{H} \text{ iff } h(a) \in A$$

which is (1) above! Done. $\square$

**7.13 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 $f$,

$$f(a,x) = \begin{cases} 0 & \text{if } \phi_a(a) \downarrow \\ \uparrow & \text{if } \phi_a(a) \uparrow \end{cases}$$

and the same $h$ as above.

Note that the following statement is the same as (2) of the previous example, but in different notation!

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

Recall that "$\emptyset$" denotes both the empty set and the empty function. In the bottom case above it is the empty function. Thus,

$$\phi_{h(a)} \text{ has finite domain, indeed } (\textit{empty}!) \text{ iff } \phi_a(a) \uparrow$$

Another way to say the above (remember $B$?) is to say

$$a \in \overline{H} \text{ iff } h(a) \in B$$

thus

$$\overline{H} \le B$$

and 7.11 does the rest. □

**7.14 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 lead to ***finite range***. We want one case to lead to finite range, the other to infinite.

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

We want, as usual, to show that

$$a \in \overline{H} \equiv k(a) \in C \tag{$i$}$$

for some recursive function $k$. OK, define

$$g(a,x) = \begin{cases} x & \text{if } \phi_a(a) \downarrow \\ \uparrow & \text{if } \phi_a(a) \uparrow \end{cases} \tag{$ii$}$$

Here is an algorithm for $g$:

Given $a, x$. Fetch machine $M_a$ from the standard listing, and call it on input $a$. If it ever halts, then print "$x$" and halt everything. If it never halts then you will never return from the call, which is the correct behaviour for $g(a,x)$.

By CT, $g$ is partial recursive, thus $g = \phi_e^{(2)}$ for some $e$. By parametrization, there is a recursive $k$ such that

$$g(a, x) = \phi_e^{(2)}(a, x) = \phi_{k(a)}(x), \text{ for all } a, x$$

Thus, by (ii)

$$\phi_{k(a)} = \begin{cases} \lambda x.x & \text{if } \phi_a(a) \downarrow \\ \emptyset & \text{if } \phi_a(a) \uparrow \end{cases}$$

or, more suggestively,

$$\phi_{k(a)} = \begin{cases} \text{has infinite range (all of } \mathbb{N}) & \text{if } \phi_a(a) \downarrow \\ \text{has finite range (empty)} & \text{if } \phi_a(a) \uparrow \end{cases}$$

In short, $(i)$ holds. By Corollary 7.11, "$x \in C$" cannot be recognizable! $\square$

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

**7.15 Theorem. (Projection theorem)** *A relation $Q(\vec{x}_n)$ is semi-recursive **iff** there is a recursive 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 TM $N$ that implements the above pseudo-code. Clearly, this TM semi-decides (2).

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

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

$$S(y, \vec{x}_n) \stackrel{\text{by Def}}{\equiv} \begin{cases} 0 & \text{if } M \text{ on input } \vec{x}_n \text{ halts in exactly } y \text{ steps} \\ 1 & \text{otherwise} \end{cases}$$

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

Given an input $\langle y, \vec{x}_n \rangle$, the algorithm is to crank $M$ (the machine for $Q$) on input $\vec{x}_n$ and keep track of the number of steps of $M$ as it computes. If at its $y$-th step $M$ halts, then print "0" and halt everything.

If $M$ is still cranking at the $y$-th step, then print "1" and halt everything.

Finally, if $M$ has halted *before* its $y$-th step, then again print "1" and halt everything.

By CT, the above algorithm can be formalized into a TM, $N$, which decides $S$, i.e., $S \in \mathcal{R}_*$.

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

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

and

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

$\square$

Here is an application:

**7.16 Example.** The set $A = \{\langle x, y, z \rangle : \phi_x(y) = z\}$ is semi-recursive. Indeed, let the relation $Q(w, x, y, z)$ be defined as

$$Q(w, x, y, z) \stackrel{\text{by Def}}{\equiv} M_x \text{ with input } y \text{ produces output } z \text{ in exactly } w \text{ steps}$$

$Q$ is recursive (by CT), for here is how to decide it:

Given input $\langle w, x, y, z \rangle$. Fetch $M_x$. Crank it for exactly $w$ steps with input $y$, and then halt the simulation. If (and only if!), at this $w$-th step of the simulation, $M_x$ actually has halted on its own (had a terminal ID) ***and*** has produced the (numerical) output $z$ on its tape, then halt everything and print "0". Otherwise, halt everything and print "1".

Clearly,

$$A(x, y, z) \equiv (\exists w) Q(w, x, y, z)$$

and we are done by the Projection Theorem. $\square$

# *8. Why "Recursively Enumerable"?*

In this section we explore the rationale behind the alternative name "recursively enumerable", in short "r.e.", for semi-recursive relations. To avoid cumbersome codings (of $n$-tuples, by single numbers) we restrict attention to the one variable case.

We show that

**8.1 Theorem. (What’s behind the name “r.e.”)**

*Any non empty semi-recursive relation $A$ ($A \subseteq \mathbb{N}$) is the range of some (emphasis:* **total***) recursive function of one variable.*

*Conversely, every set $A$ such that $A = ran(f)$—where $\lambda x.f(x)$ is recursive—is semi-recursive (and, trivially, nonempty).*

Thus, the semi-recursive relations are exactly those that we can ***algorithmically enumerate***, by a recursive function.[†] Hence the name “r.e.”, replacing the non technical “algorithmically” by the technical “recursively”.

Note that we had to hedge and ask that $A \neq \emptyset$, because no recursive function (remember: these are total) can have an empty range.

*Proof.* (A) We prove the first sentence of the theorem. So, let $A \neq \emptyset$ be semi-recursive.

**Case 1.** We look at the “hard case” first, where $A$ is an *infinite set.*

By the projection theorem (7.15) there is a recursive relation $Q(y, x)$ such that

$$x \in A \equiv (\exists y)Q(y, x), \text{ for all } x \tag{1}$$

Let $M$ be a TM that decides $Q(y, x)$.

Here is an algorithm that algorithmically lists *all* of $A$:

Form two lists, **List 1** and **List 2**:

**List 1** is an algorithmic listing of all possible inputs $\langle y, x\rangle$ for $M$ (the machine that we said exists, and decides $Q(y, x)$).

That is, in **List 1** we list

All strings $1^{y+1}01^{x+1}$, for $y = 0, 1, 2, \ldots$ and $x = 0, 1, 2, \ldots$
by length, and in each length group lexicographically, taking $0 < 1$

For example we list (in the length-6 group) as follows

$$101111, 110111, 111011, 111101$$

For each $1^{y+1}01^{x+1}$ that is listed, run machine $M$ ($M$ always halts, being a “decider”). If $M$ prints a “0”, then add $x$ to **List 2**.

**Comment.** This is so because a “0” printout means $Q(y, x)$ is true, hence, by (1), $x \in A$ is true.

We can now use the algorithmic listing, **List 2**, to compute a total function $\lambda x.f(x)$, such that

$$A = \text{ran}(f) = \{f(x) : x \in A\} \tag{2}$$

Given $x$, go down the list **List 2**[‡] until the $x$-th entry is found.

---

[†] Sipser [2] talks of “enumerators.

[‡] We have said, on many occasions, “go down a(n algorithmic) list until you find the $x$-th entry”. This means, of course, that you **generate** the list, **algorithmically**—from scratch—and stop exactly once the $x$-th entry has been generated. It does **not** mean that an entire infinite list just sits there and is being “searched”!

Since the list "**List 2**" of all of $A$ is generated algorithmically, *and has infinite length*, the $x$-th item exists, *and* will be (algorithmically) found.

Output that item and halt.

Clearly, this $f$ satisfies (2). By CT, $f$ is in $\mathcal{R}$ (since it is total, and algorithmic).

**Case 2.** The finite case. Say $A = \{a_0, \ldots, a_k\}$, where I have listed its members in ascending numerical order ($k \geq 0$, since $A \neq \emptyset$). The following $f$ is algorithmic (table look-up!) and does the job (by CT, it is (total) recursive).

$$f(x) = \begin{cases} a_0 & \text{if } x = 0 \\ a_1 & \text{if } x = 1 \\ a_2 & \text{if } x = 2 \\ \vdots & \vdots \\ a_{k-1} & \text{if } x = k-1 \\ a_k & \text{if } x \geq k \end{cases}$$

(B) Proof of the second sentence of the theorem. So, let (2), above, hold. The relation $f(y) = x$ is decidable. Indeed, (using CT) given the input $\langle y, x \rangle$, compute $f(y)$. This will eventually produce a number $z$ as output ($f$ is recursive!). If $x = z$ then print "0" and halt, else print "1" and halt.

Since $x \in A \equiv (\exists y)(f(y) = x)$, we are done by the Projection Theorem. $\square$

# 9. *Some closure properties of decidable and semi-decidable relations*

**9.1 Theorem.** *$\mathcal{R}_*$ is closed under all Boolean operations, $\neg, \wedge, \vee, \Rightarrow, \equiv$.*

*Proof.* It suffices to prove closure under $\neg$ and $\vee$.

The case for $\neg$ has already been witnessed: Any decider can be trivially converted to a decider of the negation (or complement) by reversing the "reporting": "1" for "0", and "0" for "1".

Let next $Q(\vec{x}_n)$ and $S(\vec{y}_m)$ be decidable. This means that

$$c_Q \text{ and } c_S \text{ are recursive} \tag{1}$$

Clearly

(i) $c_{Q \vee S}(\vec{x}_n, \vec{y}_n) = c_Q(\vec{x}_n) \cdot c_S(\vec{y}_n)$, and

(ii) $c_{Q \vee S}$ is recursive.

To see the latter, given the input $\vec{x}_n$ and $\vec{y}_n$, we compute $c_{Q \vee S}(\vec{x}_n, \vec{y}_n)$ as follows: Compute each of $c_Q(\vec{x}_n)$ and $c_S(\vec{y}_n)$. Multiply the results, return the product and halt.

The remaining Boolean operations are expressible via $\neg$ and $\vee$, so there is no more to say. $\square$

Rephrasing in "set-jargon" we get:

**9.2 Corollary.** *Decidable sets are closed under complement, union, intersection.*

**9.3 Remark.** We took the point of view of Computability, in looking at TMs as number processors. As such it is not readily meaningful to ask if decidable sets are closed under string operations, such as concatenation, because the sets of our theory are *sets of numbers, or tuples of numbers.*

Pretend, however, for a moment that we never restricted the input format, and that we allowed our TMs to process strings rather than "numerical inputs"[†]

Under these (hypothetical) circumstances, we ask: Are decidable sets (of strings!) closed under concatenation?

That is, if $A$ and $B$ are decidable, is $A \cdot B$ also decidable?

Easily, yes. Given input $z$ we test "$z \in A \cdot B$" as follows:

For each decomposition of $z$ as $z = xy$[‡] we process (using deciders for $A$ and $B$) the queries $x \in A$ and $y \in B$. If (and only if) some such decomposition allows **both** queries to print "0", then I print a "0" and halt everything.

Otherwise (no decomposition worked), I print "1" and halt everything.

**9.4 Theorem.** *$\mathcal{P}_*$ is closed under $\wedge$ and $\vee$. It is also closed under $(\exists y)$, or, as we say, "under projection".*

*It is* **not** *closed under negation (complement), nor under $(\forall y)$.*

*Proof.* Let $Q(\vec{x}_n)$ be semi-decided by a TM $M$, and $S(\vec{y}_m)$ be semi-decided by a TM $N$.

Here is how to semi-decide $Q(\vec{x}_n) \vee S(\vec{y}_m)$:

Given input $\langle \vec{x}_n, \vec{y}_m \rangle$, we call machine $M$ with input $\vec{x}_n$, and machine $N$ with input $\vec{y}_m$ and let them crank simultaneously ("co-routines").

If **either one** halts, then halt everything!

For $\wedge$ it is almost the same, but our halting criterion is different:

Here is how to semi-decide $Q(\vec{x}_n) \wedge S(\vec{y}_m)$:

Given input $\langle \vec{x}_n, \vec{y}_m \rangle$, we call machine $M$ with input $\vec{x}_n$, and machine $N$ with input $\vec{y}_m$ and let them crank simultaneously ("co-routines").

If **both** halt, then halt everything!

The $(\exists y)$ is very interesting as it introduces a new technique:

Let $Q(y, \vec{x}_n)$ be semi-decidable, say, by machine $M$.

Here is how to semi-decide $(\exists y)Q(y, \vec{x}_n)$:

Idea: Given an input $\vec{x}_n$ for $(\exists y)Q(y, \vec{x}_n)$, we want to **search** for a $y$ such that the input $\langle y, \vec{x}_n \rangle$—supplied to $M$—makes $M$ halt. Obviously we cannot try each $y = 0, 1, 2, \ldots$ ***in turn*** until we succeed, because it is possible that, for example, $\langle 0, \vec{x}_n \rangle$ makes $M$ "loop forever", while $\langle 1, \vec{x}_n \rangle$ makes it halt!

[†] True, numerical inputs are also denoted by strings, but strings of a **very restricted** type.

[‡] By "$xy$" I mean concatenation of $x$ and $y$ in that order. Note that there are exactly $|z|+1$ such decompositions, "$|z|$" denoting the length of $z$.

Instead we would like to try "all the values $0, 1, 2, \ldots$ of $y$ in parallel"—running infinitely many copies of $M$, one for each input $\langle y, \vec{x}_n \rangle$,[†], and halt **iff** any $M$-copy halted.

Of course, this "infinite parallelism" is nonsense, and no amount of Church's Thesis will save us if we attempt it.

The next best thing to do is use "the poor person's parallelism", known as "dovetailing" in the theory.[‡]

**proc** $EQ(\vec{x}_n)$ /* To semi-decide $(\exists y)Q(y, \vec{x}_n)$ */
$T \leftarrow 1$
**while**(1)
{

1. Perform exactly $T$ steps of the $M$-computation, ***for each of the finitely many inputs*** $\langle y, \vec{x}_n \rangle$ obtained by letting $y = 0, 1, 2, 3, \ldots, T-1$.
2. If **any** input $\langle y, \vec{x}_n \rangle$, above, causes $M$ to halt, then halt everything! (**exit**)
3. $T \leftarrow T + 1$

}

The above pseudo-code semi-decides $(\exists y)Q(y, \vec{x}_n)$ because it does search systematically all the $y$'s, until it finds one that works, **iff** there is one (in which case the process halts).

By CT (which guarantees a formal version of our code), this makes $(\exists y)Q(y, \vec{x}_n)$ semi-decidable.

Why is $\mathcal{P}_*$ not closed under negation (complement)?
Because we know that $H \in \mathcal{P}_*$, but $\overline{H} \notin \mathcal{P}_*$.

Why is $\mathcal{P}_*$ not closed under $(\forall y)$?
This is slightly trickier: $x \in \overline{H}$ is not semi-decidable—we know this.
Now,

$$x \in \overline{H} \equiv \phi_x(x) \uparrow$$

Let us translate to machines, computations, and "steps":

$$\phi_x(x) \uparrow \equiv M_x \text{ with input } x \text{ has } \textbf{no} \text{ "terminating computation"}$$

or

$$\phi_x(x) \uparrow \equiv \text{for all } y,\ M_x \text{ with input } x \text{ has } \textbf{no} \text{ computation that halts in } y \text{ steps}$$

Aha! But we know that

$$\text{"}M_x \text{ with input } x \text{ has } \textbf{no} \text{ computation that halts in } y \text{ steps"} \tag{1}$$

is decidable (We do? Where from?).

[†]That is, testing $Q(y, \vec{x}_n)$, for all the $y$ values simultaneously.
[‡]An Operating Systems expert, which I am not, may call it "time-slicing".

So, the **NOT** semi-decidable $x \in \overline{H}$ is equivalent to some $(\forall y)Q(y,x)$, where $Q(y,x)$ **IS** decidable hence also semi-decidable ($Q(y,x)$ here is the statement in quotes in (1) above). We rest our case. $\square$

**9.5 Example.** We look at an easy application (which would not be so easy if we did not have the theorem above!).

Suppose that $f$ is partial recursive.

We prove that the relation $y \in \mathrm{ran}(f)$ is semi-recursive. Indeed, let $f = \phi_e$, for some fixed $e$. We already know (see Example 7.16) that $\phi_z(x) = y$ is semi-recursive, hence, "freezing $z$" (i.e., substituting the constant $e$ into $z$), we obtain the semi-recursive $\phi_e(x) = y$.

**Digression.** Two things: **One**, we may use $\lambda$-notation to indicate "freezing" of variables. I could write just $\lambda xy.\phi_z(x) = y$. Now I know that $z$ is **not** an input! I can also write $\lambda xy.\phi_e(x) = y$. The name of the "constant" (or "parameter", i.e., the "non-input") is immaterial. It could be named $e, z, w$, whatever.

**Two**, a question: Why freezing an input of a recognizable relation leads to another recognizable relation? Is this also true for decidable relations? **End of Digression.**

We now continue: $y \in \mathrm{ran}(f) \equiv (\exists x)\phi_e(x) = y$. Case rests by Theorem 9.4.

# Bibliography

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

[2] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Co., Boston, 1997.