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

0.1. Recursively Enumerable Sets

In this section we explore the rationale behind the alternative name "*recursively enumerable*" or "*computably enumerable*" that is used in the literature for the semi-recursive or semi-computable sets/predicates. Short names for this alternative jargon are "r.e." and "c.e." respectively. To avoid cumbersome codings (of n-tuples, by single numbers) we restrict attention to the one variable case in this section. I.e., predicates that are subsets of $\mathbb{N}^n$ for the case $n = 1$.

First we define:

0.1.1 Definition. A set $A \subseteq \mathbb{N}$ is called *computably enumerable* (c.e.) or *recursively enumerable* (r.e.) precisely if one of the following cases holds:

- $A = \emptyset$
- $A = \text{ran}(f)$, where $f \in \mathcal{R}$.

□

Thus, the c.e. or r.e. relations are exactly those we can ***algorithmically enumerate*** as **the set of outputs** of a (*total*) recursive function:

$$A = \{f(0), f(1), f(2), \ldots, f(x), \ldots\}$$

Hence the use of the term "c.e." replaces the non technical term "algorithmically" (in "algorithmically" enumerable) by the technical term "computably".

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

Next we prove:

0.1.2 Theorem. ("c.e." or "r.e." vs. semi-recursive)
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).

For short, the semi-recursive sets are precisely the c.e. or r.e. sets. For $A \neq \emptyset$ this is the content of 0.1.2 while $\emptyset$ is r.e. by definition and known to us to be also semi-recursive.

Before we prove the result, here is an example:

0.1.3 Example. The set $\{0\}$ is c.e. Indeed, $f = \lambda x.0$, our familiar function Z, effects the enumeration with repetitions (lots of them!)

x	$=0$	1	2	3	4	...
$f(x)$	$=0$	0	0	0	0	...

□

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

By the projection theorem (see Notes #7) there is a **decidable** (recursive) relation $Q(y, x)$ such that

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

Thus,

every $x \in A$ will have some associated value y such that $Q(y, x)$ holds. (2)

and conversely,

if $Q(y, x)$ holds for some y, x pair, then $x \in A$. (2′)

(2) and (2′) jointly rephrase (1), but also suggest a very high level process to **enumerate** all $x \in A$: We should look for **all** pairs (y, x) in a **systematic manner**, and for each such pair that is in Q we should just output (enumerate) the x-component!

Ah! But we know how to generate all pairs in a systematic manner, algorithmically!

for $z = 0, 1, 2, 3, \ldots$ **do**
generate the pair consisting of $(z)_0$ (represents "y" in the blue text above) and $(z)_1$ (represents "x").

Wait! Will the above generate **all** pairs? Sure: For any x and y, the pair (y, x) is guaranteed to be output when we reach the z-value $\langle y, x\rangle \quad (= 2^{y+1}3^{x+1})$.

Here is then how to enumerate all of A (**first draft**!)

for $z = 0, 1, 2, 3, \ldots$ **do**
{

1. **generate** the pair $\Big((z)_0, (z)_1\Big)$
2. if $Q\Big((z)_0, (z)_1\Big)$ is true then output $(z)_1$
3. if $Q\Big((z)_0, (z)_1\Big)$ is false then do not output anything in this iteration

}

Is the above algorithm correct? Well, yes:

- One, if $x \in A$, then some y certifies (1) above, that is $Q(y, x)$ is true. But when we iterate with $z = 2^{y+1}3^{x+1}$ we will have $Q(y, x)$ verified and indeed the algorithm will output $x = (z)_1$ (step 2).
- Two, if $x \notin A$, then no y paired with x will make $Q(y, x)$ true, and unless we verify such truth we do not output anything (step 3. in the algorithm).

By CT, the above procedure, mathematised in our theory via URMs, defines a **computable function** $\lambda z.f(z)$ such that $\mathrm{ran}(f) = A$, that is f enumerates A.

But, you will protest, we did not use the assumption $A \neq \emptyset$, nor seems that we can conclude that f is total —see step 3. above, which declines output.

That is why I said "first draft" earlier! So, *let us work with* $A \neq \emptyset$. Then there is an $a \in A$. We do not need to find it! The theorem says there **IS** a recursive enumerating function. It does not say **construct** one! So, with the guarantee of $A \neq \emptyset$, and fixing attention on **one** $a \in A$, we modify the f above as

for $z = 0, 1, 2, 3, \ldots$ **do**
$\{$

1. **generate** the pair $\Big((z)_0, (z)_1\Big)$
2. if $Q\Big((z)_0, (z)_1\Big)$ is true then output $(z)_1$
3. if $Q\Big((z)_0, (z)_1\Big)$ is false then output a **Comment**. It is always correct to output a.

$\}$

Trivially, the function defined by the red program still enumerates A (overdoing the case of $a \in A$) but now the function defined **is total!**

(II) **Proof of the second sentence of the theorem**. So, let $A = \text{ran}(f)$ —where f is recursive. Thus,

$$x \in A \equiv (\exists y) f(y) = x \tag{1}$$

By Grz-Ops, the fact that $z = x$ is in $\mathcal{R}_*$ and the assumption $f \in \mathcal{R}$, the relation $f(y) = x$ is decidable (recursive). By (1) we are done by the Projection Theorem. □

0.1.4 Corollary. *If A is semi-recursive, then $A = \text{ran}(f)$ for some $f \in \mathcal{P}$.*

Proof. This follows from the "draft" solution to Part (I) of the previous proof. Indeed, if $A \neq \emptyset$, then use the draft solution to obtain an $f \in \mathcal{P}$ such that $A = \text{ran}(f)$. If $A = \emptyset$ —of course such A is in $\mathcal{P}\mathcal{R}_* \subseteq \mathcal{R}_* \subseteq \mathcal{P}_*$— then A is the range of the empty function which is in $\mathcal{P}$. □

Do we have a converse? Is the range of any partial recursive function semi-recursive? **Yes**! Wait for Section 0.3, Theorem 0.3.2.

0.1.5 Corollary. *An $A \subseteq \mathbb{N}$ is semi-recursive iff it is r.e. (c.e.)*

Proof. For nonempty A this is Theorem 0.1.2. For empty A we note that this is r.e. by 0.1.2 but also semi-recursive by $\emptyset \subseteq \mathcal{P}\mathcal{R}_* \subseteq \mathcal{R}_* \subseteq \mathcal{P}_*$. □


Corollary 0.1.5 allows us to prove some non-semi-recursiveness results by good old-fashioned Cantor diagonalisation. See below.


0.1.6 Theorem. *The complete index set $A = \{x : \phi_x \in \mathcal{R}\}$ is not semi-recursive.*


This sharpens the undecidability result for A that we established in Note #7.


Proof. By the equivalence of c.e.-ness and semi-recursiveness we prove that A is not c.e.

If not, note first that $A \neq \emptyset$ since, e.g., $Z \in \mathcal{R}$ and thus at least one ϕ-index is in A (a ϕ-index for Z).

Thus, theorem 0.1.2 applies and there is an $f \in \mathcal{R}$ such that $A = \text{ran}(f) = \{y : y = f(x), \text{ for some } x\}$, that is

$$y \in A \equiv (\exists x) f(x) = y$$

In words, a ϕ-index y is in A **iff** it has the form $f(x)$ for some x.

Define

$$d = \lambda x.1 + \phi_{f(x)}(x) \tag{1}$$

Seeing that $\phi_{f(x)}(x) = h(f(x), x)$, we obtain $d \in \mathcal{P}$. But $\phi_{f(x)}$ is total since all the $f(x)$ are ϕ-indices of total functions by the red comment above.

By the same comment,

$$d = \phi_{f(i)}, \text{ for some } i \tag{2}$$

Let us compute $d(i)$: $d(i) = 1 + \phi_{f(i)}(i)$ by (1). Also, $d(i) = \phi_{f(i)}(i)$ by (2), thus

$$1 + \phi_{f(i)}(i) = \phi_{f(i)}(i)$$

which is a contradiction since both sides of "=" are defined. □

One can take as d different functions, for example, either of $d = \lambda x.42 + \phi_{f(x)}(x)$ or $d = \lambda x.1 \dot{-} \phi_{f(x)}(x)$ works. And infinitely many other choices do!

0.2. Some closure properties of decidable and semi-decidable relations

We already know that

0.2.1 Theorem. *$\mathcal{R}_*$ is closed under all Boolean operations, $\neg, \wedge, \vee, \Rightarrow, \equiv$, as well as under $(\exists y)_{<z}$ and $(\forall y)_{<z}$.*

0.2.2 Remark. We took the point of view of Computability, as founded via URMs, as the theory of computing functions of (natural) number inputs and outputs. 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 URMs to process strings rather than "numerical inputs"[†]

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

Under these (hypothetical) circumstances, we ask at the very informal level: 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" ("yes"), then I print a "0" and halt everything.

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

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

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

Proof.

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

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

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

   If **either one** halts, then halt everything! This is the case of "yes" (input verified).

2. 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 $\vec{x}_n, \vec{y}_m$, 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!

3. **The $(\exists y)$ is very interesting as it relies on the Projection Theorem:**

   Let $Q(y, \vec{x}_n)$ be **semi**-decidable. Then, by Projection Theorem, a **decidable** $P(z, y, \vec{x}_n)$ exists such that

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

   It follows that

   $$(\exists y) Q(y, \vec{x}_n) \equiv (\exists y)(\exists z) P(z, y, \vec{x}_n) \tag{2}$$

   This does not settle the story, as I cannot readily conclude that $(\exists y)(\exists z) P(z, y, \vec{x}_n)$ is semi-decidable because the Projection Theorem requires a *single* $(\exists y)$

[‡]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.

in front of a decidable predicate!

What do I do? Coding to the rescue!

Well, instead of saying that there are **two** values y and z that verify (along with $\vec{x}_n$) the predicate $P(z, y, \vec{x}_n)$, I can say the same thing —after setting $w = 2^{z+1}3^{y+1} = \langle z, y\rangle$— as $(\exists w)P((w)_0, (w)_1, \vec{x}_n)$ and thus I have

$$(\exists y)Q(y, \vec{x}_n) \equiv (\exists w)P((w)_0, (w)_1, \vec{x}_n) \tag{3}$$

But now $P((w)_0, (w)_1, \vec{x}_n)$ is decidable by the decidability of P and Grz-Ops, and in (3) we quantified the decidable $P((w)_0, (w)_1, \vec{x}_n)$ with just **one** $(\exists w)$. The Projection Theorem now applies!

4. For $(\exists y)_{<z}Q(y, \vec{x})$, where $Q(y, \vec{x})$ is semi-recursive, we first note that

$$(\exists y)_{<z}Q(y, \vec{x}) \equiv (\exists y)\Big(y < z \wedge Q(y, \vec{x})\Big) \tag{$*$}$$

By $\mathcal{PR}_* \subseteq \mathcal{R}_* \subseteq \mathcal{P}_*$, $y < z$ is semi-recursive. By closure properties established already in this proof, the rhs of $\equiv$ in $(*)$ is semi-recursive, thus so is the lhs.

5. For $(\forall y)_{<z}Q(y, \vec{x})$, where $Q(y, \vec{x})$ is semi-recursive, we first note that (by Strong Projection) a **decidable** P exists such that

$$Q(y, \vec{x}) \equiv (\exists w)P(w, y, \vec{x})$$

By the above equivalence, we need to prove that

$$(\forall y)_{<z}(\exists w)P(w, y, \vec{x}) \text{ is semi-recursive} \tag{$**$}$$

$(**)$ says that

for **each** $y = 0, 1, 2, \ldots, z-1$ there is a w-value w_y so that $P(w_y, y, \vec{x})$ holds

Since all those w_y are finitely many (z many!) there is a value u bigger than **all** of them (for example, take $u = \max(w_0, \ldots, w_{z-1}) + 1$). Thus $(**)$ says (i.e., **is equivalent to**)

$$(\exists u)(\forall y)_{<z}(\exists w)_{<u}P(w, y, \vec{x})$$

The blue part of the above is **decidable** (by closure properties of $\mathcal{R}_*$, since $P \in \mathcal{R}_*$ —you may peek at 0.2.1). We are done by *strong projection*.

Why not work as in case 4. above and use the equivalent to $(\forall y)_{<z}Q(y, \vec{x})$ expression $(\forall y)\Big(y < z \to Q(y, \vec{x})\Big)$?

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

7. Why is $\mathcal{P}_*$ not closed under $(\forall y)$?
   Well,
   $$x \in K \equiv (\exists y) Q(y, x) \tag{1}$$
   for some recursive Q (Projection Theorem) and by the fact (quoted again above) that $K \in \mathcal{P}_*$.
   (1) is equivalent to
   $$x \in \overline{K} \equiv \neg(\exists y) Q(y, x)$$
   which in turn is equivalent to
   $$x \in \overline{K} \equiv (\forall y) \neg Q(y, x) \tag{2}$$
   Now, by closure properties of $\mathcal{R}_*$, $\neg Q(y, x)$ is recursive, hence also in $\mathcal{P}_*$ since $\mathcal{R}_* \subseteq \mathcal{P}_*$.

   So, if $\mathcal{P}_*$ were closed under $(\forall y)$, then the above $(\forall y)\neg Q(y, x)$ would be semi-recursive. But that is $x \in \overline{K}$! □

0.3. Computable functions and their graphs

We prove a fundamental result here, that

0.3.1 Theorem. *$\lambda\vec{x}.f(\vec{x}) \in \mathcal{P}$ iff the graph $y = f(\vec{x})$ is in $\mathcal{P}_*$.*

Proof.

- ($\rightarrow$, that is, the **Only if**) Let $\lambda\vec{x}.f(\vec{x}) \in \mathcal{P}$. By an easy adaptation of the proof in Example 0.2.14 of Notes #7 (the universal function h is NOT involved here!) it follows that $y = f(\vec{x})$ is semi-computable.
- ($\leftarrow$, that is, the **If**) Let $y = f(\vec{x})$ be semi-computable.

  Here is an obvious idea: Let M be a **verifier** for $y = f(\vec{x})$. Program as follows:

  1. **for** $z = 0, 1, 2, \ldots$ **do**:
  2. **if** M **verified** $z = f(\vec{x})$ **then return** (z)

Let us emphasise: The verifier M does not compute $f(\vec{x})$ but rather verifies when a pair $z, \vec{x}$ *belongs to the graph* of f. If we knew *a priori* how to compute $f(\vec{x})$ we would not need to deal with the graph and its verifier at all!

Alas, the above idea does **not** work! For any z-value that is $< f(\vec{x})$ in the above search for the "correct" z[†] the verifier says "**no**" by **looping** forever! We will never reach the correct z, *if there is one.*[‡]

We must be more sophisticated in **what** and **how** we are searching for:

By (strong projection theorem)

$$y = f(\vec{x}) \equiv (\exists z) Q(z, y, \vec{x}) \tag{1}$$

for some decidable Q. The idea of how to find the correct y, **if any**, once we are *given* an $\vec{x}$, is to search (**simultaneously**!) for a z **and** y that "work" —i.e., they satisfy $Q(z, y, \vec{x})$ for the given $\vec{x}$.[§] So, informally, we search the sequence

$$w = 0, 1, 2, 3, \ldots$$

and stop as soon as we note that $Q((w)_0, (w)_1, \vec{x})$ is true —if this ever happens!

As $(w)_0$ plays the role of z and $(w)_1$ plays the role of y, we obviously report $(w)_1$ as our answer, **if and when we stop the search**.

Mathematically,

$$f(\vec{x}) = \Big((\mu w) Q((w)_0, (w)_1, \vec{x}) \Big)_1$$

f is in $\mathcal{P}$ by closure properties. □

We can now settle

0.3.2 Theorem. *If $A = \mathrm{ran}(f)$ and $f \in \mathcal{P}$, then $A \in \mathcal{P}_*$.*

Proof. By 0.3.1 $y = f(x)$ is semi-recursive. By closure properties of $\mathcal{P}_*$, so is $(\exists x) y = f(x)$. But $(\exists x) y = f(x) \equiv y \in \mathrm{ran}(f)$, that is, $(\exists x) y = f(x) \equiv y \in A$ since $\mathrm{ran}(f) = A$. Done. □

0.4. Some tricky reductions

This section highlights a more sophisticated reduction scheme that improves our ability to effect reductions of the type $\overline{K} \leq A$.

0.4.1 Example. Prove that $A = \{x : \phi_x \text{ is a constant}\}$ is not semi-recursive. This is not amenable to the technique of saying "OK, if A is semi-recursive, then it is r.e. Let me show that it is not so by diagonalisation". This worked

[†] That is, such that $z = f(\vec{x})$.
[‡] It may well be that $f(\vec{x}) \uparrow$ for the given $\vec{x}$.
[§] We saw this idea in the proof of Theorem 0.1.2 at the beginning of this note.

for $B = \{x : \phi_x \text{ is total}\}$ but no obvious diagonalisation comes to mind for A.

Nor can we simplistically say, OK, start by defining

$$g(x,y) = \begin{cases} 0 & \text{if } x \in \overline{K} \\ \uparrow & \text{othw} \end{cases}$$

The problem is that if we plan next to say "by CT g is partial recursive hence by S-m-n, etc.", then the underlined part is wrong. $g \notin \mathcal{P}$, *provably*! For if it is computable, then so is $\lambda x.g(x,x)$ by Grz-Ops. But

$$g(x,x)\downarrow \text{ iff we have the top case, iff } x \in \overline{K}$$

Thus

$$x \in \overline{K} \equiv g(x,x)\downarrow$$

which proves that $\overline{K} \in \mathcal{P}_*$ using the verifier for "$g(x,x)\downarrow$". **Contradiction**. □

0.4.2 Example. (0.4.1 continued) Now, "**Plan B**" is to "**approximate**" the top condition $\phi_x(x)\uparrow$ (same as $x \in \overline{K}$).

The idea is that, "**practically**", if the computation $\phi_x(x)$ after a "huge" number of steps y has still not hit **stop**, this situation approximates —let me say once more— "practically", the situation $\phi_x(x)\uparrow$. This fuzzy thinking suggests that we try next

$$f(x,y) = \begin{cases} 0 & \text{if the computation } \phi_x(x) \text{ has not reached } \textbf{stop} \text{ after } y \text{ steps} \\ \uparrow & \text{othw} \end{cases}$$

The "othw" says, of course, that the computation of the call $\phi_x(x)$ —or $h(x,x)$, where h is the universal function— did halt in y steps or fewer.

Next step is to enable the S-m-n theorem application, so we must show that f defined above is computable. Well here is an informal algorithm:

(0) **proc** $f(x,y)$
(1) **Call** $h(x,x)$, that is, $\phi_x(x)$, and keep count of its computation steps
(2) **Return** 0 if $\phi_x(x)$ did **not** hit **stop** in y steps
(3) **Loop** if $\phi_x(x)$ **halted** in $\leq y$ steps

Of course, the "command" **Loop** means

"transfer to the subprogram" **while** 1=1 **do** { }

By CT, the pseudo algorithm (0)–(3) is implementable as a URM. That is, $f \in \mathcal{P}$.

By S-m-n applied to f there is a recursive k such that

$$\phi_{k(x)}(y) = \begin{cases} 0 & \text{if } \phi_x(x) \text{ is still not at } \textbf{stop} \text{ after } y \text{ steps} \\ \uparrow & \text{othw} \end{cases} \tag{1}$$

Analysis of (1) in terms of the "key" conditions $\phi_x(x)\uparrow$ and $\phi_x(x)\downarrow$:

(A) Case where $\phi_x(x)\uparrow$.

Then, $\phi_x(x)$ did **not** halt in y steps, for any y!

Thus, by (1), we have $\phi_{k(x)}(y) = 0$, for all y, that is,

$$\phi_x(x)\uparrow \implies \phi_{k(x)} = \lambda y.0 \tag{2}$$

(B) Case where $\phi_x(x)\downarrow$. Let m = *smallest* y such that the call $\phi_x(x)$ —i.e., $h(x,x)$— ended in m steps. Therefore,

- for step counts $y = 0, 1, 2, \ldots, m-1$ the computation of $h(x,x)$ has not yet hit stop, so the **top** case of definition (1) holds. We get

  for y $=0, \quad 1, \quad \ldots, \quad m-1$
  $\phi_{k(x)}(y)=0, \quad 0, \quad \ldots, \quad 0$

- for step counts $y = m, m+1, m+2, \ldots$ the computation of $h(x,x)$ has already halted (it hit **stop**), so the **bottom** case of definition (1) holds. We get

  for y $=m, \quad m+1, \quad m+2, \quad \ldots$
  $\phi_{k(x)}(y)=\uparrow, \quad \uparrow, \quad \uparrow, \quad \ldots$

  for short:

$$\phi_x(x)\downarrow \implies \phi_{k(x)} = \overbrace{(0,0,\ldots,0)}^{\text{length } m} \tag{3}$$

In

$$\phi_{k(x)} = \overbrace{(0,0,\ldots,0)}^{\text{length } m}$$

we depict the function $\phi_{k(x)}$ as an array of m output values.

Two things: *One*, in English, when $\phi_x(x)\downarrow$, the function $\phi_{k(x)}$ is NOT a constant! Not even total!

Two, m depends on x, of course, when said x brings us to case (B). Regardless, the non-constant / non total nature of $\phi_{k(x)}$ is still a fact; just the length m of the finite array $\overbrace{(0,0,\ldots,0)}^{\text{length } m}$ changes.

Our analysis yielded:

$$\phi_{k(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x)\uparrow \\ \text{not a constant function} & \text{if } \phi_x(x)\downarrow \end{cases} \tag{4}$$

We conclude now as follows for $A = \{x : \phi_x$ is a constant$\}$:

$k(x) \in A$ iff $\phi_{k(x)}$ is a constant iff the top case of (4) applies iff $\phi_x(x)\uparrow$

That is, $x \in \overline{K} \equiv k(x) \in A$, hence $\overline{K} \leq A$. □

0.4.3 Example. Prove (again) that $B = \{x : \phi_x \in \mathcal{R}\} = \{x : \phi_x \text{ is total}\}$ is not semi-recursive.

We piggy back on the previous example and the same f through which we found a $k \in \mathcal{R}$ such that

$$\phi_{k(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x)\uparrow \\ \overbrace{(0,0,\ldots,0)}^{\text{length } m} & \text{if } \phi_x(x)\downarrow \end{cases} \tag{5}$$

The above is (4) of the previous example, but we will use different words now for the bottom case, which we displayed explicitly in (5). Note that $\overbrace{(0,0,\ldots,0)}^{\text{length } m}$ is a non-recursive (nontotal) function listed as a finite array of outputs. Thus we have

$$\phi_{k(x)} = \begin{cases} \lambda y.0 & \text{if } \phi_x(x)\uparrow \\ \text{nontotal function} & \text{if } \phi_x(x)\downarrow \end{cases} \tag{6}$$

and therefore

$k(x) \in B$ iff $\phi_{k(x)}$ is total iff the top case of (6) applies iff $\phi_x(x)\uparrow$

That is, $x \in \overline{K} \equiv k(x) \in B$, hence $\overline{K} \leq B$. □

0.4.4 Example. In Notes #7, Exercise 0.2.11 I ask you to prove that $D = \{x : \text{ran}(\phi_x) \text{ is infinite}\}$ is not recursive. This is directly based on Example 0.2.10 of Notes #7, and the work on the complement of D, that was called $C = \{x : \text{ran}(\phi_x) \text{ is finite}\}$ in said example.

Here I show that D actually is not **semi**-recursive either, a fact that furnishes an example of a set that neither it, nor its complement are semi-recursive!

We (heavily) piggy back on Example 0.4.2 above. We want to find $j \in \mathcal{R}$ such that

$$\phi_{j(x)} = \begin{cases} \text{inf. range} & \text{if } \phi_x(x)\uparrow \\ \text{finite range} & \text{if } \phi_x(x)\downarrow \end{cases} \tag{$*$}$$

OK, define ψ (almost) like f of Example 0.4.2 by

$$\psi(x,y)=\begin{cases} y & \text{if the computation } \phi_x(x) \text{ has still not hit } \textbf{stop} \text{ after } y \text{ steps} \\ \uparrow & \text{othw} \end{cases}$$

other than the trivial difference (function name) the important difference is that we force infinite range in the top case by outputting the input y.

The argument that $\psi \in \mathcal{P}$ goes as the one for f in Example 0.4.2. The only difference is that in the algorithm (0)–(3) we change "**Return** 0" to "**Return** y".

The question $\psi \in \mathcal{P}$ settled, by S-m-n there is a $j \in \mathcal{R}$ such that

$$\phi_{j(x)}(y)=\begin{cases} y & \text{if the computation } \phi_x(x) \text{ has not hit } \textbf{stop} \text{ after } y \text{ steps} \\ \uparrow & \text{othw} \end{cases} \qquad (\dagger)$$

Analysis of (†) **in terms of the "key" conditions** $\phi_x(x)\uparrow$ **and** $\phi_x(x)\downarrow$**:**

(I) Case where $\phi_x(x)\uparrow$.

Then, for all input values y, $\phi_x(x)$ is still not at **stop** after y steps. Thus by (†), we have $\phi_{j(x)}(y)=y$, for all y, that is,

$$\phi_x(x)\uparrow \Longrightarrow \phi_{j(x)}=\lambda y.y \qquad (1)$$

(II) Case where $\phi_x(x)\downarrow$. Let m = *smallest* y such that the call $\phi_x(x)$ — i.e., $h(x,x)$— ended in m steps. Therefore, as before we find that for $y=0,1,\ldots,m-1$ we have $\phi_{j(x)}(y)=y$, that is,

for y	$=0,$	$1,$	$\ldots,$	$m-1$
$\phi_{j(x)}(y)$	$=0,$	$1,$	$\ldots,$	$m-1$

and as before,

for y	$=m,$	$m+1,$	$m+2,$	$\ldots$
$\phi_{j(x)}(y)$	$=\uparrow,$	$\uparrow,$	$\uparrow,$	$\ldots$

that is,

$$\phi_x(x)\downarrow \Longrightarrow \phi_{j(x)}=(0,1,\ldots,m-1) \text{ —finite range} \qquad (2)$$

(1) and (2) say that we got $(*)$ —p.12— above. Thus

$j(x)\in D$ iff $\mathrm{ran}(\phi_{j(x)})$ is infinite, iff we have the top case, iff $\phi_x(x)\uparrow$

Thus $\overline{K}\leq D$ via j. □

0.5. An application of the GraphTheorem

A definition like

$$f(x,y)=\begin{cases}0 & \text{if } x\in K\\ \uparrow & \text{othw}\end{cases} \tag{1}$$

is a special case of a so-called "Definition by Positive Cases". That is

- The cases listed explicitly (here $x\in K$) are semi-recursive, but the "othw" is **not** semi-recursive. Therefore, as the latter cannot be **verified**, we let the function output be undefined in this case.

In *any* definition by cases

$$g(\vec{x})=\begin{cases}\vdots & \vdots\\ g_i(\vec{x}) & \text{if } R_i(\vec{x})\\ \vdots & \vdots\end{cases}$$

we have

$$\text{If } R_i(\vec{x}) \text{ then } g_i(\vec{x})$$

that is, we only need **verify** $R_i(\vec{x})$ —even if it is (primitive)recursive— to select the answer $g_i(\vec{x})$. However, in the **(primitive)recursive case** the "othw" is the negation of $R_1(\vec{x})\vee R_2(\vec{x})\vee\ldots\vee R_m(\vec{x})$, where $R_m(\vec{x})$ is the last explicit condition/case. By closure properties of $\mathcal{R}_*$, the "othw" case is recursive as well.

- In a Definition by Positive Cases the g_i are partial recursive.

The general form of Definition by Positive Cases is

$$g(\vec{x})=\begin{cases}\vdots & \vdots\\ g_i(\vec{x}) & \text{if } R_i(\vec{x})\\ \vdots & \vdots\\ g_k(\vec{x}) & \text{if } R_k(\vec{x})\\ \uparrow & \text{othw}\end{cases} \tag{2}$$

where the g_i are in $\mathcal{P}$ and the R_i are in $\mathcal{P}_*$.

Note that $\mathcal{P}_*$ is **not** closed under negation, thus the "othw" in (2) is not in general semi-recursive. This is so in the case of (1) where the "othw" is $x\in\overline{K}$.

Does a definition like (2) yield a partial recursive g?

Yes:

0.5.1 Theorem. *g in (2), under the stated conditions, is partial recursive.*

Proof. We use the graph theorem, so it suffices to prove

$$y = g(\vec{x}) \text{ is semi-recursive} \tag{3}$$

Now, (3) is true precisely when $g(\vec{x}) \downarrow$ **and** the output is the **number** y. For this to happen, *some* **explicit** condition $R_i(\vec{x})$ was true and $y = g_i(\vec{x})$ was also true. For short, $y = g_i(\vec{x}) \wedge R_i(\vec{x})$ was true. Thus we prove (3) by noting

$$y = g(\vec{x}) \equiv y = g_1(\vec{x}) \wedge R_1(\vec{x}) \vee y = g_2(\vec{x}) \wedge R_2(\vec{x}) \vee \ldots \vee y = g_k(\vec{x}) \wedge R_k(\vec{x})$$

The rhs of $\equiv$ is semi-recursive since each $R_i(\vec{x})$ is (given) and each $y = g_i(\vec{x})$ is ($g_i \in \mathcal{P}$ and 0.3.1) at which point we invoke closure properties of $\mathcal{P}_*$ (0.2.3). □

The immediate import of 0.5.1 is that, for example, we can prove without using CT that functions given as in (1), p.14, are in $\mathcal{P}$.