March 5

0.1 Semi-computable Relations; Unsolvability

We next define a $\mathcal{P}$-counterpart of $\mathcal{R}_*$ and $\mathcal{PR}_*$ and look into some of its closure properties.

0.1.1 Definition. (Semi-computable Relations) A relation $P(\vec{x})$ is called *semi-computable* or *semi-recursive* iff for some $f \in \mathcal{P}$, we have, for all $\vec{x}_n$,

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

The set of all semi-computable relations is denoted by $\mathcal{P}_*$.

If $f = \phi_a^{(n)}$ in (1) above, then we say that "a is *a semi-computable index* or just a *semi-index* of $P(\vec{x}_n)$". If $n = 1$ (thus $P \subseteq \mathbb{N}$) and a is one of the semi-indices of P, then we write $P = W_a$ [Rog67]. □

We are making the symbol $\mathcal{P}_*$ up, in complete analogy with the symbols $\mathcal{PR}_*$ and $\mathcal{R}_*$. It is not standard in the literature.

We have at once:

0.1.2 Theorem. (Normal Form Theorem for Semi-computable Relations)

$P(\vec{x}_n) \in \mathcal{P}_*$ iff, for some $a \in \mathbb{N}$, we have (for all $\vec{x}_n$) $P(\vec{x}_n) \equiv (\exists z)T^{(n)}(a, \vec{x}_n, z)$.

Proof. Only if-part. Let $P(\vec{x}_n) \equiv f(\vec{x}_n)\downarrow$, with $f \in \mathcal{P}$. Then, $f = \phi_a^{(n)}$ for some $a \in \mathbb{N}$.

If-part: The given equivalence translates into $P(\vec{x}_n) \equiv \phi_a^{(n)}(\vec{x}_n)\downarrow$. But $\phi_a \in \mathcal{P}$. □

0.1.3 Corollary. (Strong Projection Theorem) *$P(\vec{x}_n) \in \mathcal{P}_*$ iff, for some recursive predicate $Q(\vec{x}_n, z)$, we have (for all $\vec{x}_n$) $P(\vec{x}_n) \equiv (\exists z)Q(\vec{x}_n, z)$.*

Proof. ($\rightarrow$): Say $P(\vec{x}_n) \in \mathcal{P}_*$. Then invoke the theorem above ("$Q(\vec{x}, z)$" will be $T^{(n)}(a, \vec{x}_n, z)$) for an appropriate a.

($\leftarrow$): Let $P(\vec{x}_n) \equiv (\exists z)Q(\vec{x}_n, z)$. This is the same as $P(\vec{x}_n) \equiv (\mu z)Q(\vec{x}_n, z)\downarrow$. But $\lambda\vec{x}_n.(\mu z)Q(\vec{x}_n, z) \in \mathcal{P}$. □

0.1.4 Corollary. *$P(\vec{x}_n) \in \mathcal{P}_*$ iff, for some $\lambda\vec{x}_n.g(\vec{x}_n) \in \mathcal{P}$, we have (for all $\vec{x}_n$) $P(\vec{x}_n) \equiv g(\vec{x}_n) = 0$.*

Proof. The *only if* is immediate from 0.1.1: Let $f \in \mathcal{P}$ such that, for all $\vec{x}_n$, $P(\vec{x}_n) \equiv f(\vec{x}_n)\downarrow$. Take $g = \lambda\vec{x}_n.Z\Big(f(\vec{x}_n)\Big)$.

For the *if*, note that $g = \phi_i^{(n)}$ for some i, thus

$$P(\vec{x}_n) \equiv g(\vec{x}_n) = 0 \text{ and } g(\vec{x}_n) = 0 \equiv (\exists z)\left(T^{(n)}(i, \vec{x}_n, z) \wedge d(z) = 0\right)$$

We are done by 0.1.3.

Alternatively, the function

$$h(x) = \begin{cases} 0 & \text{if } x = 0 \\ \uparrow & \text{otw} \end{cases}$$

is in $\mathcal{P}$. Of course, letting $One = \lambda xy.1$, the "$\uparrow$" above is short for $(\mu y)One(x, y)$ that is in $\mathcal{P}$. Clearly,

- $\lambda\vec{x}_n.h(g(\vec{x}_n)) \in \mathcal{P}$ (substitution)

  and

- $P(\vec{x}_n) \equiv h(g(\vec{x}_n))\downarrow$, since $h(g(\vec{x}_n))\downarrow$ precisely when $g(\vec{x}_n) = 0$; hence $P(\vec{x}_n) \in \mathcal{P}_*$. □

The preceding corollary has a known to us analogue of for $\mathcal{PR}_*$ and $\mathcal{R}_*$. It provides, among other things, direct proofs for the facts $\mathcal{PR}_* \subseteq \mathcal{P}_*$ and $\mathcal{R}_* \subseteq \mathcal{P}_*$. For example, say, $Q(\vec{x}) \in \mathcal{PR}_*$. Then, for some $g \in \mathcal{PR}$, $Q(\vec{x}) \equiv g(\vec{x}) = 0$, for all $\vec{x}$. But $g \in \mathcal{P}$ as well, and we can invoke 0.1.4.

0.1.5 Corollary. (Graphs of Partial Recursive Functions) $\lambda\vec{x}_n.f(\vec{x}_n) \in \mathcal{P}$ *iff* $y = f(\vec{x}_n)$ *is semi-recursive.*

Proof. For the *only if*, let $f = \phi_i^{(n)}$. Then

$$y = f(\vec{x}_n) \equiv (\exists z)\big(T^{(n)}(i, \vec{x}_n, z) \wedge d(z) = y\big)$$

We conclude by 0.1.3. For the if part, let (again, 0.1.3)

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

To compute $f(\vec{x}_n)$—given $\vec{x}_n$—we enumerate all pairs $\langle z, y\rangle$ and stop at the "first", if any, that satisfies $Q(z, \vec{x}_n, y)$; we output y. Mathematically,

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

Clearly $f \in \mathcal{P}$. □

Pause. Why not argue the *if* part more simply, in view of 0.1.4? Let $g \in \mathcal{P}$ such that

$$y = f(\vec{x}_n) \equiv g(y, \vec{x}_n) = 0$$

Then $f(\vec{x}_n) = (\mu y)g(y, \vec{x}_n)$, for all $\vec{x}_n$, and thus $f \in \mathcal{P}$.◀

0.1.6 Remark. (Deciders and Verifiers) A computable relation $P(\vec{x}_n)$ is, by definition, one for which $\chi_P \in \mathcal{R}$; thus it has an associated URM M that *decides membership* of any $\vec{a}_n$ in P *both ways*: "yes" (output 0) if it is in; "no" (output 1) if it is not.

Thus this M is a *decider* for $P(\vec{x}_n)$.

A semi-computable relation $Q(\vec{x}_m)$, on the other hand, comes equipped only with a *verifier*, i.e., a URM N that verifies $\vec{a}_m \in Q$, *if true*, by virtue of halting on input $\vec{a}_m$.

A verifier gives no tangible information about the non membership cases, which cause it to enter a so-called "infinite loop" (it enters a *non terminating* computation).

While, *mathematically* speaking, $\vec{a}_m \notin Q$ is also "verified" by virtue of *looping forever* on input $\vec{a}_m$, *computationally speaking* this is *no verification at all* as we do *not* have a way of knowing whether N is looping forever as opposed to simply being awfully slow, planning perhaps to halt in a couple of trillion years (cf. *halting problem* 0.1.12).

In the algorithmic sense, a verifier (of a semi-computable set of m-tuples) *verifies only* the "yes" instances of questions such as "Is $\vec{a}_m \in Q$?"—hence its name. □

Thus, the output of a verifier for a semi-computable relation $Q(\vec{x})$, when it halts, is irrelevant. It has verified membership of its input to Q *simply by virtue of terminating its computation.*

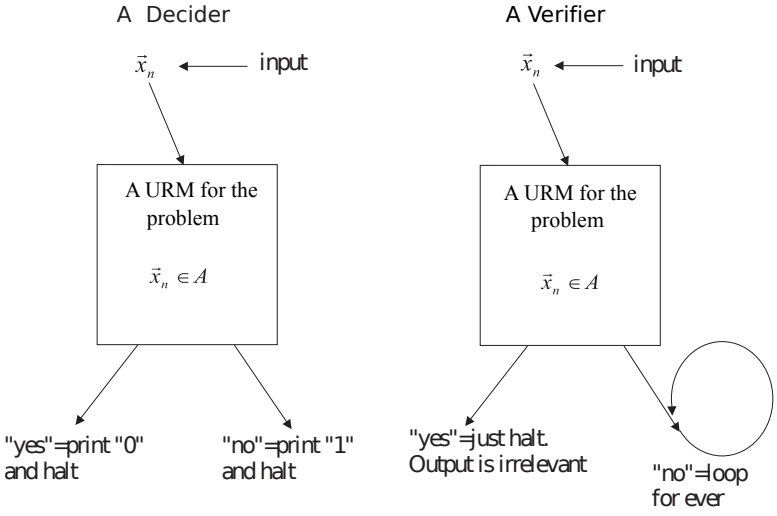

Figure 1: A decider and a verifier, pictorially, for handling the query, "$\vec{x}_n \in A$?", by a URM.

0.1.7 Definition. (Undecidable Problems) A *problem* is a *question* of the form $\vec{x} \in Q$. Synonymously, a question of the form $Q(\vec{x})$.

Thus, *a problem is a predicate.*

We say that a problem $Q(\vec{x})$ is *decidable* or (recursively)* *solvable*, iff there is a *decider* for it, which mathematically is expressed by "$Q(\vec{x}) \in \mathcal{R}_*$"—i.e., Q is recursive. In the opposite case we say that $Q(\vec{x})$ is *undecidable* or (recursively) *unsolvable.*

A problem $Q(\vec{x})$ is *semi-decidable* iff there is a a verifier for it, that is, iff $Q(\vec{x})$ is semi-computable. □

*The parenthetical qualifier is usually omitted.

Intuitively, we see that if we have a verifier for a relation $Q(\vec{x}_n)$ *and* also have a verifier for its *complement* (negation) $\neg Q(\vec{x}_n)$, then we can build a decider for $Q(\vec{x}_n)$: On input $\vec{a}_n$ we simply run *both* verifiers simultaneously. If the one for Q halts, then we print 0 and stop the computation. If, on the other hand, the one for $\neg Q$ halts, then we print 1 and stop. Of course, one or the other *will* halt, since one of $Q(\vec{a}_n)$ or $\neg Q(\vec{a}_n)$ *is* true!

This process computes $\chi_Q(\vec{a}_n)$. Put more mathematically,

0.1.8 Proposition. *If $Q(\vec{x}_n)$ and $\neg Q(\vec{x}_n)$ are in $\mathcal{P}_*$, then both are in $\mathcal{R}_*$.*

Proof. Let i and j be semi-indices of Q and $\neg Q$ respectively, that is (0.1.2),

$$Q(\vec{x}_n) \equiv (\exists z)T^{(n)}(i, \vec{x}_n, z)$$
$$\neg Q(\vec{x}_n) \equiv (\exists z)T^{(n)}(j, \vec{x}_n, z)$$

Define

$$g = \lambda \vec{x}_n.(\mu z)\big(T^{(n)}(i, \vec{x}_n, z) \vee T^{(n)}(j, \vec{x}_n, z)\big)$$

Intuitively, g implements (mathematically) the process in which we run the two verifiers simultaneously, (coded as) i and j, and look for one that halts, by looking for the smallest z that codes a computation of i or j as the case may be.

Trivially, $g \in \mathcal{P}$. Hence, $g \in \mathcal{R}$, since it is total (why?). We are done by noticing that $Q(\vec{x}_n) \equiv T^{(n)}(i, \vec{x}_n, g(\vec{x}_n))$. By closure properties of $\mathcal{R}_*$, $\neg Q(\vec{x}_n)$ is in $\mathcal{R}_*$, too. □

0.1.9 Remark. (Undecidable Problems and Uncomputable Functions Exist)

We can readily show, albeit in a somewhat intangible manner, that *undecidable problems* and therefore *uncomputable total functions* (their characteristic functions) exist.

This readily follows from a so-called *cardinality argument*: By Kleene's Normal Form theorem, we have only a countable set of partial recursive functions

$$\{\phi_i : i \in \mathbb{N}\} \tag{1}$$

Thus the *subset* of total (computable) 0-1-valued functions (and hence, decidable problems) is countable. However, the set of all total functions $f : \mathbb{N} \to \{0, 1\}$ is uncountable. So there must be many such functions that do not belong to the enumeration (1)! Each such function f not only provides an example of an uncomputable function, but being 0-1-valued provides an example of an undecidable problem, this one: $f(x) = 0$.

We called this an "intangible demonstration" of the existence of undecidable problems as it produced no specific meaningful problem that is undecidable. We remedy this below. □

0.1.10 Definition. (The Halting Problem) The *halting problem* has central significance in computability. It is the question whether "*program x will ever halt if it starts computing on input x*". That is, if we set $K = \{x : \phi_x(x)\downarrow\}$, then the halting problem is $x \in K$. We denote the complement of K by $\overline{K}$. □

0.1.11 Exercise. The halting problem $x \in K$ is semi-recursive.

Hint. The problem is "$\phi_x(x)\downarrow$". Now invoke the normal form theorem. □

0.1.12 Theorem. (Unsolvability of the Halting Problem) *The halting problem is undecidable.*

Proof. In view of the preceding exercise (and 0.1.8), it suffices to show that $\overline{K}$ is not semi-computable. Suppose instead that i is a semi-index of this set. Thus, $x \in \overline{K} \equiv (\exists z)T(i, x, z)$, or, making the part $x \in \overline{K}$—that is, $\phi_x(x)\uparrow$—explicit:

$$\neg(\exists z)T(x, x, z) \equiv (\exists z)T(i, x, z) \tag{1}$$

Substituting i into x in (1) we get a contradiction. □

0.1.13 Remark. (1) By 0.1.1 a set $S \subseteq \mathbb{N}$ is semi-recursive iff "it is a W_i", that is, for some i, $S = W_i$. The above proof says that "$\overline{K}$ is not a W_i". Is this surprising? Well, no!

This goes back to the Cantor diagonalization that shows that D ($\subseteq \mathbb{N}$), below,

$$D = \{x : x \notin S_x\}$$

is not an S_i (cf. our introductory MATH lectures), where each S_i is a subset of $\mathbb{N}$. Indeed,

$$x \in W_i \overset{0.1.1}{\equiv} \phi_i(x)\downarrow \equiv (\exists y)T(i, x, y)$$

hence $x \notin W_i \equiv \neg(\exists y)T(i, x, y)$ and, in particular, $i \notin W_i \equiv \neg(\exists y)T(i, i, y)$. But the right hand side says "$\phi_i(i)\uparrow$", that is, $i \in \overline{K}$. Thus

$$\overline{K} = \{x : x \notin W_x\}$$

and Cantor's diagonalization argument shows that "$\overline{K}$ is not a W_i". *So the proof of 0.1.12 was a well-concealed diagonalization argument!*

(2) Since $K \in \mathcal{P}_*$, we conclude that the inclusion shown in class, $\mathcal{R}_* \subseteq \mathcal{P}_*$, is proper, i.e., $\mathcal{R}_* \subset \mathcal{P}_*$.

(3) The characteristic function of K provides an *example* of a *total* uncomputable function.

(4) We saw an example of how to remove "points of non definition" from a function so that it *remains computable* after it has been extended to a total function: Cases of $\lambda xy.x^y$ and $\lambda xy.\lfloor x/y \rfloor$ from text and class. Can we *always* do that?

No. For example, the function $f = \lambda x.\phi_x(x) + 1$ *cannot* be extended to a *total* computable function. Of course, by the Normal Form Theorem, $f \in \mathcal{P}$,

since, for all x, $f(x) = d\big((\mu y)T(x,x,y)\big) + 1$. Here is why: Suppose that $g \in \mathcal{R}$ extends f. Thus, $g = \phi_i$ for some i. Let us look at $g(i)$: We have

$$g(i) \underset{\text{by } g=\phi_i}{=} \phi_i(i) \underset{\text{both sides defined}}{\neq} \phi_i(i)+1 \underset{\text{def. of } f}{=} f(i)$$

But since $f(i)\downarrow$, we also have $g(i) = f(i)$ as g extends f, a contradiction. □

Once we have built a class of functions or predicates, we next look at their closure properties.

0.1.14 Theorem. (Closure Properties of $\mathcal{P}_*$) *$\mathcal{P}_*$ is closed under $\vee$, $\wedge$, $(\exists y)_{<z}$, $(\exists y)$, and $(\forall y)_{<z}$. It is* not *closed under either $\neg$ or $(\forall y)$.*

Proof. Given semi-computable relations $P(\vec{x}_n)$, $Q(\vec{y}_m)$, and $R(y, \vec{u}_k)$ of semi-indices p, q, r, respectively. In each case we will express the relation we want to prove semi-computable as a strong projection (0.1.3):

$\vee$

$$\begin{aligned} P(\vec{x}_n) \vee Q(\vec{y}_m) &\equiv (\exists z) T^{(n)}(p, \vec{x}_n, z) \vee (\exists z) T^{(m)}(q, \vec{y}_m, z) \\ &\equiv (\exists z)\big(T^{(n)}(p, \vec{x}_n, z) \vee T^{(m)}(q, \vec{y}_m, z)\big) \end{aligned}$$

$\wedge$

$$\begin{aligned} P(\vec{x}_n) \wedge Q(\vec{y}_m) &\equiv (\exists z) T^{(n)}(p, \vec{x}_n, z) \wedge (\exists z) T^{(m)}(q, \vec{y}_m, z) \\ &\equiv (\exists w)\big((\exists z)_{<w} T^{(n)}(p, \vec{x}_n, z) \wedge (\exists z)_{<w} T^{(m)}(q, \vec{y}_m, z)\big) \end{aligned}$$

$(\exists y)_{<z}$

$$\begin{aligned} (\exists y)_{<z} R(y, \vec{u}_k) &\equiv (\exists y)_{<z} (\exists w) T^{(k+1)}(r, y, \vec{u}_k, w) \\ &\equiv (\exists w)(\exists y)_{<z} T^{(k+1)}(r, y, \vec{u}_k, w) \end{aligned}$$

$(\exists y)$

$$\begin{aligned} (\exists y) R(y, \vec{u}_k) &\equiv (\exists y)(\exists w) T^{(k+1)}(r, y, \vec{u}_k, w) \\ &\equiv (\exists z)(\exists y)_{<z} (\exists w)_{<z} T^{(k+1)}(r, y, \vec{u}_k, w) \end{aligned}$$

$(\forall y)_{<z}$

$$\begin{aligned} (\forall y)_{<z} R(y, \vec{u}_k) &\equiv (\forall y)_{<z} (\exists w) T^{(k+1)}(r, y, \vec{u}_k, w) \\ &\equiv (\exists v)(\forall y)_{<z} (\exists w)_{<v} T^{(k+1)}(r, y, \vec{u}_k, w) \end{aligned}$$

As for possible closure under $\neg$ and $\forall y$, K provides a counterexample to $\neg$: $K \in \mathcal{P}_*$ (0.1.11) but $\overline{K} \notin \mathcal{P}_*$ (0.1.12). Closure under $\forall y$ is also untenable as $\neg T(x, x, y)$ provides a counterexample: Being primitive recursive, it is in $\mathcal{P}_*$. However, $(\forall y)\neg T(x, x, y)$ is not, since this is $\neg(\exists y) T(x, x, y)$—that is, $x \in \overline{K}$. □

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

Proof.

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

By 0.1.5 and 0.1.14, the right hand side, and hence the left hand side, of $\equiv$ is semi-recursive. □

0.1.16 Example. This is our first example of a *reduction argument*, a trivial one. We introduce a generalization, K_0, of the halting set K, by

$$K_0 \stackrel{\text{Def}}{=} \{(x,y) : \phi_x(y)\downarrow\}$$

We show that the problem $(x,y) \in K_0$ is undecidable, that is,

$$K_0 \notin \mathcal{R}_* \tag{1}$$

Suppose that (1) is false. Then the characteristic function, $\lambda xy.\chi_{K_0}(x,y)$ of K_0 is in $\mathcal{R}$. But then so is the function $f = \lambda x.\chi_{K_0}(x,x)$ obtained from χ_{K_0} by identification of variables. However, f is the characteristic function of the halting set, and we just have shown that the halting problem is decidable!

This contradiction shows that (1) is correct, after all.

We have just witnessed an instance of an argument that went like this: *If I have an algorithm that solves problem B,*[†] *then I know how to build another algorithm that uses the one for B and solves problem A.*[‡]

That is, we *reduced problem A to problem B* (this makes A "more decidable" than B; and makes B "more undecidable" than A).


This reduction shows that *if we know that A is undecidable, then so must be B.*


We will encounter many more reduction arguments later. □

[†]Here $(x,y) \in K_0$.
[‡]Here $x \in K$.

0.1.17 Example. (A Very Hard Problem) The *equivalence problem* is: *given two programs, decide if they compute the same function or not.*

A "program" here can be any *finite way* of describing a function. This finite way could be an actual program, such as a URM or a loop program. Or it could be a derivation, say, within $\mathcal{PR}$ or $\mathcal{P}$, which defines a function.

To fix ideas, let us focus attention on primitive recursive functions, *finitely defined via loop programs.*

We ask: Is the problem of determining whether *two such functions* are equal decidable?

Well, if it is, then in particular so will be the special case of determining whether $\lambda y.1$ and $\lambda y.\chi_T(x,x,y)$—where T is the Kleene predicate—are the same function or not, for any given x. The reader may readily imagine—due to the primitive recursiveness of both functions—that they are given by loop programs.

The question, mathematically, is $(\forall y)(1 = \chi_T(x,x,y))$. In terms of T this says $(\forall y)\neg T(x,x,y)$, or $\neg(\exists y)T(x,x,y)$.

We recognize the last expression as $x \in \overline{K}$, which we know that is *not semi-computable* (0.1.12), let alone recursive!

Pause. Why "let alone"?◀

Thus the equivalence problem of primitive recursive functions is incredibly hard: There is not even a verifier for it! □

March ??

0.1.18 Remark. (Computably Enumerable Sets) There is an interesting characterization of *non-empty* semi-computable sets that is found in all introductions to the theory of computation. These sets are precisely those that can be "enumerated effectively" or "computably", that is, we can prove that

A non-empty set $S \subseteq \mathbb{N}$ is semi-computable iff it is the range *of some $f \in \mathcal{PR}$.*

The enumeration is *not* required to be 1-1, so there may be repetitions. Notice that since the enumerating function is total, there will *necessarily* be repetitions in the case when S is finite.

What is the intuition for this? Well,

(1) Assume first that we have an algorithmic enumeration of all the members of S. Here is then how to verify (semi-decide) the question $x \in S$: Given x, start the algorithmic enumeration and keep an eye on what it "prints". If and when x is printed, then stop. We have verified $x \in S$. What if $x \notin S$? Well, then it will never be printed by the enumeration and we will never stop our process.

(2) Conversely, assume that we have a verifier, M (a URM), for $x \in S$. We write a new program N that behaves as follows: It *systematically generates* all pairs of numbers (x, y), one at a time.

For example, one can enumerate all numbers

$$0, 1, 2, 3, \ldots, z, \ldots$$

in turn, and, for each z generated, one can generate the pair $\big((z)_0, (z)_1\big)$.

For each pair generated, N checks whether M, on input x, *halts within y computation steps.*[§] If so, x is printed (as it clearly belongs to S).

Pause. Why make it so complicated and not instead enumerate all numbers x in turn, and if M halts on x, then print x?◀

The technique in (2) above is called *dovetailing* several computations (of M) *for several inputs* "at once". Well, strictly speaking, not "at once". The method implements, indeed *sequentially* simulates, a "poor person's *parallelism*", because, in essence, it simulates the *in parallel* examination of several questions of the type "does M halt on x?".

[§]Think of a "step" as the passage from one ID to the next.

The essential feature of parallelism is not the temporal simultaneity of testing the questions "does M halt on x?" for various x, but rather the fact that if an input $x = a$ causes M to run forever, this *does not affect, nor block, the testing of other inputs for which M halts.* A true parallel "environment" allocates one process to each input x. On the other hand, dovetailing captures this *key property of parallelism* and does so with *a single computation process* (or "single processor")!

The simulation of parallelism is effected by allowing to each question gradually more and more *time* (number of steps) to reach an answer. Notice that since there are infinitely many pairs (a, y) with first component a, if M ever halts on a—say, using $y = b$ computation steps—then this fact will be eventually verified in the process (2): It will happen precisely when we will be testing the pair (a, b).

An intuitively more immediate *rearrangement of the dovetailing process* of (2), which demonstrates the sense in which dovetailing is approaching true parallelism "in the limit", is captured by the matrix below:

$$\begin{array}{l} 0;\ 1 \\ 0{,}1;\ 2 \\ 0{,}1{,}2;\ 3 \\ 0{,}1{,}2{,}3;\ 4 \\ \vdots \\ 0{,}1{,}2{,}3{,}\ldots{,}i;\ i+1 \\ \vdots \end{array}$$

The number at the far right in each row is the number of steps that we let M run. The other numbers in each row are the inputs we test *for said number of steps.* In the "limit", it is as if we are testing all inputs "simultaneously": input 0 for one step; inputs 0 and 1 for two steps; inputs 0, 1 and 2 for three steps; $\ldots$, inputs $0, 1, \ldots, i$ for $i + 1$ steps; and so on. □

Mathematically, we repeat the above informal argument, (1) and (2), in 0.1.19 below to prove the italicized statement at the beginning of the previous remark. Central in our preceding discussion was the concept of "step". But what *is* a step mathematically? We take as "step" to be the entire computation, coded as y in the Kleene predicate $T(i, x, y)$. That is, for any ϕ_i, its *step-counting* or *complexity function* is

$$\Phi_i = \lambda x.(\mu y)T(i, x, y) \qquad (*)$$

This is reasonable, since the computation y is a strictly increasing function of how many ID-to-ID "real steps" took place in the (terminating) computation.

In fact, [Blu67] takes as the key, indeed *defining*, properties of the concept of complexity of ϕ_i the following two:

(I) $\phi_i(x)\downarrow$ iff $\Phi_i(x)\downarrow$; that is, the program i halts on input x iff a complexity of computation can be assigned for said input.

(II) $\Phi_i(x) \leq y$ is *recursive*; that is, we can *decide* whether machine i halts within y (i.e., in $\leq y$) "steps".

[Blu67] takes (I)–(II) as the *axioms* for complexity theory, that is, without specifying Φ explicitly. Many concrete choices of Φ that satisfy the axioms are possible. By the way, for our chosen Φ in $(*)$, (I) is trivially obtained directly from the normal form theorem. As for (II), $\Phi_i(x) \leq y \equiv (\exists z)_{\leq y} T(i, x, z)$ which is more than recursive: primitive recursive.

On the other hand $y < \Phi_i(x) \equiv \neg\Phi_i(y) \leq y$; also in $\mathcal{PR}_*$.

It is important to observe that we bypass (I), above, when we assess

$$\Phi_i(x) \begin{matrix} \leq \\ = \\ \geq \end{matrix} y$$

We do *not* compute $\Phi_i(x)$ (which may diverge!) to figure out the answer.

0.1.19 Theorem. *A non-empty set $S \subseteq \mathbb{N}$ is semi-computable iff it is the* range *of some $f \in \mathcal{PR}$.*

Proof. For the part (1), let f be primitive recursive such that $\text{ran}(f) = S$. That is,

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

Given that $f(x) = y$ is in $\mathcal{PR}_*$, $y \in S$ is semi-computable by the projection theorem (0.1.3).

For the dovetailing part, (2) of 0.1.18, assume that the non-empty S is semi-computable. Let i be a semi-index for S, thus,

$$x \in S \equiv (\exists y) T(i, x, y) \qquad (*)$$

for all x. Following directly on the idea in (2), with the concept of "step" made mathematically precise in the preceding remarks, we define the enumerating function by:

$$f(z) = \begin{cases} (z)_0 & \text{if } T(i, (z)_0, (z)_1) \\ a & \text{otherwise} \end{cases}$$

where "a" is some fixed member of S that we keep outputting every time the condition "$T(i, (z)_0, (z)_1)$" fails,[¶] ensuring that f is total. Of course f is primitive recursive.

Is it true that $\text{ran}(f) = S$?

Indeed, as $\text{ran}(f)$ contains only numbers of the form $(z)_0$ such that $T(i, (z)_0, (z)_1)$ holds, it is immediate by $(*)$ that $\text{ran}(f) \subseteq S$. Conversely, let $x \in S$ and let b be a value of y that makes $(*)$ true. But then $f([x, b]) = x$, so $x \in \text{ran}(f)$. □

The above result justifies the following nomenclature:

[¶]Condition failed: Either because we did not let the computation $\phi_i(x)$ to go on long enough, or no terminating computation exists.

0.1.20 Definition. A set $S \subseteq \mathbb{N}$ is called *computably enumerable* (c.e.) or *recursively enumerable* (r.e.) iff it is either empty, or is the range of a primitive recursive function. □

There is no loss of generality in presenting the above definition for subsets of $\mathbb{N}$ since via coding $\langle \ldots \rangle$ it can be trivially and naturally extended to sets of n-tuples for $n > 1$. A set $S \subseteq \mathbb{N}^n$ is c.e. iff there is a primitive recursive f such that $\text{ran}(f) = \{\langle \vec{x} \rangle : S(\vec{x})\}$.

0.1.21 Corollary. *A non-empty set $S \subseteq \mathbb{N}$ is semi-recursive iff it is c.e. (r.e.)*

0.1.22 Corollary. *A non-empty set $S \subseteq \mathbb{N}^n$ is semi-recursive iff it is c.e. (r.e.)*

Proof. The *if* is straightforward, while the *only if* is a direct adaptation of the proof of 0.1.19: Let

$$\vec{x}_n \in S \equiv (\exists y) T^{(n)}(i, \vec{x}_n, y) \qquad (**)$$

for all x. The enumerator f is given by

$$f(z) = \begin{cases} \langle (z)_0, \ldots, (z)_{n-1} \rangle & \text{if } T^{(n)}(i, (z)_0, \ldots, (z)_{n-1}, (z)_n) \\ \langle \vec{a}_n \rangle & \text{otherwise} \end{cases}$$

where "$\vec{a}_n$" is some fixed member of S. □

0.1.23 Corollary. *A set $S \subseteq \mathbb{N}^n$ is semi-recursive iff it the range of an $f \in \mathcal{P}$.*

Proof. The *only if* is proved as above, where we just drop the "$\langle \vec{a}_n \rangle$ otherwise". For the *if*, suppose that

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

By 0.1.5 and 0.1.14, the above yields $S \in \mathcal{P}_*$. □

0.1.24 Example. (Another Very Hard Problem) The set $\mathscr{R} = \{x : \phi_x \in \mathcal{R}\}$—which trivially is the same as $\{x : \phi_x \text{ is total}\}$—is very important in computability. One certainly wants to know whether or not we can "tell" if a program x computes a total function. We can tell in one of two ways: We can fully (algorithmically) *decide* the question $x \in \mathscr{R}$, or we can just *verify* it when true. Which one is it here?

Neither. $\mathscr{R}$ is not semi-recursive, hence nor is it recursive. In that sense this is another very hard—and very meaningful—problem of which we cannot even verify the positive instances.

We prove the non semi-recursiveness by proving that $\mathscr{R}$ is not c.e. using diagonalization. So, by way of contradiction, let $f \in \mathcal{PR}$ be such that $\mathscr{R} = \text{ran}(f)$. This means that $\{\phi_{f(x)} : x \in \mathbb{N}\}$ is the set of all total computable functions of one variable. Consider the function $d = \lambda x.\phi_{f(x)}(x) + 1$.

By the preceding remark, and composition with the successor function, $d \in \mathcal{R}$. Thus, for some i,

$$d = \phi_{f(i)} \qquad (1)$$

since $\mathscr{R}$ is the set of *all* programs that compute total 1-argument functions, thus a program m for d must be an $f(i)$.

What do we know of $\phi_{f(i)}(i)$? Well,

$$\phi_{f(i)}(i) \overset{\text{By (1)}}{=} d(i) \overset{\text{By Def. of } d}{=} \phi_{f(i)}(i) + 1$$

A contradiction, since all sides of $=$ are defined. So no such f exists, and $\mathscr{R}$ is not c.e. □

0.1.25 Exercise. (Definition by Positive Cases) Consider a set of *mutually exclusive* relations $R_i(\vec{x})$, $i = 1, \ldots, n$, that is, $R_i(\vec{x}) \wedge R_j(\vec{x})$ is false for each $\vec{x}$ as long as $i \neq j$.

Then we can define a function f *by positive cases* R_i from given functions f_j by the requirement (for all $\vec{x}$) given below:

$$f(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } R_1(\vec{x}) \\ f_2(\vec{x}) & \text{if } R_2(\vec{x}) \\ \ldots & \ldots \\ f_n(\vec{x}) & \text{if } R_n(\vec{x}) \\ \uparrow & \text{otherwise} \end{cases}$$

Prove that if each f_i is in $\mathcal{P}$ and each of the $R_i(\vec{x})$ is in $\mathcal{P}_*$, then $f \in \mathcal{P}$.

Hint. Use 0.1.5 along with closure properties of $\mathcal{P}_*$ relations to examine $y = f(\vec{x})$. □

A semi-recursive predicate is "positive" having the form $(\exists y)Q(y, \vec{x})$ for some recursive Q (0.1.3). It is also known as a Σ_1 predicate.

It is important to note about the last case in the definition:

(1) The *otherwise* condition, is the negation of a *positive* predicate, namely, of the semi-recursive $R_1 \vee \cdots \vee R_n$. A "negative" predicate such as this negation has the form $(\forall y)R(y, \vec{x})$, for some recursive R, since it is the negation of one of the form $(\exists y)Q(y, \vec{x})$, for some recursive Q. Such negative predicates are also called Π_1 predicates.

(2) Note that the "output" in the last case is $\uparrow$. This, intuitively, is as much as is expected in general, given that, for example, the "otherwise" of some positive cases, such as $x \in K$, are not even semi-recursive so that the obvious "program" for the function f will enter into an infinite loop when pondering the condition "otherwise".

Bibliography

[Blu67] E. Blum, *A machine-independent theory of the complexity of recursive functions*, ACM **14** (1967), 322–336.

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