Chapter 5

(Un)Computability via Church's Thesis

We noted that computability is the part of logic that gives a mathematically precise formulation to the concepts *algorithm*, *mechanical procedure*, *computation*, and *calculable* or *computable* function (or relation), with a view of, on one hand, being able to mathematically study "mechanical procedures", to determine which tasks or problems admit such procedures and which do not —and to understand the "why"!— and, on the other hand, to classify functions and relations into two groups, those that are computable and those that are not.

Powerful tools have been developed in such a theory of "mechanical procedures" —for some of which the reader is expected to become a competent user— to prove that many tasks, indeed *uncountably*[†] infinitely many, do not admit mechanical procedures. We will see that one such problem is that of "program correctness": To *determine* whether an *arbitrary program* —say, written in C— is *faithful to its design specifications* for *all inputs*;[‡] for short it is "correct".

The advent of computability was strongly motivated, in the 1930s, by Hilbert's program, in particular by his belief that the *Entscheidungsproblem*, or *decision problem*, for axiomatic theories, that is, the problem "Is this formula a theorem of that theory?" was solvable by a *mechanical procedure* —dependent on the particular theory— that was waiting to be discovered.

Now, since antiquity, mathematicians have invented "mechanical procedures", e.g., Euclid's algorithm for the "greatest common divisor",[§] and had no problem recognising such procedures when they encountered them.

Thus, to show that a problem admits a mechanical procedure solution the

[†]In Cantor's sense of uncountable sets such as the set of the reals vs. countable sets such as the set of the natural numbers. Cantor explained the precise reason why the former set is "more infinite" than the latter.

[‡]Clearly, *not* by running the program on *all* inputs! We will not live long enough to see the answer!

[§]That is, the largest positive integer that is a common divisor of two given integers.

idea is straightforward: just *find* one.[†] This is a "programming" problem, and can be handled with some patience and ingenuity —in principle.

But how can I be *sure* that a mechanical procedure for a particular problem does *not exist*? Surely we cannot propose to try each one from the set of infinitely many mechanical procedures on our problem until we verify that none of them works?

Pause. Can you convince yourself that there are infinitely many syntactically correct, say, C programs? ◄

To prove the negation of an existential statement such as this you need a *mathematical formulation* of what a "mechanical procedure" *precisely is* and develop and exploit the mathematical properties of the set of *all such procedures* to prove that no member of that set can possibly solve our problem, or that any procedure that solves the problem cannot be in our set, contradicting the qualifier "all".

The above paragraph will make a lot of sense later in this volume.

Intensive activity by many pioneers of computability (Post [Pos36, Pos44], Kleene [Kle43], Church [Chu36b], Turing [Tur37], Markov [Mar60]) led in the 1930s to several alternative formulations of *computable* function and relation, each purporting to mathematically *capture* the concepts *algorithm*, *mechanical procedure*, and *computable function*. All these formulations were quickly proved to be pairwise equivalent; that is, the calculable functions admitted by any one of these formulations were the same as those that were admitted by any other. This led Alonzo Church to formulate his conjecture, widely known as "Church's Thesis", that any *intuitively* computable function is also computable within any of these mathematical frameworks of computability.[‡]

Incidentally, Church proved ([Chu36a, Chu36b]) that Hilbert's *Entscheidungsproblem* admits no solution by functions that are calculable within any of the known equivalent mathematical frameworks of computability. Thus, if

[†]... and prove that it does so!

[‡]I should be clear that even though this "thesis" has the flavour of a "completeness *theorem*" in the realm of computability, it is not.

In logic a *mathematical* definition for the intuitive (experiential) concept of *validity* or *universal truth* is given: a formula is universally true (in a technical sense) iff it is true in *all interpretations* ("interpretation" is also a technical, mathematical term). Gödel's *Completeness theorem* then states that every universally true formula of logic has a syntactic (also called *formal*; depending only on form) *proof* —which is a *finite syntactic* object— without using any mathematical axioms.

But we have no *mathematical* definition for the intuitive (experiential) concept of "computable" function *a priori* —we are searching for one! Thus, the best we can do here is to *speculate* about the above mentioned *equivalent mathematical* formulations of "computable" function —via finite programs in certain (essentially) programming formalisms— that each (fully) *captures* the intuitive notion of computable function.

In other words, Church's Thesis is an empirically formed belief rather than a provable result. It is not surprising that some researchers in this field, for examp, Péter [Ṕ67] and Kalmár [Kal57], pointed out that it is conceivable that the *intuitive* concept of calculability may in the future be extended to exceed the power of the various mathematical models of computation that we currently know.

we accept his "Thesis", the Entscheidungsproblem admits no algorithmic solution, period!

The eventual introduction of computers further fuelled the study of and research on the various mathematical frameworks of computation, "models of computation" as we often say, and "computability" is nowadays a vibrant and very extensive field.

5.1 A leap of faith: Church's thesis

The aim of Computability is to *mathematically capture* (for examp, via URMs) the *informal* notions of "algorithm" and "computable function" (or "computable relation").

Several mathematical models of computation, that were very different in their syntactic details and semantics, have been proposed in the 1930s by several people (Post, Church, Kleene, Turing), and, more recently, by Shepherdson and Sturgis ([SS63]).

They were all *proved to compute exactly the same number theoretic functions* —that is, all those functions in the set of the *partial recursive functions* $\mathcal{P}$ introduced in our earlier Notes.[†]

This "empirical" evidence prompted Church to state his *belief*, known as "*Church's Thesis*", that

> Every *informal* algorithm (pseudo-program) that we propose for the computation of a function can be implemented (*made mathematically precise*, in other words) in each of the known mathematical models of computation. In particular, *it can be "programmed" as a URM.*

We note that at the present state of our understanding of the concept of "algorithm" or "algorithmic process", *there is no known way* to define —via a pseudo-program of sorts— an "intuitively computable" function on the natural numbers, *which is outside of* $\mathcal{P}$.[‡]

Thus, as far as we know, $\mathcal{P}$ appears to *formalise* the *largest* —i.e., most inclusive— set of "intuitively computable" functions (on the natural numbers) known.

[†] The various models, and the gory details of why they all do *precisely* the same job, can be found in [Tou84].

[‡] In the so-called relativised computability (with partial oracles) —essentially allowing infinite-size inputs such as functions on the natural numbers— Church's Thesis fails [Tou86]: an example of an intuitively computable function that is not (mathematically) computable is one that compares the lengths of two computations. The reason is that said function is *non monotone* with respect to the oracle argument, while the "standard" theories of computability with partial oracles, e.g., [Dav58a, Mos69], compute only monotone functions. [Tou86] introduced a mathematical model of non monotone computability where the aforementioned counterexample to Church's Thesis above does not apply.

Church's Thesis is not a theorem. It cannot be, as it "connects" precise mathematical objects (URM, $\mathcal{P}$) with imprecise *informal* ones ("algorithm", "computable function").

However, if used, it provides the operational convenience and pedagogical advantage of concentrating on the high level of detail of why a program does what we say it does —or why a mathematical definition produces a computable function— without having to push too many symbols around in the process.

Another side-effect used to its fullest advantage (e.g., in [Rog67], an advanced book) is that, if we take the leap of faith and rely on Church's Thesis, then we present shorter, more comprehensible arguments —we save space *and* time of exposition.[†]

Since we *are* more interested in the essence of things in these Notes, and less in detail, we will heavily rely on Church's Thesis —to which we will refer, for short, as "CT"— to justify that various constructions we jot down yield computable functions.

In the literature, Rogers, as noted, heavily relies on CT. On the other hand, [Dav58a, Tou84, Tou12] never use CT, and give all the necessary constructions (implementations) in their full gory details —*this is the price to pay, if you avoid CT.*

Here is the template of **how** to use CT:

- We **completely** present —that is, no essential detail is missing— an algorithm in *pseudo-code.*

  ►BTW, "pseudo-code" does not mean "sloppy-code"!◄

- We then say: By CT, there is a URM that implements our algorithm. Hence the function that our pseudo code computes is in $\mathcal{P}$.

5.2 The effective list of all URMs

For ease of reference we repeat some introductory material from Section 1.1. The new material in this section starts with Remark 5.2.2 below.

We recall from the definition of URM programs —introduced in Section 1.1 and in particular in 1.1.1— that these programs are *strings* over a finite alphabet A:

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

[†]If you are ever in doubt about the legitimacy of a piece of "high-level pseudo code", then you ought to try to implement the pseudo program in detail, as a URM, or, at least, as a "real" C-program or equivalent! E.g., is the instruction "IF the present program outputs 0 for all inputs, GOTO label L ELSE GOTO label R" legitimate? That is, is this instruction a finitely describable "macro" that can be built using the URM instructions from 1.1.1? *An example of a legitimate macro* is "GOTO L". It abbreviates "IF $\mathbf{x} = 0$ GOTO L ELSE GOTO L".

Just like any other high level programming language, URM manipulates the contents of *variables*. *All variables are of natural number type.*

X and 1 finitely generate the variables of the URM programs as

$$X, X1, X11, \ldots X1^n, \ldots \tag{2}$$

where

$$1^n \stackrel{Def}{=} \overbrace{1\ldots1}^{n\ 1s}, \text{ where } 1^0 \stackrel{Def}{=} \lambda, \text{ the empty string}$$

while the symbols $0, 1, 2, \ldots, 9$ finitely generate natural *number constants* (in decimal notation) that we generically denote by a, b, c, with or without subscripts and primes, and *instruction labels* that we generically denote by L, R, P, with or without subscripts and primes.

As is customary for the sake of convenience, we will also utilise the bold face lower case letters $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{u}, \mathbf{v}, \mathbf{w}$, with or without subscripts or primes as *meta* names that stand for unspecified strings of the type $X1^n$ in most of our discussions of the URM, and in examps of specific examp programs (where yet more, convenient metanotations for variables may be employed).

We have defined that

5.2.1 Definition. (URM Programs) A *URM program* is a *finite (ordered) sequence* of instructions (or commands) of the following five types:

$$\begin{aligned}
L &: \ \mathbf{x} \leftarrow a \\
L &: \ \mathbf{x} \leftarrow \mathbf{x} + 1 \\
L &: \ \mathbf{x} \leftarrow \mathbf{x} \dot{-} 1 \\
L &: \ \mathbf{stop} \\
L &: \ \mathbf{if}\ \mathbf{x} = 0\ \mathbf{goto}\ M\ \mathbf{else\ goto}\ R
\end{aligned} \tag{3}$$

where L, M, R, a, written in decimal notation, are in $\mathbb{N}$, and $\mathbf{x}$ is some variable. We call instructions of the last type *if-statements*.

Any two consecutive instructions in a syntactically correct URM program are separated ("glued") by the ¶ symbol that serves as an instruction separator.

We chose ¶, the "hard return" symbol, for the role of instruction separator in order to be consistent with the expositional practise of *writing programs vertically*, one instruction per line. Thus, as in ordinary text, ¶ is invisible in the programs that we will write in these notes, but causes us to write the next instruction on the next line.

Each instruction in a URM program *must be numbered* by its *position number*, L, in the program, where ":" separates the position number from the instruction. We call these numbers *labels*. Thus, the label of the first instruction must always be "1". The instruction **stop** must *occur only once* in a program, as the **last instruction**. It is syntactically illegal for the *if-statement* L : **if** $\mathbf{x} = 0$ **goto** M **else goto** R to refer to labels M and R that are not actually labels that occur in the program where the *if-statement* appears. □

5.2.2 Remark. It is obvious from 1.1.1 that we can algorithmically check the syntactic correctness of a URM program. Further, if we assign a number to each alphabet symbol as in the matrix below

$$\begin{array}{cccccccccccccccccccc} \leftarrow & + & \dot{-} & : & X & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \mathbf{if} & \mathbf{else} & \mathbf{goto} & \mathbf{stop} & \P \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \end{array}$$

then we can view each URM as a string of symbols from the (or, "over the", as we say) set

$$\{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20\}$$

which we interpret as a number base-21. Conversely, any number, which when viewed base-21 has no zero digits, represents a string over A.†

Therefore, a question like "is the predicate $URM(z)$ —that states 'z is a *string* over A that parses correctly as a URM'— decidable?" can be dealt with by our computability theory despite the fact that our theory deals only with number theoretic predicates and functions.

In fact, by CT and the opening sentence in this remark we can answer, "yes", viewing the string z as a number, written base-21, and the predicate $URM(z)$ as a number-theoretic predicate.‡ $\square$

Now we can show that we can algorithmically, or as we also say, *effectively* enumerate all URMs.

5.2.3 Theorem. *The set of all URMs can be effectively enumerated, in the sense that there is a* total computable *function E of one variable such that*

- *For each z, we have $URM(E(z))$*
- *If $URM(w)$ is true, then for some z, $E(z) = w$*

Proof. Consider the pseudo program below whose computation results in a *non-ending* enumeration of all URMs in a "standard" listing (sequence) $List_2$:

(A) We can algorithmically build the list $List_1$ of *all* strings over A: *List by increasing length* and in each length-group enumerate *in lexicographic order*.

(B) Simultaneously to building $List_1$, build $List_2$ as follows: For every string w placed in $List_1$, copy it into $List_2$ **iff** $URM(w)$ is true (cf. Remark 5.2.2).

A modification of the above pseudo-program can ensure that $E(z)$ is the z-th URM in the enumeration for all $z \in \mathbb{N}$:

proc $E(z)$

†If we allow digit zero then we lose the 1-1 correspondence between number "codes" and strings. For examp, if we assigned code 0 to $\leftarrow$ then the strings $\leftarrow$, $\leftarrow\leftarrow$, $\leftarrow\leftarrow\leftarrow$, etc., *all* have numerical code 0. So 0 does not decode uniquely to a string under these circumstances.

‡I.e., a subset of $\mathbb{N}$ in the one-variable case.

(A′) **Comment**. Given $z \geq 0$ as argument. The procedure $E(z)$ will output the z-th URM from $List_2$.

(B′) $w \leftarrow 0$; **Comment**. Keeps track of how many strings u we placed in $List_2$.

(C′) Algorithmically build the list $List_1$ as described above;

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

For every string u placed in $List_1$

if $URM(u)$ is **true**, then **do**

{

- copy u into $List_2$;
- **if** $w = z + 1$, **then Return**(u) **else** $w \leftarrow w + 1$;

}

By CT, the above procedure defines a (total) computable function $\lambda z.E(z)$ such that $E(z)$ is the z-th URM. Why total? Because there are infinitely many URMs, thus, for every z there *will* be a z-th URM to be listed. □

5.2.4 Corollary. *The set of all partial computable functions* of one variable *can be effectively enumerated using their URMs as proxies, that is, we enumerate them as* $N^{\mathbf{x}}_{\mathbf{x}'}$*, where* N *runs over the list of all URMs and the* $\mathbf{x}$ *and* $\mathbf{x}'$ *run over all choices of pairs of of input-output variables from among the variables of* N*.*

Every computable function f is some $N^{\mathbf{x}}_{\mathbf{x}'}$ and thus occupies **at least one** position i in the listing. Why not exactly one? Because for every N we can add at its end, but *before* the **stop** instruction, one or more instructions $\mathbf{z} \leftarrow 1$ where $\mathbf{z}$ is *fresh* (a new variable). Any one of the modified N, call it N', satisfies $N^{\mathbf{x}}_{\mathbf{x}'} = N'^{\mathbf{x}}_{\mathbf{x}'}$. Thus every function $f \in \mathcal{P}$ has infinitely many programs that compute it. This entails that the enumeration of 5.2.4 is with infinitely many repetitions, for each unary $f \in \mathcal{P}$.

Proof. This has the status of a corollary since the proof is an easy modification of the theorem's proof. The obvious idea is to enumerate the $N^{\mathbf{x}}_{\mathbf{x}'}$ by using $List_2$ as source, and for each N generated, to list *all* strings $N00\mathbf{x}00\mathbf{x}'$ —which stand for $N^{\mathbf{x}}_{\mathbf{x}'}$, for all $\mathbf{x}$ and $\mathbf{x}'$ in N— *lexicographically* with respect to the "tail" $00\mathbf{x}00\mathbf{x}'$. Incidentally, for each N we have finitely many such strings.

Thus, if the enumerating function is called F, we have the pseudo-program below that computes $F(z)$ for $z \geq 0$. $F(z)$ is the z-th $N^{\mathbf{x}}_{\mathbf{x}'}$ in the listing of all computable partial functions of one argument.

proc $F(z)$

(A′) **Comment**. Given $z \geq 0$ as argument. The procedure $F(z)$ will output the z-th unary partial computable function $N^{\mathbf{x}}_{\mathbf{x}'}$ —as $N00\mathbf{x}00\mathbf{x}'$— placed in $List_3$.

(B′) $w \leftarrow 0$; **Comment**. Keeps track of how many strings u we placed in $List_3$.

(C′) Algorithmically build the list $List_1$ as described earlier;

(D′) Simultaneously to building $List_1$ also build $List_3$ as follows:

For every string u placed in $List_1$

if $URM(u)$ is **true**, then **do**

$\Big\{$

- for each pair of variables $\mathbf{x}$ and $\mathbf{x}'$ in the URM u, **do**

  {

  - add the string $u00\mathbf{x}00\mathbf{x}'$ in $List_3$, arranging each of these additions in lexicographic order of the strings $00\mathbf{x}00\mathbf{x}'$.
  - **if** $w = z + 1$, **then Return**$(u00\mathbf{x}00\mathbf{x}')$ **else** $w \leftarrow w + 1$;

  }

$\Big\}$

By CT, the above procedure defines a (total) computable function $\lambda z.F(z)$ such that $F(z)$ is the z-th unary computable partial function. □

5.2.5 Corollary. *The set of all partial computable functions* of n variables *can be effectively enumerated using their URMs as proxies, that is, we enumerate them as* $N^{\vec{\mathbf{x}}_n}_{\mathbf{x}'}$*, where* N *runs over the list of all URMs and the* $\vec{\mathbf{x}}_n$ *and* $\mathbf{x}'$ *represent all pairs of choice of input (n-vector)-output variables from among the variables of* N*.*

Proof. Trivial modification of the proof of 5.2.4. Here we enumerate, for each URM N that we find (in our URM enumeration), all functions $N^{\vec{\mathbf{x}}_n}_{\mathbf{x}'}$, for all choices of $\vec{\mathbf{x}}_n$ and $\mathbf{x}'$ in N. The symbol $N^{\vec{\mathbf{x}}_n}_{\mathbf{x}'}$ is rendered in one dimension as the string $N00\mathbf{x}_1 0\mathbf{x}_2 0 \ldots 0\mathbf{x}_n 00\mathbf{x}'$. □

5.3 The universal function theorem

The following is an extremely useful tool in the development of computability theory. It is Kleene's "*universal function theorem*".

5.3.1 Theorem. (Universal function theorem) *There is a* partial computable *two-variable function* h *with this property: For any one-variable function* $f \in \mathcal{P}$*, there is a number* $i \in \mathbb{N}$ *such that* $h(i, x) = f(x)$ *for all* x*. Equivalently,* $\lambda x.h(i, x) = f$*.*

Recall (1.1.28) that "=" for partial function *calls*, $f(\vec{x})$ and $g(\vec{y})$, means the usual —equality of numbers— if both side are *defined*. $f(\vec{x}) = g(\vec{y})$ is also true if both sides are *undefined*. In symbols,

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

The "universality" of h lies in the fact that it (or the URM that computes it) acts like a "stored program" (i.e., general purpose or universal) "computer": To compute a function f we present both a "*program*" for f —*coded* as the number i— and the input *data* (the x) to h and then we let it crank along.

Proof. Each $\lambda x.f(x) \in \mathcal{P}$ is a $M^{\mathbf{x}}_{\mathbf{y}}$, by definition.

In 5.2.4 we proved that we can algorithmically enumerate *all* $\lambda x.f(x) \in \mathcal{P}$, with repetitions, by algorithmically enumerating all strings of the form $N^{\mathbf{x}}_{\mathbf{x}'}$ using the computable enumerator $\lambda i.F(i)$ that maps $i \in \mathbb{N}$ to the $N^{\mathbf{x}}_{\mathbf{x}'}$ —where N runs over all URMs.

Now we have three things to do:

1. *Define* $\lambda ix.h(i,x)$. Well, by the last sentence in the statement of the corollary, for each $i \in \mathbb{N}$, define $\lambda x.h(i,x)$ *to be* $F(i)$ from the proof of 5.2.4; that is, some $N^{\mathbf{x}}_{\mathbf{x}'}$.

2. Show that $h \in \mathcal{P}$. *Here is how the universal* h *is computed*

   - Given input i and x.
   - Call $F(i)$. This returns a unary computable function $N^{\mathbf{x}}_{\mathbf{x}'}$, for some N and variables $\mathbf{x}$ and $\mathbf{x}'$ in N.
   - Now run program N with x inputed into the input program-variable $\mathbf{x}$. If and when N stops, then we return the value held in the program-variable $\mathbf{x}'$ of N.

   By CT, $h \in \mathcal{P}$.

3. *Universality*: Given $\lambda x.f(x) \in \mathcal{P}$. Thus, $f = N^{\mathbf{x}}_{\mathbf{x}'}$, for some N and variables $\mathbf{x}$ and $\mathbf{x}'$ in N. By 5.2.4, there is a z such that $F(z) = N^{\mathbf{x}}_{\mathbf{x}'}$. By 1. above, h fulfils $\lambda x.h(z,x) = f$. □

We will next introduce a standard notation due to Rogers ([Rog67]):

5.3.2 Definition. In all that follows, ϕ_i will denote the i-th unary function in *the algorithmic list of* all $M^{\mathbf{x}}_{\mathbf{y}}$. □

5.3.3 Remark.

(1) Equipped with the above definition we can rephrase the Universal Function Theorem 5.3.1 as

$$h(i,x) = \phi_i(x), \text{ for all } i \text{ and } x$$

or even (better)

$$\lambda x.f(x) \in \mathcal{P} \text{ iff, for some } i \in \mathbb{N}, \text{ we have } f = \phi_i$$

It is worth "parsing" this "iff" above:

$\rightarrow$ direction: The hypothesis means $f = N_{\mathbf{v}}^{\mathbf{u}}$ for some N. If $N_{\mathbf{v}}^{\mathbf{u}}$ occupies location i in the list, then, by 5.3.2, $f = \phi_i$.

$\leftarrow$ direction: The hypothesis $f = \phi_i$ means that $f = N_{\mathbf{v}}^{\mathbf{u}}$, where $N_{\mathbf{v}}^{\mathbf{u}}$ occupies location i in the list. But, $f = N_{\mathbf{v}}^{\mathbf{u}}$ says that f is indeed computable; in $\mathcal{P}$.

(2) $\lambda ix.\phi_i(x) \in \mathcal{P}$ because $\lambda ix.h(i,x) \in \mathcal{P}$.

(3) Intuitively, 5.3.1 says that our theory is powerful enough to allow us to program a "compiler" for one-argument functions of $\mathcal{P}$: Indeed, a URM M with I/O convention such that $h = M_{\mathbf{z}}^{\mathbf{uv}}$ is such a compiler. In order to compute $\phi_x(y)$ we input the "program" x in $\mathbf{u}$ and the "data" y in $\mathbf{v}$ and, if and when the computation ends, $\mathbf{z}$ will hold the value $\phi_x(y)$.

(4) Calling x the "program" for $\lambda y.\phi_x(y)$ is not exact, but *is eminently apt*: x is just a number, not a set of URM instructions; but this number is the *address* (location) of *a URM program for* $\lambda y.\phi_x(y)$*. Given the address, we* can retrieve *this program from a list via a computational procedure, F of 5.2.4, in a finite number of steps!*

(5) *In the literature the address x in ϕ_x is called a ϕ-index. So, if $f = \phi_i$ then i is one of the infinitely many addresses where we can find how to program f.*

$\square$

5.3.4 Corollary. *For each $n \geq 1$, there is a* partial computable $(n+1)$*-variable function $H^{(n+1)}$ with this property: For any n-variable function $f \in \mathcal{P}$, there is a number $i \in \mathbb{N}$ such that $H^{(n+1)}(i, \vec{x}_n) = f(\vec{x}_n)$ for all $\vec{x}_n$. Equivalently, $\lambda \vec{x}_n.H^{(n+1)}(i, \vec{x}_n) = f$.*

Proof. As that for h, but using the enumeration of the $N_{\mathbf{x}'}^{\vec{\mathbf{x}}_n}$ instead (cf. Corollary 5.2.5). $H^{(2)} = h$. $\square$

Correspondingly we extend Rogers' notation:

5.3.5 Definition. In all that follows, $\phi_i^{(n)}$ —that is, in terms of the notation in the preceding corollary also $\lambda \vec{x}_n.H^{(n+1)}(i, \vec{x}_n)$— will denote the i-th n-ary function in *the algorithmic list of* all $M_{\mathbf{y}}^{\vec{\mathbf{x}}_n}$. Thus, $\phi_i^{(1)}$ is ϕ_i by definition (compare with 5.3.2). $\square$

5.4 The Kleene T-predicate and the normal form theorems

5.4.1 Definition. (The Kleene T predicate)
For any fixed $n > 0$, we define $T^{(n)}(z, \vec{a}_n, y)$ by

$$T^{(n)}(z, \vec{a}_n, y) \stackrel{Def}{\equiv} \text{the } z\text{-th URM } M_{\mathbf{x_1}}^{\vec{\mathbf{x}}_n} \text{ (5.2.5) on input } \vec{a}_n \text{ converges in } y \text{ steps}$$

If $n = 1$, then we write $T(z, a, y)$ for $T^{(1)}(z, a, y)$. □

5.4.2 Lemma. *For each $n > 0$, $T^{(n)}(z, \vec{a}_n, y)$ is in $\mathcal{PR}_*$.*

Proof. Refer to 2.4.25, 5.3.4 and 5.3.5.

Let $M_{\mathbf{x_1}}^{\vec{\mathbf{x}}_{n+1}}$ compute the universal $(n+1)$-variable function $H^{(n+1)}$ of 5.3.4. This is universal for all n-argument partial recursive functions $\phi_i^{(n)}$:

$$H^{(n+1)}(i, \vec{a}_n) = \phi_i^{(n)}(\vec{a}_n), \text{ for all } i \text{ and } \vec{a}_n$$

Our $T^{(n)}$ here is the T_M of 2.4.23, for the $(n+1)$-input URM

$$M_{\mathbf{x_1}}^{\vec{\mathbf{x}}_{n+1}} = \lambda z \vec{a}_n . H^{(n+1)}(z, \vec{a}_n)$$

thus is in $\mathcal{PR}_*$.

The Kleene *Normal Form theorem* is a fundamental result and tool in computability. It states,

5.4.3 Theorem. (Kleene Normal Form) *For each fixed $n > 0$ we have, for all $z, \vec{a}_n$,*

(1) $\phi_z^{(n)}(\vec{a}_n) \downarrow \equiv (\exists y) T^{(n)}(z, \vec{a}_n, y)$ *and*

(2) $\phi_z^{(n)}(\vec{a}_n) = out\Big((\mu y) T^{(n)}(z, \vec{a}_n, y), z, \vec{a}_n\Big)$

Proof. Refer to 2.4.25, 5.3.4, 5.3.5, and 5.4.2.

Let $M_{\mathbf{x_1}}^{\vec{\mathbf{x}}_{n+1}}$ compute the universal $(n+1)$-variable function $H^{(n+1)}$ of 5.3.4. By 2.4.25, we have for all $z, \vec{a}_n$,

$$H^{(n+1)}(z, \vec{a}_n) \downarrow \equiv (\exists y) T_M(z, \vec{a}_n, y) \tag{$*$}$$

and

$$H^{(n+1)}(z, \vec{a}_n) = out_M\Big((\mu y) T_M(z, \vec{a}_n, y), z, \vec{a}_n\Big) \tag{$**$}$$

where "T_M" and "out_M" are those of 2.4.25.

By the last remark in the proof of Lemma 5.4.2, "$T^{(n)}$ here is the T_M of 2.4.23" (here associated with $M_{\mathbf{x_1}}^{\vec{\mathbf{x}}_{n+1}}$). Replacing $H^{(n+1)}(z, \vec{a}_n)$ by $\phi_z^{(n)}(\vec{a}_n)$ in $(*)$ and $(**)$ and setting

$$out \stackrel{Def}{=} out_M$$

we obtain (1) and (2) of the theorem statement. □

5.5 A number-theoretic definition of $\mathcal{P}$

We know that $\mathcal{P}$ contains Z, S and all the U_i^n, for $n > 0$ and $1 \leq i \leq n$, and is closed under composition, (μy) and *prim*. Let us define then

5.5.1 Definition. ($\mathcal{P}$-derivations) The set

$$\mathcal{I} = \left\{ S, Z, \left(U_i^n \right)_{n \geq i > 0} \right\}$$

is the set of *Initial* $\mathcal{P}$-functions.[†]

A $\mathcal{P}$-*derivation* is a finite (ordered!) sequence of number-theoretic functions,

$$f_1, f_2, \ldots, f_i, \ldots, f_n$$

where, for each i , one of the following holds

1. $f_i \in \mathcal{I}$.
2. $f_i = prim(f_j, f_k)$ and $j < i$ and $k < i$ —that is, f_j, f_k appear *to the left of* f_i.
3. $f_i = \lambda \vec{y}.g\big(r_1(\vec{y}), r_2(\vec{y}), \ldots, r_m(\vec{y})\big)$, and *all* of the $\lambda \vec{y}.r_q(\vec{y})$ and $\lambda \vec{x}_m.g(\vec{x}_m)$ appear *to the left of* f_i in the sequence.
4. $f_i = \lambda \vec{x}.(\mu y) f_r(y, \vec{x})$, where $r < i$.

Any f_i in a derivation is called a $\mathcal{P}$-derived *function.* The symbol $\widetilde{\mathcal{P}}$, stands for the set of $\mathcal{P}$-derived functions, that is, That is,

$$\widetilde{\mathcal{P}} \stackrel{Def}{=} \{f : f \text{ is } \mathcal{P}\text{-derived}\} \qquad \square$$

The aim is to show that $\mathcal{P}$ is the set of all $\mathcal{P}$-derived functions as the terminology in 5.5.1 ought to clearly betray. Of course, we could also have said that $\widetilde{\mathcal{P}}$ is the *closure* of $\mathcal{I}$ above, under the operations *composition* and *primitive recursion* and *unbounded search* (cf. **??**).

We will achieve our aim by proving $\mathcal{P} = \widetilde{\mathcal{P}}$.

First a lemma:

5.5.2 Lemma. $\mathcal{PR} \subseteq \widetilde{\mathcal{P}}$.

Proof. Let $f \in \mathcal{PR}$. Then f is $\mathcal{PR}$-derived. But then it is also $\widetilde{\mathcal{P}}$-derived —a $\widetilde{\mathcal{P}}$-derivation need not necessarily use the (μy)-step 4 in 5.5.1. So, $f \in \widetilde{\mathcal{P}}$. $\square$

5.5.3 Theorem. $\mathcal{P} = \widetilde{\mathcal{P}}$.

[†]Same as the set of initial $\mathcal{PR}$-unctions of 2.1.1.

Proof. **Case** $\mathcal{P} \supseteq \widetilde{\mathcal{P}}$: This is by an easy induction on the length of derivation of an $f \in \widetilde{\mathcal{P}}$. The basis (length=1) is since $\mathcal{I} \subseteq \mathcal{P}$. The induction steps 2–4 (from Definition 5.5.1) follow from the closure properties of $\mathcal{P}$.

Case $\mathcal{P} \subseteq \widetilde{\mathcal{P}}$: Let $\lambda \vec{x}_n.f(\vec{x}_n) \in \mathcal{P}$. By 5.4.3, for some i,

$$f = \lambda \vec{x}_n.out\Big((\mu y)T^n(i, \vec{x}_n, y), \vec{x}_n\Big) \tag{1}$$

By the lemma, the right hand side of (1) is in $\widetilde{\mathcal{P}}$ (recall also 2.4.23 and 5.4.2). So is f, then. $\square$

Among other things, 5.5.3 allows us to prove properties of $\mathcal{P}$ by induction on $\mathcal{P}$-derivation length, and to show that $f \in \mathcal{P}$ via a way other than URM-programming: Place f is a $\mathcal{P}$-derivation.

The number-theoretic characterisation of $\mathcal{P}$ given here was one of the foundations of computability proposed in the 1930s, due to Kleene.

5.6 The S-m-n theorem

A fundamental theorem in computability is the *Parametrisation* or *Iteration* or also "*S-m-n*" theorem of Kleene. In fact, the *S-m-n*-theorem along with the universal function theorem and a handful of additional initial computable functions are known to be *sufficient* tools towards founding computability axiomatically —but we will not get into this matter in this volume.

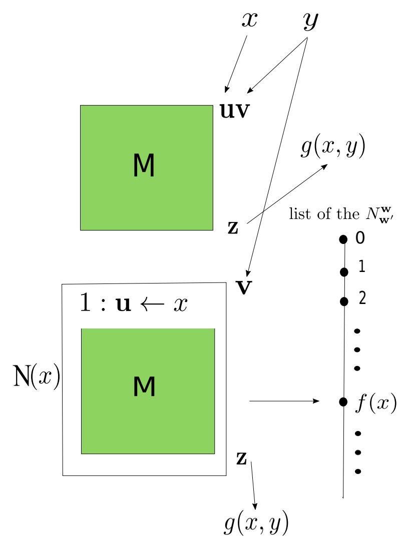
5.6.1 Theorem. (Parametrisation theorem) *For every $\lambda xy.g(x, y) \in \mathcal{P}$ there is a function $\lambda x.f(x) \in \mathcal{R}$ such that*

$$g(x, y) = \phi_{f(x)}(y), \textit{ for all } x, y \tag{1}$$

Preamble. (1) above is based on these observations: Given a program M that computes the function g as $M_{\mathbf{z}}^{\mathbf{uv}}$ with $\mathbf{u}$ receiving the input value x and $\mathbf{v}$ receiving the input value y —each via an "implicit" read statement— *we can, for* any *fixed value* x, *construct* a new program *dependent on the value* x, which behaves exactly as M does, because it consists of all of M's instructions, plus one more: The new program $N(x)$ —the notation "(x)" conveying the dependency of N on x— inputs x into $\mathbf{u}$ *explicitly* via an assignment statement *added* at the very top of M: $1 : \mathbf{u} \leftarrow x$.

Of course, if $x \neq x'$, the programs $N(x)$ and $N(x')$ differ in their first instruction, so they are different.

Let us denote, for each value x, the position of $N(x)_{\mathbf{z}}^{\mathbf{v}}$ in our standard effective enumeration of all the $N_{\mathbf{w}'}^{\mathbf{w}}$ by the expression $f(x)$, to convey the dependency on x. Clearly the correspondence $x \mapsto f(x)$ is functional (single valued), and moreover, by the last remark in the preceding paragraph, it is a 1-1 function.



In sum, the new program, $N(x)$, constructed from M and the value x is at location $f(x)$ of the standard listing —in the notation of 5.2.4, $F(f(x)) = N(x)^{\mathbf{v}}_{\mathbf{z}}$. Thus $N(x)^{\mathbf{v}}_{\mathbf{z}}$ with input y outputs $g(x, y)$ for said x, that is, in the notation introduced in Definition 5.3.2, we have

$$g(x, y) = \phi_{f(x)}(y), \text{ for all } y \text{ and the fixed } x \text{ —that is, for all } x \text{ and } y \tag{2}$$

Proof. Of the S-m-n theorem. The proof is encapsulated by the preceding figure, and much of the argument was already presented in the Preamble located between the two signs above (in particular, we have shown (2)).

Below we just settle the claim that we can compute the address $f(x)$ from x, that is, $\lambda x.f(x) \in \mathcal{R}$.

So, fix an input x for the variable $\mathbf{u}$ of program M. Next, *construct* $N(x)$. A trivial algorithm exists for the construction:

- Given M and x.
- Modify M into $N(x)$ by adding $1 : \mathbf{u} \leftarrow x$ at the top of M as a *new* "first" instruction. See the above figure.
- Change nothing else in the M-part of $N(x)$, but do *renumber* all the original instructions of M, from "$L : \ldots$" to "$L + 1 : \ldots$".

  Of course, every original M-instruction of the type

$$L : \textbf{if } \mathbf{x} = 0 \textbf{ goto } P \textbf{ else goto } R$$

must also change "in its action part", namely, into

$$L+1: \textbf{if } \mathbf{x}=0 \textbf{ goto } P+1 \textbf{ else goto } R+1$$

- Now —to compute $f(x)$— go down the effective list of **<u>all</u>** $N^{\mathbf{w}}_{\mathbf{w}'}$ and keep comparing to $N(x)^{\mathbf{v}}_{\mathbf{z}}$, until you find it in the list and return its address.

More explicitly,

proc $f(x)$
for $z = 0, 1, 2, \ldots$ **do**
if $F(z) = N(x)^{\mathbf{v}}_{\mathbf{z}}$ **then return** z

- The returned value z is equal to $f(x)$. Note that the if-test in the pseudo code will eventually succeed and terminate the computation, since *all* $N^{\mathbf{x}}_{\mathbf{x}'}$ are in the range of F of 5.2.4. In particular, this means that f is total.

By Church's thesis the informal algorithm above —described in five bullets— can be realised as a URM. Thus, $f \in \mathcal{R}$. □

Worth Repeating: It must not be lost between the lines what we have already observed: that the S-m-n function f is 1-1.

Two important corollaries suggest themselves:

5.6.2 Corollary. *For every* $\lambda x\vec{y}_n.g(x,\vec{y}_n) \in \mathcal{P}$ *there is a function* $\lambda x.f(x) \in \mathcal{R}$ *such that*

$$g(x,\vec{y}_n) = \phi_{f(x)}(\vec{y}_n), \textit{ for all } x, \vec{y}_n$$

Proof. Imitate the proof of 5.6.1 using the fact that we have an effective enumeration of all n-ary computable partial functions (5.2.5). □

5.6.3 Corollary. *There is a function* $S^m_1 \in \mathcal{R}$ *of* 2 *variables such that*

$$\phi^{(m+1)}_i(x,\vec{y}_m) = \phi^{(m)}_{S^m_1(i,x)}(\vec{y}_m), \textit{ for all } i, x, \vec{y}_m$$

Proof. The proof is that of 5.6.1 with a small twist: In the proof of 5.6.1 we start with a URM M for g. Here instead we have an *address* i of a URM for $\phi^{(m+1)}_i$, the latter being the counterpart of g in the current case.

The program $N(x)$ that we have built in the proof of 5.6.1 *depends* on the value x that is inputed via an assignment rather a read statement. Said program is a trivial modification of the program M for g, where the first input variable $\mathbf{u}$ loses its "input status" and participates instead in the very first instruction as "$1: \ \mathbf{u} \leftarrow x$".

The corresponding program here we will call $N(i,x)$ due to its obvious dependence on i that (indirectly) tells us *which* program "M" for $\phi^{(m+1)}_i$ we start with.

So, the construction of $N(i,x)$ is

1. Fetch the program for $\phi_i^{(m+1)}$ found in location i of the effective listing of all $N_{\mathbf{x}'}^{\vec{\mathbf{x}}_{m+1}}$. Call it $M_{\mathbf{z}}^{\mathbf{u},\vec{\mathbf{v}}_m}$, where we have also indicated its input/output variables.

2. Build $N(i,x)$ by adding $1:\ \mathbf{u} \leftarrow x$ before the first instruction of M. Shift all labels of M by 1, so that $N(i,x)$ is syntactically correct (cf. 5.6.1).

3. The $N(i,x)$ program, with its input/output variables indicated, is $N(i,x)_{\mathbf{z}}^{\vec{\mathbf{v}}_m}$ and can be located in the effective list of all $N(i,x)_{\mathbf{x}'}^{\vec{\mathbf{x}}_m}$ (cf. 5.2.5).

The argument for the recursiveness of S_1^m has a bit more subtlety than that of $f(x)$ of 5.6.1 due to the dependency on i. To compute the expression $S_1^m(i,x)$,

- Given i, x.

- Find the program at location i in the effective enumeration of all $N_{\mathbf{x}'}^{\vec{\mathbf{x}}_{m+1}}$. See step 1. in the construction above.

- Build $N(i,x)_{\mathbf{z}}^{\vec{\mathbf{v}}_m}$ as in step 2. above, and locate it in the effective list of all $N(i,x)_{\mathbf{x}'}^{\vec{\mathbf{x}}_m}$ (cf. 5.2.5).

- **Return** the address you found in the previous step. This is $S_1^m(i,x)$.

By CT and the 1–3 algorithm above, $S_1^m \in \mathcal{R}$. □

5.6.4 Corollary. *There is a function $S_n^m \in \mathcal{R}$ of $n+1$ variables such that*

$$\phi_i^{(m+n)}(\vec{x}_n, \vec{y}_m) = \phi_{S_n^m(i,\vec{x}_n)}^{(m)}(\vec{y}_m), \textit{ for all } i, \vec{x}_n, \vec{y}_m$$

Proof. This is now easy! In step 1. in the previous proof fetch the program for $\phi_i^{(m+n)}$ —instead of that of $\phi_i^{(m+1)}$— found in location i of the effective listing of all $N_{\mathbf{x}'}^{\vec{\mathbf{x}}_{m+n}}$. Call it $M_{\mathbf{z}}^{\vec{\mathbf{u}}_n,\vec{\mathbf{v}}_m}$, where we have also indicated its input/output variables. The counterpart of step 2. above is now to place the *program segment* below *before* all instructions of M:

$1:\ \mathbf{u}_1 \leftarrow x_1$
$2:\ \mathbf{u}_2 \leftarrow x_2$
$\vdots$
$n:\ \mathbf{u}_n \leftarrow x_n$

taking all the $\mathbf{u}_i$ off input duty.

The rest is routine and entirely analogous with the preceding proof, thus is left to the reader. □

The notation of the symbol S_n^m indicates that the first n variables of $\phi_i^{(m+n)}$ are taken off input duty while the last m of the original $m+n$ input variables have still input duty.

5.7 Unsolvable "problems"; the halting problem

Some of the comments below (and Definition 5.7.1) occurred already in earlier sections (2.2.1). We revisit and introduce some additional terminology (e.g., "decidable").

Recall that a number-theoretic *relation* Q is a subset of $\mathbb{N}^n$, where $n \geq 1$. A relation's outputs are **t** or **f** (or "yes" and "no"). However, a number-theoretic relation *must* have values ("outputs") also in $\mathbb{N}$.

Thus we *re-code* **t** and **f** as 0 and 1 respectively. This convention is preferred by recursion theorists (as people who do research in computability like to call themselves) and is the opposite of the re-coding that, say, the C language employs (0 for **f** and non-zero for **t**).

5.7.1 Definition. (Computable or Decidable relations) "*A relation* $Q(\vec{x}_n)$ *is* computable*, or* decidable" means that the function

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

is in $\mathcal{R}$.

The collection (set) of *all* computable relations we denote by $\mathcal{R}_*$. Computable relations are also called *recursive*.

By the way, we call the function $\lambda \vec{x}_n . c_Q(\vec{x}_n)$ —which does the re-coding of the outputs of the relation— the *characteristic function* of the relation Q ("c" for "characteristic"). □

Thus, "a relation $Q(\vec{x}_n)$ is computable or decidable" means that some URM computes c_Q. But that means that some URM behaves as follows:

On input $\vec{x}_n$, it halts and outputs 0 iff $\vec{x}_n$ satisfies Q (i.e., iff $Q(\vec{x}_n)$), it halts and outputs 1 iff $\vec{x}_n$ does *not* satisfy Q (i.e., iff $\neg Q(\vec{x}_n)$).

We say that the relation has a *decider*, i.e., the URM that *decides* membership of *any* tuple $\vec{x}_n$ in the relation.

5.7.2 Definition. (Problems) A "*Problem*" is a formula of the type "$\vec{x}_n \in Q$" or, equivalently, "$Q(\vec{x}_n)$".

Thus, by definition*, a "problem" is a* membership question. □

5.7.3 Definition. (Unsolvable Problems) A problem "$\vec{x}_n \in Q$" is called any of the following:

Undecidable
Recursively unsolvable
or just
Unsolvable
iff $Q \notin \mathcal{R}_*$—in words, iff Q is *not* a computable relation. □

Here is the most famous undecidable problem:

$$\phi_x(x)\downarrow \tag{1}$$

A different formulation of problem (1) is

$$x \in K$$

where

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

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

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

5.7.4 Theorem. *The halting problem is unsolvable.*

Proof. We show, *by contradiction*, that $K \notin \mathcal{R}_*$.

Thus we start by assuming the opposite.

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

that is, we can *decide membership* in K via a URM, or, what is the same, we can *decide truth or falsehood* of $\phi_x(x)\downarrow$ for any x:

Consider then the infinite matrix below, each row of which denotes a function in $\mathcal{P}$ as an array of outputs, the outputs being a natural number, or the special symbol "$\uparrow$" for any undefined entry $\phi_x(y)$.

By 5.3.1 *each* one argument function of $\mathcal{P}$ sits in some row (as an array of outputs).

$$\begin{array}{cccccc}
\color{red}{\phi_0(0)} & \phi_0(1) & \phi_0(2) & \ldots & \phi_0(i) & \ldots \\
\phi_1(0) & \color{red}{\phi_1(1)} & \phi_1(2) & \ldots & \phi_1(i) & \ldots \\
\phi_2(0) & \phi_2(1) & \color{red}{\phi_2(2)} & \ldots & \phi_2(i) & \ldots \\
\vdots & & & & & \\
\phi_i(0) & \phi_i(1) & \phi_i(2) & \ldots & \color{red}{\phi_i(i)} & \ldots \\
\vdots & & & & &
\end{array}$$

We will *show* that under the assumption (3) *that we hope to contradict*, the flipped diagonal[†] represents a *partial recursive function* as an array of outputs,

[†]All three [Rog67, Tou84, Tou12] use K for this set, but this notation is by no means standard. It is unfortunate that this notation clashes with that for the first projection K of a pairing function J. However the context will manage to fend for itself!

[†]Flipping all $\uparrow$ red entries to $\downarrow$ and vice versa. This flipping is a mechanical procedure by (3).

and hence *must* fit the matrix along some row i since we have that all ϕ_i (as arrays) are rows of the matrix.

On the other hand, flipping the diagonal is diagonalising*, and thus the diagonal function constructed* cannot *fit. Contradiction!* So, we must blame (3) and thus we have its negation proved: $K \notin \mathcal{R}_*$

In more detail, or as most texts present this, we have defined the flipped diagonal for all x as

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

Strictly speaking, the above does not *define* d since the "$\downarrow$" in the top case is not a value; it is ambiguous. Easy to fix:

One way to do so is

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

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

Given x, do:

- Use the decider for K (for $\phi_x(x)\downarrow$, that is) —assumed to exist by (3)— to test which condition obtains in (4); top or bottom.
- If the top condition is true, then we return 42 and stop.
- If the bottom condition holds, then transfer to an infinite loop, for examp:

while $1 = 1$ **do**
end

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

Say now

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

What can we say about $d(i) = \phi_i(i)$? Well, we have two cases:

Case 1. $\phi_i(i)\downarrow$. Then we are in the bottom case of (4). Thus $d(i)\uparrow$. But we also have $d(i) = \phi_i(i)$ by (5), thus we have just contradicted the case hypothesis, $\phi_i(i)\downarrow$.

Case 2. $\phi_i(i)\uparrow$. We have $d(i) = 42$ in this case, thus, $d(i)\downarrow$. By (5) $d(i) = \phi_i(i)$, thus again we have contradicted the case hypothesis, $\phi_i(i)\uparrow$.

So we reject (3). □

In terms of *theoretical significance*, the above is perhaps the most significant unsolvable problem that enables the process of discovering more! How?

As a first examp we illustrate the "program correctness problem" (see below).

But how does "$x \in K$" help? Through the following technique of *reduction*:

Let P be a new *problem* (relation!) for which we want to see whether $\vec{y} \in P$ can be solved by a URM. We build a *reduction* that goes like this:

(1) Suppose that we have a URM M that decides *$\vec{y} \in P$, for all $\vec{y}$.*

(2) Then we show how to use M as a subroutine *to also decide $x \in K$, for all x.*

(3) Since the latter problem is unsolvable*, no such URM M exists!* For short, $P(\vec{y})$ is unsolvable too.

The *equivalence problem* is

> Given two programs M and N can we test to see whether they compute the same function?

Of course, "testing" for such a question *cannot be done by experiment*: We *cannot just run M* and N for *all inputs* to see if they get the same output, because, for one thing, "all inputs" are infinitely many, and, for another, there may be inputs that cause one or the other program to run forever (infinite loop).

By the way, the equivalence problem is the general case of the "*program correctness*" problem which asks

> Given a program P and a *program specification* S, does the program *fit* the specification *for all inputs*?

since we can view a specification as just another formalism to express a function computation. By CT, all such formalisms, programs or specifications, boil down to URMs, and hence the above asks whether two given URMs compute the same function —program equivalence.

Let us show now that the program equivalence problem cannot be solved by any URM.

5.7.5 Theorem. (Equivalence problem) *The equivalence problem of URMs is the problem "given i and j; is $\phi_i = \phi_j$?"*[†]

This problem is undecidable.

Proof. The proof is by a reduction (see above), hence by contradiction. We will show that if we have a URM that solves it, "yes"/"no", then we have a URM that solves the halting problem too!

So assume we have an algorithm (URM) E for the *equivalence problem.* $(*)$

Let us use it to answer the question "$a \in K$"—that is, "$\phi_a(a)\downarrow$", for any a.

[†]If we set $P = \{(i,j) : \phi_i = \phi_j\}$, then this problem is the question "$(i,j) \in P$?" or "$P(i,j)$?".

$$\text{So, fix an } a \text{ that we want to test.} \tag{2}$$

Consider the following two computable functions given by:
For all x:

$$Z(x) = 0$$

and

$$\widetilde{Z}(x) = \begin{cases} 0 & \text{if } x = 0 \land \phi_a(a) \downarrow \\ 0 & \text{if } x \neq 0 \end{cases}$$

Both functions are intuitively computable: For Z we already have shown a URM M that computes it (first Note on URMs). For $\widetilde{Z}$ and input x compute as follows:

- Print 0 and stop if $x \neq 0$.
- On the other hand, if $x = 0$ then, using the universal function h start computing $h(a, a)$, which is the same as $\phi_a(a)$ (cf. 5.3.1). If this ever halts just print 0 and halt; otherwise let it loop forever.

By CT, $\widetilde{Z}$ is in $\mathcal{P}$, that is, it has a URM program, say $\widetilde{M}$.

We can *compute* the locations i and j of M and $\widetilde{M}$ respectively by going down the list of all $N^{\mathbf{w}}_{\mathbf{w}'}$. Thus $Z = \phi_i$ and $\widetilde{Z} = \phi_j$.

By assumption $(*)$ above, we proceed to feed i and j to E. This machine will halt and answer "yes" (0) precisely when $\phi_i = \phi_j$; will halt and answer "no" (1) otherwise. But note that $\phi_i = \phi_j$ iff $\phi_a(a) \downarrow$. We have thus solved the halting problem since a is arbitrary! This is a contradiction to the existence of URM E. □