A user-friendly Introduction to Computability via Turing Machines (and "Church's Thesis")

One way to think of a *Turing Machine*, or "TM" in short, is as an abstract model of a "computer". Like a computer, it can faithfully carry out simple instructions. To avoid technology-dependent limitations, it is defined so that it has unbounded "memory" (or storage space). Thus, it never runs out of storage during a computation.

A more accurate way to think of the class of all TMs is as a *programming formalism—or language*—and, therefore, to think of any *particular* TM, M, as a *program* written in that programming language.

We should allow our imagination to jump back and forth between the "machine model" and the "programming model", as convenience dictates. Beyond imagination, we will need a formal (mathematical) model of Turing Machines, for only with such a model we can hope to develop a ***theory*** of *Computability* (or *Recursion Theory*—as logicians prefer to call it[†]).

Thinking of a TM, M, as a "machine" we can describe it—informally at first—as follows:

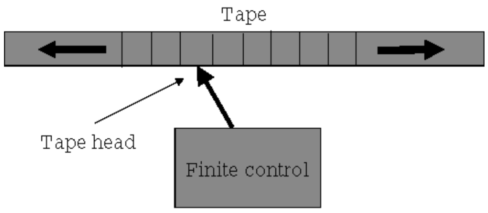
M consists of

(a) An infinite two-way tape.

(b) A read/write tape-head.

(c) A "black-box", the finite control, which can be at any one of a (fixed) *finite* set of internal states (or just, *states*). Pictorially, a TM is often represented as in the figure below:

[†]Computability is that part of Logic and Theoretical Computer Science which (1) formalizes the notion of computation and *computable function*, and (2) separates functions into two sets: One is that of computable functions, the other is that of the uncomputable functions.



The tape is subdivided into squares, each of which can hold a *single symbol* out of a finite set of admissible symbols associated with the TM—the *tape-alphabet*, Γ.

The tape-head can scan only *one square* at a time. The TM tape corresponds to the memory of an actual computer. In *one move*, the tape-head can move to scan the next square, to the left or to the right of the present square, but it has the option to stay on the current square.

The tape-head can read or write a symbol on the scanned square. Writing a symbol is assumed to erase first what was on the square previously.

There is a distinguished alphabet symbol, the blank***—denoted by*** B***—which appears everywhere except in a finite set of tape squares. This symbol is*** not ***used as "an input symbol".***

The machine *moves*, at each computation step, as follows:

Depending on

(a) The currently scanned symbol

(b) The current state

the machine will:

(i) Write a symbol or leave the symbol unchanged on the scanned square

(ii) Enter a (possibly) new state

(iii) Move the head to the left or right or it will leave it stationary.

We shall require TMs to be *deterministic*, i.e., that they behave as follows: Given the current symbol/state pair, they have a *uniquely defined* response.

Nondeterministic TMs do *not* have more computing power than the deterministic model but are important devices in the area known as *Complexity Theory.*[†] We *may* have time to show the equivalence of deterministic and nondeterministic TM formalisms (see Sipser [4] for a proof outline in the meanwhile).

[†]Which goes one step further than Computability: It classifies functions that we *can* compute to "easy" and "hard" (to compute, that is).

A TM computation *begins* by positioning the tape-head on the left-most non blank symbol on the tape (that is, the first symbol of the input string), "initializing" the machine (by putting it in a distinguished state, q_0), and then letting it go.

Our convention for *stopping the machine* is similar to that for PDAs (following Davis [1]): The machine will stop (or "*halt*", as we prefer to say) ***iff*** at some instance it is not specified how to proceed, given the current symbol/state pair. At that time (when the machine has halted), whatever is on the tape (that is, the largest string of symbols that starts and ends with some non blank symbol), is the result or output of the TM computation for the given input.

A question might now naturally arise in the reader's mind: How will the "TM operator" (a human) ever be sure that she/he has *seen* "the largest string on tape that starts and ends with some non blank symbol", when the tape is infinite? Is she or he doomed to search the tape forever and never be sure of the output?

This "problem" is not real. It arises here (now that I asked) due to the *informality* of our discussion above, which included talk about an "infinitely long tape medium". Mathematically—as we will shortly see—a TM just describes a "process" that produces a sequence of (*finite*!) strings, the ***last string*** of the sequence, if it exists,[†] being the output "generated".

At each step of the "computation" that produces these strings, the machine *finitely modifies* the current string in order to produce the next. All the operator has to do is read the last string, if and when the machine halts.

Essentially, the tape is the "medium" on which the machine writes these strings. Instead of an infinite tape, one can think then of an extensible, ***but finite*** tape. The tape is always just long enough to hold the current string!

1. *Definitions*

Now the formalities:

1.1 Definition. (TM—static description) A Turing Machine (TM), M, is a 4-tuple $M = (\Gamma, Q, q_0, \delta)$, where $\Gamma = \{s_0, s_1, \ldots, s_n\}$ is a finite set of *tape symbols* called the *tape-alphabet*, and $s_0 = B$ (the *distinguished* ***blank*** symbol).

$Q = \{q_0, q_1, \ldots, q_r\}$ is a finite set of states of which q_0 is ***distinguished***: It is the *start-state*.

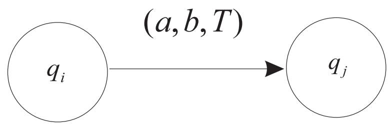
δ is the behaviour or *transition function*, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$. In using a TM one often chooses a *subset*, Σ, of Γ that never includes the blank symbol. Σ is the alphabet used to form input strings; the *input alphabet*. □

In the specification of δ above, L (respectively R, S) stands for "head moves left" (respectively "right", "stays put"). Also, requiring the δ relation to be a *function* makes it "single-valued". This translates to what we announced already: Our TMs are deterministic.

[†]It will not exist iff the sequence has infinite length.

Finally, note that $\delta(q_j, s_i) = (q_m, s_k, S)$ means that if the machine is in state q_j, and the head scans s_i, then the machine replaces s_i by s_k (of course, it *may* be $s_i = s_k$) enters state q_m (of course, it *may* be $q_m = q_j$) and the head does not move.

Since δ is a function on a finite set, *it is itself a finite set* and therefore there are a number of ways it can be *listed*. For example, as a table—where each s_i-symbol labels exactly one row and each q_j-state labels exactly one column (or the other way around)—or as a state-diagram, as in the figure below that represents $\delta(q_i, a) = (q_j, b, T)$ with $T \in \{L, R, S\}$.



The latter is usually much more user-friendly than the former when it comes to illustrative “programming examples”. As usual, the “node” for the start-state q_0 is pointed at by an arrow that originates nowhere.

For theoretical studies of Turing Machines it is advantageous to represent δ in yet another way, as ***a set of quintuples*** (essentially Turing’s own approach; [1] uses a similar representation, but instead of quintuples he uses quadruples).

A quintuple has the form $q_j s_i s_k q_m T$ and stands for the relation “$\delta(q_j, s_i) = (q_m, s_k, T)$”, where $T \in \{L, R, S\}$.

Thus we may give an alternative “static” definition of Turing Machines.

1.2 Definition. (TM—alternative static definition, quintuples) A TM, M, over the (tape) alphabet $\Gamma = \{s_0, \ldots, s_n\}$—where $s_0 = B$—with state-set $Q = \{q_0, \ldots, q_m\}$, is just ***a finite set of quintuples of the form*** $qabq'T$, where $\{q, q'\} \subseteq Q$, $\{a, b\} \subseteq \Gamma$ and $T \in \{S, L, R\}$. It is required that no two quintuples begin with the same two symbols. $\square$

The last restriction ensures that the relation that maps (q, a) to (q', b, T) is a function.

1.3 Definition. (IDs) Fix a TM, M, over (tape) alphabet Γ and state alphabet Q . An *instantaneous description* (*ID*) of M is a string $t_1 q a t_2$, where $q \in Q$, $a \in \Gamma$ and $\{t_1, t_2\} \subseteq \Gamma^*$. $\square$

Intuitively, an ID is a “snapshot” of a computation at a moment in “time”. The tape contents at the time is the string $t_1 a t_2$—that is, ***the string produced, or computed, so far***—the current state is q and the scanned symbol is a.

Compare with the case of PDA IDs. There we only needed the “unspent” part of the tape in an ID because the (input) head never moves left. Here we have to include the tape contents both at the left and right of the tape head.

1.4 Definition. (Yields) Given two IDs α and β of a TM, M, we say that "α *yields* β", in symbols $\alpha \vdash \beta$ (or $\alpha \vdash_M \beta$ if we want to emphasize that we are referring to machine M) iff one of the following conditions holds (see 1.3 for the meaning of the symbols used below):

(i) $qabq'S \in M$, $\alpha = t_1qat_2$ and $\beta = t_1q'bt_2$

(ii) $qabq'R \in M$, $\alpha = t_1qact_2$ for some $c \in \Gamma$ and $\beta = t_1bq'ct_2$

(iii) $qabq'R \in M$, $\alpha = t_1qa$ and $\beta = t_1bq'B$

(iv) $qabq'L \in M$, $\alpha = t_1cqat_2$ for some $c \in \Gamma$ and $\beta = t_1q'cbt_2$

(v) $qabq'L \in M$, $\alpha = qat_2$ and $\beta = q'Bbt_2$

□

The intuitive idea that the machine tape is unbounded (or infinitely extensible) is incorporated in the cases (iii) (extensible to the right—or "right-infinite") and (v) (extensible to the left—or "left-infinite") of Definition 1.4.

NOTE. Sipser ([4]) does not allow left-infinite tapes.

The following is similar to the corresponding definition for PDAs.

1.5 Definition. (Terminal or Final IDs) t_1qat_2 is a *final* or *terminal* ID of a TM, M, iff M contains *no quintuple* that starts with the string qa. □

1.6 Remark. A terminal (or, final) ID causes M to stop (halt) its computation. The above definition follows Turing (see also Davis [1]) and says that a TM halts with ID t_1qat_2 iff $\delta(q, a) \uparrow$ where, in general, we denote by the symbol "$f(c) \uparrow$" the statement "$f(x)$ is **undefined** at $x = c$" (correspondingly, "$f(c) \downarrow$" denotes the statement "$f(x)$ is **defined** at $x = c$").

Alternative conventions require the machine to halt as soon as it enters any one of a distinguished subset of states called halting states (see Hopcroft and Ullman [2] for this approach as well as for a discussion of a large variety of variants of the TM model).

Sipser even requires the presence of "accept" and "reject" states, but, as we have discussed in class, this over-design is unnecessary for TMs since they can write on the tape. For example, they can write the strings "yes" or "no" on the tape to indicate acceptance or rejection. Another reason that reject/accept states are not only useless but also annoying is that neither accept nor reject states are ***of any relevance at all*** when we apply a TM as a "*computer for functions*".

Finally, the concept of TM-computation:

1.7 Definition. (Computations) A *computation* of a TM M is a ***finite sequence*** of IDs $\alpha_1, \ldots, \alpha_k$ such that

(1) α_k is final

(2) if $k > 1$, then $\alpha_i \vdash \alpha_{i+1}$, for $i = 1, \ldots, k-1$

(3) $\alpha_1 = q_0 t$, ***the initial ID***, where q_0 is, of course, the start-state and $t \in \Sigma^+$.

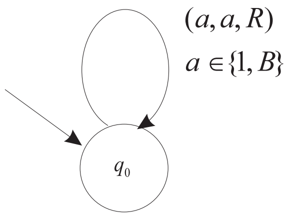
Recall that Σ, the ***input alphabet***, is a subset of $\Gamma - \{B\}$.

For any i, $\alpha_i \vdash \alpha_{i+1}$ is ***one step*** of the computation. $\square$

1.8 Remark. In the next section we will fix $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, B\}$, ***without any loss of generality***. That is, these small *fixed* alphabets are going to suffice to found the theory of all computable functions.[†]

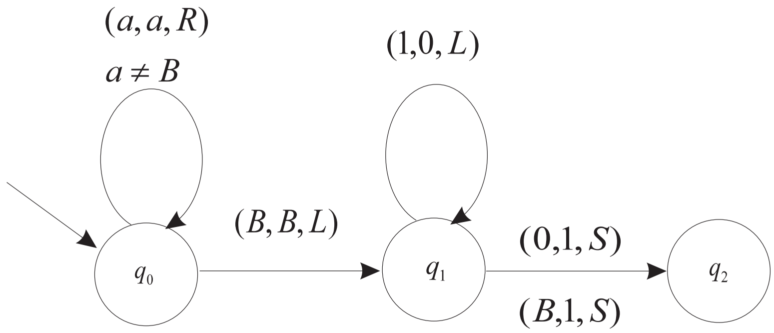
Here are some examples.

1.9 Example. The following is a TM over $\Gamma = \{1, B\}$ that never halts, regardless of what string in $\{1\}^+$ is presented as input.



$\square$

1.10 Example. The following is a TM over $\Gamma = \{0, 1, B\}$ that computes $x + 1$ if the number x is presented in *binary notation*.



$\square$

In Computability one studies exclusively *number theoretic functions* and relations.

[†]As a matter of fact, Minsky has shown that $\Gamma = \{1, B\}$ is all you ever need.

"Number-theoretic" ***means*** that both the inputs and the outputs are members of the set of natural numbers, $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.[†]

This choice of "what to attempt computing" presents no loss of generality, because all finite ***objects with which we may want to compute—e.g., negative integers, rational numbers, strings, matrices, graphs, etc.—can be coded by natural numbers (indeed, by "binary strings"***[‡] ***which in turn we may think of as natural number representations)***.

Having fixed the "game" to be a theory of number theoretic (computable) functions and relations, the next issue is:

What is a convenient input/output convention for Turing Machines, when these are used to "do Computability"?

The custom is to adopt the following input/output ("I/O") conventions for TMs (see [1, 3, 5]):

1.11 Definition. (I/O) A number x is presented as input in ***unary notation***, that is, it is represented as a string of length $x + 1$ over $\{1\}$.

Note the length!

More generally, a "vector input" $x_1, x_2, \ldots, x_n$, often abbreviated as $\vec{x}_n$—or simply $\vec{x}$ if the length n is unimportant, or understood—is represented as

$$1^{x_1+1}01^{x_2+1}0\ldots01^{x_n+1}$$

where, as always, for a string v and a positive integer i

$$v^i \stackrel{\text{def}}{=} \underbrace{vv\cdots v}_{i \text{ copies}}$$

If, on a given input, the TM halts, and β is its (unique) terminal ID, then the ***integer-valued output*** is ***the total number of occurrences of the symbol "1" in the string*** β.

If α is the initial ID of a computation, then we use the symbol

$Res(\alpha)$, or $Res_M(\alpha)$ if we want to say what machine we have in mind

to indicate the ***integer-valued*** output at the end of this computation, as defined above. □

We emphasize two points:

(1) If α is some initial ID ($q_0 t$) of a TM M, we have no guarantee that $Res(\alpha)$ is defined (because we have no guarantee that we have a computation starting with α). E.g., *no* $q_0 1^{x+1}$ leads to a computation in the machine of Example 1.9.

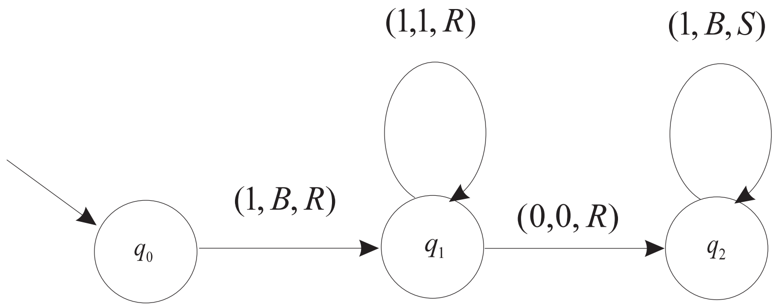
(2) Looking back at Definition 1.7 we see that the term "computation" is reserved for *terminating computation.*

Nevertheless, we often abuse terminology and say "non terminating computation" (or, we add unnecessary emphasis, and say "terminating computation").

[†] A relation returns "yes" or "no", or equivalently, "true" or "false". Traditionally, we code "yes" by 0 and "no" by 1—this is opposite to the convention of the C-language!—thus making relations a special case of (number theoretic) functions.

[‡] That is, strings over $\{0, 1\}$. Such strings we may also call ***bit strings***.

1.12 Example. The following TM computes the function.
input: x, y
output: $x + y$



We indulged above in a bit of "obscure programming" to avoid a 4th state: The last "loop" is not a loop at all. Once a 1 is replaced by a B and the head does not move, q_2 has no moves ($\delta(q_2, B) \uparrow$) hence the machine halts.

It is easy to check that the computations (for any choices of values for x, y) here are

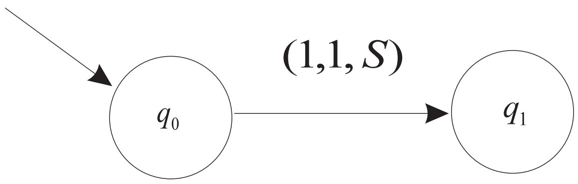
$$q_0 1^{x+1} 0 1^{y+1} \vdash^* B 1^x 0 q_2 B 1^y$$

thus, as we set out to do

$$Res(q_0 1^{x+1} 0 1^{y+1}) = x + y$$

$\square$

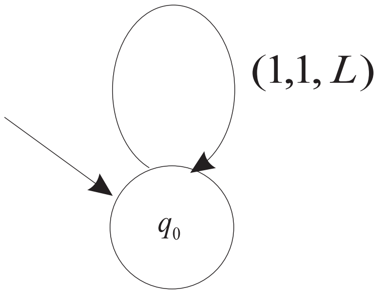
1.13 Example. The following TM computes the function.
input: x
output: $x + 1$
This time we do it in unary, as our "Computability conventions" dictate.



Here

$$q_0 1^{x+1} \vdash^* q_1 1^{x+1}$$

Another solution is



Here

$$q_0 1^{x+1} \vdash^* q_0 B 1^{x+1}$$

□

1.14 Example. A TM does ***not*** determine its number of arguments (1-vector, 2-vector, etc.), unlike "practical" programming languages where "read" statements and/or procedure parameters leave no doubt as to ***what is the correct number of inputs***.

► ***There is no "correct" or predetermined number of inputs for a TM.***

As long as we stick to the conventions of Definition 1.11, we can supply vectors of any length whatsoever, as input, to any given TM. ◄

For example, since the last TM above also has the computation

$$q_0 1^{x+1} 0 1^{y+1} 0 1^{z+1} \vdash^* q_0 B 1^{x+1} 0 1^{y+1} 0 1^{z+1}$$

it also computes the function

input: x, y, z

output: $x + y + z + 3$

As another example, looking back to Example 1.9 we have, for all x that

$$Res(q_0 1^{x+1}) \uparrow$$

that is, that TM computes the (unary, or one-argument) ***empty function***, denoted by $\emptyset$ just like the empty set. That is,

input: x

output: UNDEFINED

However, give two (or more) inputs, and it computes something altogether different!

For example, it computes

input: x, y

output: $x + y + 2$

since

$$q_0 1^{x+1} 0 1^{y+1} \vdash^* 1^{x+1} q_0 0 1^{y+1}$$

□

We witnessed some "neat" things above. There is one thing that (purposely) stuck out throughout, to motivate the following definition. That was the cumbersome designation of functions by writing

input: bla-bla
output: bla-bla

1.15 Definition. (λ notation) There is a compact notation due to Church, called λ *notation*, that denotes a function given as

input: $x_1, x_2, \ldots, x_n$
output: E
where "E" is an expression, or a "rule" on how to obtain a value.

We write simply "$\lambda x_1 x_2 \ldots x_n.E$".

Thus "λ"–"." is a "**begin**"–"**end**" block that delimits the arguments, and immediately after the "." follows the "output rule, or expression". □

1.16 Example. Thus, the first function discussed in Example 1.14 is

$$\lambda xyz.x + y + z + 3$$

the 2nd is

$$\lambda x. \uparrow$$

NOTE. "$\uparrow$" is ***not*** a value or number! All that we say by the above notation is: "No matter what the input value x, there is **no output**".

Since we gave a name to the empty function (or *totally undefined function*) in 1.14, we may write

$$\emptyset = \lambda x. \uparrow$$

Careful! Do *not* write

$$\emptyset(x) = \lambda x. \uparrow$$

The left hand side is (an undefined) ***value***, the right hand side is a ***function***. The types of these two objects don't match; they cannot possibly be equal!

You *may* write

$$\emptyset(x) = \uparrow$$

or, better still,

$$\emptyset(x) \uparrow$$

The final function in 1.14 is

$$\lambda xy.x + y + 2$$

□

1.17 Definition. (Partial functions) A *partial (number theoretic) function* f is one that *perhaps* is not defined on all values of the input(s). Thus, ***all*** functions that our theory studies are partial.

A function is *total* iff it is defined for *all possible values* of the input(s).

In the opposite case we say we have a *nontotal* function.

Thus, if we put all total and all nontotal functions together, we obtain all the partial functions.

For any two partial functions f and g of one argument, the statement

$$f(a) = g(b)$$

means that either both sides are *undefined*, or both are *defined* and have the same (numerical) value. In symbols (borrowing from MATH1090) we write this as

$$f(a) \uparrow \wedge g(b) \uparrow \vee (\exists x)(f(a) = x \wedge g(b) = x)$$

This understanding of "=" in the presence of partial functions is due to Kleene. We may call it "Kleene equality". □

Thus "partial" is just a wishy-washy term (unlike the terms "total"/"nontotal"), that does *not* in itself tell us whether a function is *total* or *nontotal*.

It just says, "Caution! This function ***may***, for some inputs, be undefined".

At long last!

1.18 Definition. (Computable (partial) function) A function $\lambda \vec{x}_n . f(\vec{x}_n)$ is a "*Turing computable partial function*"—but (following the literature) we rather say ***partial (Turing) computable function***—iff there is a TM M such that

$$\text{For all } a_i \ (i = 1, \ldots, n) \text{ in } \mathbb{N}, \quad f(\vec{a}_n) = Res_M(q_0 1^{a_1+1} 0 1^{a_2+1} 0 \cdots 0 1^{a_n+1})$$

where the "=" is "Kleene equality" (Definition 1.17)

A partial computable function is also called ***partial recursive***. □

Why being so fancy? "A function $\lambda \vec{x}_n . f(\vec{x}_n)$".

Well, I cannot say "A function $f(\vec{x}_n)$", because this object is ***not*** a function, it is rather a number (which I do not happen to know, because you forgot to give me the values of the $\vec{x}_n$).

The alternatives

"A function f of arguments $\vec{x}_n$"

or

"A function f with

input: $\vec{x}_n$

output: $f(\vec{x}_n)$"

are rather ugly.

1.19 Definition. **($\mathcal{P}$ and $\mathcal{R}$)** The set of all *partial computable* (*partial recursive*) functions is denoted by the (calligraphic) letter $\mathcal{P}$.

The set of all *total computable* (*total recursive*) functions is denoted by the (calligraphic) letter $\mathcal{R}$.

Indeed, people say just ***recursive*** (or ***computable***) function, and they *mean* total (computable). [***We have to get used to this!***] $\square$

1.20 Theorem. *$\mathcal{R} \subset P$.*

Proof. That $\mathcal{R} \subseteq P$ is a direct consequence of definition 1.19. That the subset relation is *proper* follows from examples 1.9 and 1.14: The function $\emptyset$ is in $\mathcal{P}$, but it is not in $\mathcal{R}$. $\square$

2. Fixing the alphabet

The theory gets a boost in this section! We will show (but *not* in its full gory detail) that we can restrict our tape alphabet to just

$$\Gamma_0 = \{0, 1, B\} \tag{1}$$

and still be able to compute (with TMs so restricted) *all* the functions of $\mathcal{P}$ (that in the previous section were defined in terms of TMs with unrestricted alphabets $\Gamma = \{B, 0, 1,$ ***plus possibly*** other symbols$\}$).

So let M be a TM that has alphabet

$$\Gamma = \{s_0, s_1, s_2, \ldots, s_k\} \tag{2}$$

where $s_0 = 0, s_1 = 1, s_2 = B$ and $k > 2$.

Let $l > 0$ be such that

$$2^{l-1} \leq k < 2^l$$

(**Pause.** Is such an l always available?)

Thus, any i, $0 \leq i \leq k$ can be written in binary, ***using exactly*** l ***digits*** (using leading 0's if necessary).

We "code" each s_i by the number i in binary of fixed length l

Symbol	Length-l binary code
$s_0 = 0$	0^l
$s_1 = 1$	$0^{l-1}1$
$s_2 = B$	$0^{l-2}10$
s_3	$0^{l-2}11$
s_4	$0^{l-3}100$
$\vdots$	$\vdots$
s_k	k in binary possibly with leading 0s

We are going to build a TM N over the restricted alphabet Γ_0 (see (1) above) such that for any $\vec{x}_n$ (this "for any" includes "for any $n > 0$")

$$Res_M(q_0 1^{x_1+1}01^{x_2+1}0\cdots 01^{x_n+1}) = Res_N(q_0 1^{x_1+1}01^{x_2+1}0\cdots 01^{x_n+1})$$

The idea is simple:

N operates as follows:

Sim1. Translate the input of N

$$1^{x_1+1}01^{x_2+1}0\cdots 01^{x_n+1} \tag{3}$$

to its "binary code"

$$(0^{l-1}1)^{x_1+1}0^l(0^{l-1}1)^{x_2+1}0^l\cdots 0^l(0^{l-1}1)^{x_n+1} \tag{4}$$

Sim2. Now (that is, immediately after the translation) make N faithfully simulate M by treating (appropriate) blocks of binary digits as a single symbol, s_i, of M.

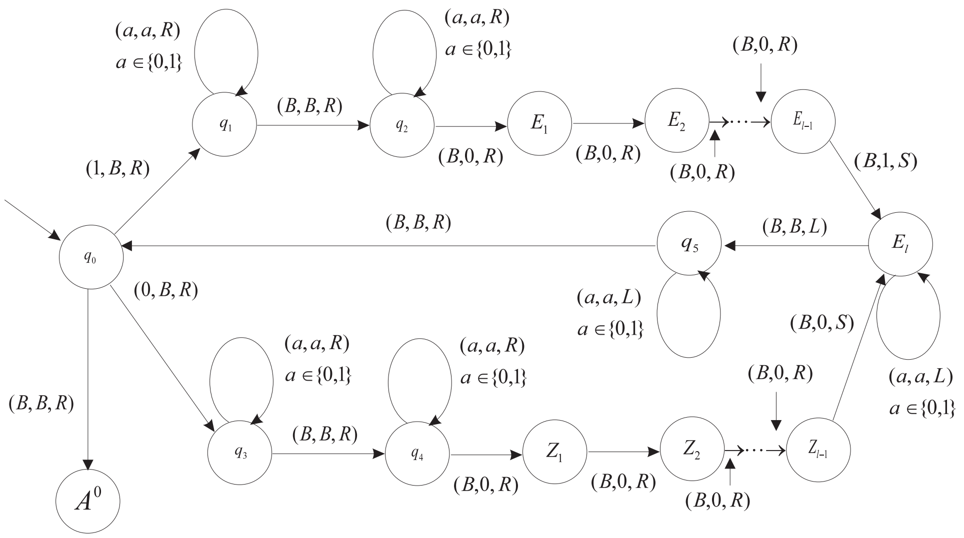
By "appropriate" we mean that N will not lose track of proper block boundaries. This is simple to do: A *complete* ***simulation***-*move* of N (left or right) that simulates **one** move of M is by l squares.[†] Moreover, N starts simulating on the leftmost symbol of the leftmost block (in state A^0; see below).

Sim3. If (and only if) the simulation halts, N erases all the blocks, except the $0^{l-1}1$ blocks (that represent "original" 1's).

The translation (**Sim1**) is done by the following TM-fragment. This halts at state A^0, where the direct simulation (**Sim2**) begins. The E-states moves code a single "1", while the Z-states do the same for "0". This fragment erases

[†]Of course, ***one*** N***-move*** is just by ***one*** N***-square***—a square that can hold ***one*** of $0, 1, B$.

input (3) one symbol at a time and replaces it (to the right of the original) by “coded input”, (4).



We turn now to **Sim2**. If $\{q_0, q_1, \ldots, q_s\}$ are the states of M, we have replaced them in N by the states $\{A^0, A^1, \ldots, A^s\}$, A^i replacing q_i. This renaming is for a more serious reason than just to avoid (state-)name clashes (names such as “q_0” have already been used to effect the input translation).

To see this, suppose that in the course of an M-computation

$$q_i s_j s_r q_m S \tag{5}$$

is applicable. That is, M is at state q_i and scans symbol s_j.

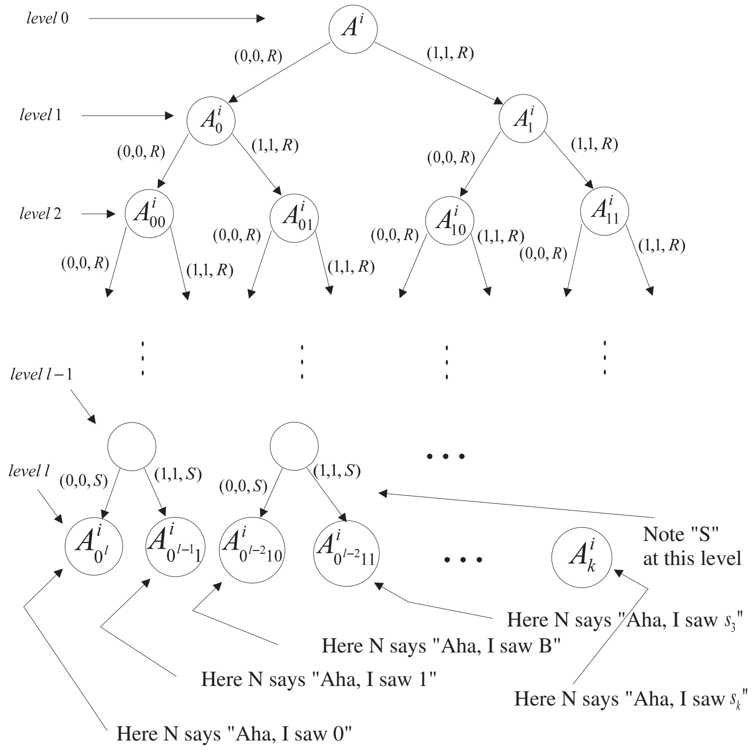
Now, N will be at state A^i, but ***unlike*** M ***which can see*** s_j ***at a glance***†, it ***cannot readily tell*** that it is looking at s_j since the latter is—***for*** N—the number j coded as a binary string of length l (possibly with leading 0’s).

N has first *to recognize* that it sees s_j, before it can simulate instruction (5) according to the “macro” (6) below (N “speaks” below)

A^i : “Aha, I saw (code for) s_j. Let me write (code for) s_r in its place, and then go to state A^m and stay in the same ‘super-square’ that I just rewrote, that is, go to its leftmost symbol.” (6)

Thus, A^i is not just q_i renamed. q_i is ready to act, while A^i has to first initiate a recognition process to figure out what symbol is being scanned. It has to implement the “Aha”. The following TM fragment shows how N does it:

† s_j occupies ***the current single square*** as far as M is concerned.



Note the persisting superscript i throughout the tree of moves above. It helps N to "remember" that A^i initiated this recognition sub-computation. Also, at every level the bit string subscripts "remember" what "bit string" has been seen so far. At level l the machine N has seen all the l "bits" and the recognition is complete.

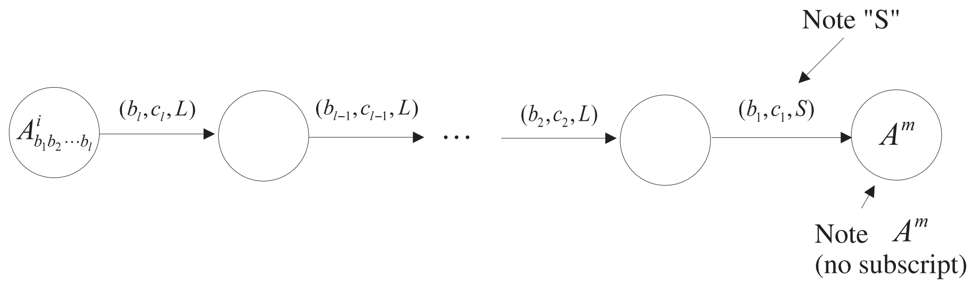
To conclude the action of the macro (6), N now goes back left and writes as it goes (right to left) the binary representation (with leading 0s) of r—this is the code for s_r. This is done by a TM-fragment like the one below, where we have written

$$b_1 b_2 \ldots b_l, \text{ with } b_i \in \{0, 1\}$$

as the binary representation of j, that is, the code of s_j, and

$$c_1 c_2 \ldots c_l, \text{ with } c_i \in \{0, 1\}$$

for the binary representation of r, that is, the code of s_r



We did not bother to give names to the "internal" states above. Note how "flow control" was surrendered to A^m at the end of this write-cycle, just as macro (6) required.

If instead of "S" we had "R" in (5), then N would have to go l squares to the right (after it finished writing s_r as above). Similarly, if we had an "L", then N would have to go l squares to the left (after it finished writing s_r as above).

Needless to point out that if N needs to simulate M in a situation where a "new B" appears (from the point of view of M) because M run out of tape (at the left or at the right), then N must write instead a ***code*** for (M's) B,[†] that is a bit string "$0^{l-2}10$".

The part **Sim3** of the simulation is trivial, and left to the reader's imagination.

We can summarize:

2.1 Theorem. *Every function in $\mathcal{P}$ (and hence in $\mathcal{R}$) can be computed by a TM over the alphabet $\Gamma = \{0, 1, B\}$. Of course, Σ the* input alphabet *is* $\{0, 1\}$, *and **the I/O convention remains the same one that we introduced in Definition 1.11**.*

3. Standard enumeration of TMs

3.1 Remark. From now on our TMs are out of the fixed alphabets $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, B\}$. Thus a TM is just a set of quintuples (these quintuples implicitly tell us what the states are!) that

(a) Are ***consistent*** , i.e., no two distinct quintuples of a TM have the same prefix "$q_i s_j$",

(b) There is at least one quintuple that begins with "$q_0 a$" for some a (in Γ).

[†]For N, of course, ***its own*** B is a B is a B. But here we are talking about N simulating M. In so doing, N ***denotes*** M***'s symbols—including*** M***'s*** B—by length-l bit strings.

We can convert the *set* of quintuples into a string, by using a *new symbol*, say "#", as "glue". Moreover we can generate the infinite variety of states with just two symbols, "q" and "1" as

State	**Code**
q_0	$q1q$
q_1	$q11q$
q_2	$q111q$
q_3	$q1111q$
$\vdots$	$\vdots$
q_i	$q1^{i+1}q$
$\vdots$	$\vdots$

Thus, we fix the alphabet $\mathcal{A}$

$$\mathcal{A} = \{0, 1, B, \#, q, S, L, R\} \tag{1}$$

and ***code*** every TM, M, (that is, a set of quintuples satisfying (a)–(b) of 3.1)

$$M = \{\ldots, q_i s_j s_m q_r R, \ldots\}$$

as a ***string*** $\langle M \rangle$[†]

$$\langle M \rangle = \# \ldots \# q1^{i+1} q s_j s_m q 1^{r+1} q R \# \ldots \# \tag{2}$$

That is, the string starts and ends with "#", and between two successive #-symbols we code a quintuple

$$q_i s_j s_m q_r T, \quad T \in \{S, L, R\}$$

by the string over $\mathcal{A}$ ((1) above)

$$q1^{i+1} q s_j s_m q 1^{r+1} q T, \quad \{s_j, s_m\} \subseteq \Gamma, \quad T \in \{S, L, R\} \tag{3}$$

In plain English, to go from

$$M = \{\ldots, q_i s_j s_m q_r R, \ldots\} \tag{4}$$

to

$$\langle M \rangle$$

do the following *replacements*:

[†]This code is not unique because we can permute the quintuples of M without changing M: M is a *set*.

1. Replace every "{", "}" and "," in (4) above by "#"
2. Replace every state q_i by the string $q1^{i+1}q$
3. (Leave all other symbols unchanged.)

Not only we can code an M by the above simple construction, but we can test by a simple algorithm whether a string over $\mathcal{A}$ is a code for some TM, and if so ***of which TM***.

Indeed, given a string w over $\mathcal{A}$, we test as follows:

(I) Test for correct format of w, that is:

  (a) w starts and ends with #

  (b) Between successive #-symbols there is a string z of form (3) above

(II) Ensure that there are *no* two distinct strings z and z' anywhere in w (as substrings), as described in (b) above, that have the same prefix "$q1^{i+1}qs_j$" (for some i, j)

(III) Ensure that there is *at least one* occurrence of a substring z of w, as described in (b) above, that has "$q1qs_j$" as a prefix ($s_j \in \Gamma$)

(IV) If all the above tests succeed, then we have a TM. It is trivial to read its quintuples (separated by #'s), in consultation with the table of p.17.

We can now generate a standard algorithmic listing of all TM codes, and, hence, of all the TMs!

Enum1. Fix an order on the elements of $\mathcal{A}$. We fix the order

$$0 < 1 < B < \# < q < S < L < R$$

Enum2. We build two lists simultaneously. "List 1" contains *all* strings over $\mathcal{A}$; "List 2" contains only those strings that code TMs.

The strings in "List 1" are generated by increasing string length. ***Within each string length, the strings are produced in increasing lexicographic order, according to the symbol order we adopted in* Enum1*, above.***

For each string, w, (of "List 1") that we generate we do:

if w codes a TM,[†] **then** also place w in the next available position in "List 2"

We note that this listing of codes lists every TM many times, since a TM can be coded by as many codes as there are permutations of its quintuples.

We also note that the term "algorithmic" has the informal, or intuitive, meaning. For now.

We summarize:

3.2 Theorem. (The TM standard-listing) *There is an algorithmic (or "constructive") listing of* **all** *Turing Machines.*

We denote this listing as $M_0, M_1, M_2, M_3, \ldots$, that is, we enumerate using "0" as the first index.

3.3 Definition. (Rogers' ϕ-notation) The symbol "$\phi_e^{(n)}$" will denote ***in the balance if this paper*** the number theoretic function of n arguments computed by the e-th TM, M_e, of our "standard-listing" (Theorem 3.2). If $n = 1$, then we will write ϕ_e rather than $\phi_e^{(1)}$.

This notation was introduced by Rogers.

The reader will recall the discussion of Example 1.14, where we saw that a TM *cannot* specify its number of inputs.

Thus, ***a given M_e will compute a function of as many arguments as we are willing to supply with the initial ID***.

Definition 3.3, along with Theorem 3.2 have a trivial (but useful) corollary:

3.4 Corollary. *A number theoretic function f, of n arguments, is in $\mathcal{P}$ (i.e., is partial recursive) iff, for some $e \in \mathbb{N}$,*

$$f(\vec{x}_n) = \phi_e^{(n)}(\vec{x}_n), \text{ for all } \vec{x}_n$$

which we can say much more simply as

$$f = \phi_e^{(n)}$$

[†]We can test for this algorithmically!

4. Church's Thesis

The aim of Computability is to formalize (for example, via Turing Machines) the *informal* notions of "algorithm" and "computable function" (or "computable relation").

We have defined TMs rigorously, but in discussing their behaviour, in particular in presenting the standard-alphabet model, in discussing its simulating power, and in presenting our TM-enumeration result we have suppressed a lot of (implementational) details, and therefore we were somewhat *informal*.

What gives?

Well, in our defense, TM is a very user-unfriendly assembly-like programming formalism. Indeed it is more unfriendly than assembly language, for it only processes (essentially) one digit at a time, whereas, at least, assembly language has the ability to add or multiply entire numbers.

The above sounds like a good "excuse" behind our informality, but what does this do to our goal: to have a ***theory*** of computable functions?

It turns out that no harm is done to the theory, (by our informal approach), because of "*Church's Thesis*".

In the mid thirties many different formalisms, proposed by Turing, Post, Kleene, Markov, Church and others, attempted to capture the concept of *computable (partial) number theoretic function.*

It turned out that all these, *seemingly different approaches to the problem*, ended up with ***the same set*** $\mathcal{P}$ ***of functions***!

Moreover, at the present state of our understanding the concept of "algorithm" or "algorithmic process", ***there is no way known*** of how to define "intuitively computable" functions ***outside of*** $\mathcal{P}$.

$\mathcal{P}$ appears to be the ***largest***—i.e., most inclusive—set of "intuitively computable" functions known.

This "empirical" evidence led Church to formulate his now famous ***belief***, known as "Church's Thesis", that

Church's Thesis: The intuitive (informal) notions of "algorithm" and "computable" (partial) function are *totally captured* by the *formal concepts* of TM, and member of $\mathcal{P}$ respectively.

That is,

(1) *any* algorithmic procedure, that is described *intuitively*, *can* be implemented on some appropriate TM, and

(2) *any* algorithmic (number theoretic) function is in $\mathcal{P}$. By "algorithmic function" (also called "intuitively computable") we mean one for which we have informally described an algorithm that computes it.

Church's Thesis is not a theorem. It can never be, as it "connects" precise concepts (TM, $\mathcal{P}$) with imprecise ones ("algorithm", "computable function").

It is simply a belief that has overwhelming empirical backing, and should be only read as an ***encouragement to present algorithms in "pseudo-code"—that is, informally.*** Thus, Church's Thesis (indirectly) suggests that we concentrate in the essence of things, in the high-level design of algorithms, and leave "coding" to TM-programmers.†

Since we are interested in the essence of things in this note, and also promised to make it user-friendly, we will heavily rely on Church's Thesis here (in short "CT") to "validate" our "high-level programs".

In the literature, Rogers ([3], a very advanced book) heavily relies on CT. On the other hand, [1, 5] never use CT, and give all the necessary constructions (implementations) in their full gory details—that is the price to pay, if you avoid CT.

We conclude this section with a useful tool, Kleene's "indexing", or "parametrization", or "S-m-n" theorem.

4.1 Theorem. (Parametrization theorem) *There is a function $\lambda ex.h(e,x)$ in $\mathcal{R}$, such that, for any $n > 0$,*

$$\phi_e^{(n+1)}(x, \vec{y}_n) = \phi_{h(e,x)}^{(n)}(\vec{y}_n), \text{ for all } e, x, \vec{y}_n \tag{1}$$

Proof. We need an algorithm for h:

input: e, x

processing:

1. Starting at state q_0, generate the number x on tape, as "$1^{x+1}0$", immediately to the left of whatever input already appeared on tape. Halt at the leftmost square of "$1^{x+1}0$".

   (**Comment.** This, in effect, adds x to the input-list, at the left, properly delimited from the rest of the input by the symbol "0".)

2. Go down the standard list of TMs until you find M_e.

3. Add the instructions of M_e to those for item 1 above, appropriately renaming states, so that M_e can continue processing (starting with its own start state) from exactly where the TM-fragment of item 1 halted. "q_0" for the thus combined machine is, of course, that of item 1.

4. Find the combined machine of item 3 in the standard list. Say it occurs, for the first time, as M_i

output: output i, from item 4 above, and halt.

The above algorithm computes a total function (it always halts). By CT, this function, that we have decided to call h, is TM-computable.

Since it is total, it is in $\mathcal{R}$.

†If ever in doubt about the legitimacy of a piece of "high-level pseudo-code", then you ought to try to implement it in detail, as a TM, or, at least, as a "real" C-program!

By definition (3.3),

$$M_i \text{ computes } \phi_i^{(n)}(\vec{y}_n) \tag{2}$$

for all $\vec{y}_n$.

What is the output?

Well, by the way i was constructed (item 4 above), M_i—on input $\vec{y}_n$—first adds the number x on tape, and then "calls" M_e with input what is on tape, that is, with input

$$1^{x+1}01^{y_1+1}0\cdots 1^{y_n+1}$$

But M_e on that input answers (if/when it halts)

$$\phi_e^{(n+1)}(x, \vec{y}_n) \tag{3}$$

Thus, M_i answers with the result (3) above, and hence, by (2), we have (1) if we remember that $i = h(e, x)$. $\square$

5. "Problems" Halting Problem

First off, a revelation: "Recursion theorists" (as those who work in Computability call themselves) use the terminology *Predicate* synonymously with the terminology *Relation*. Either means ***a set of n-tuples***. We often write

$$R(\vec{a}_n)$$

as "short" for

$$\langle a_1, \ldots, a_n \rangle \in R$$

Relations with $n = 2$ are called binary, and rather than, say,

$$< (a, b)$$

we write, in "infix",

$$a < b \tag{1}$$

As with functions, one needs to distinguish the relation (name), say, "$<$", from a statement (that may be true or false) that a and b are related, e.g., as in (1) above.

Yet, by abuse of notation and terminology people most often say "relation $a < b$" rather than "relation $<$".

We will allow ourselves the same type of "freedom" in jargon.

Let us revisit the definition of "decidable language" (known to us from our text, Sipser).

5.1 Definition. (Recursive or Decidable relations) "*A relation $Q(\vec{x}_n)$ is* ***recursive****, 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*** recursive relations we denote by $\mathcal{R}_*$.

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

Thus, "a relation $Q(\vec{x}_n)$ is recursive" means that some TM computes c_Q.

But that means that some TM 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 $\neg Q(\vec{x}_n)$).

If we translate "0" to "yes", and "1" to "no", this TM "decides" membership in Q.

Sipser has (almost) exactly the same definition, except that his yes/no is not by printout, but by colour-coded "flashing-lights" (accept/reject states).†

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

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

5.3 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 recursive relation (synonymously, predicate, set). □

Here is the most famous unsolvable problem:

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

A different formulation uses the set

$$H = \{x : \phi_x(x) \downarrow\}^{\ddagger} \tag{2}$$

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

†OK, Sipser deals with strings, we deal with numbers. But the two approaches are equivalent, via coding. We already talked about that.

‡Both [3, 5] use K instead of H, but this notation is by no means standard. Thus, I felt free to use "H" here for *H*alting.

H we shall call the "**halting set**", and (1) we shall the "**halting problem**".
Clearly, (1) is equivalent to

$$x \in H$$

5.4 Theorem. *The halting problem is unsolvable.*

Proof. We show, **by contradiction**, that $H \notin \mathcal{R}_*$.
Thus we start by assuming the opposite.

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

that is,

$$c_H \in \mathcal{R} \tag{4}$$

Define the function d which at each x behaves *differently* from $\phi_x(x)$ (with respect to the property of being defined):

$$d(x) = \begin{cases} \downarrow & \text{if } \phi_x(x) \uparrow \\ \uparrow & \text{if } \phi_x(x) \downarrow \end{cases} \tag{5}$$

Well, we need a tiny bit of extra work: We have not defined d's output in the top case yet, for "$\downarrow$" means "defined", and that could be anything: 0, or 5, or 42, or x^2, etc.

So, our final version for d is (this version also uses (4), replacing the original conditions in the definition-by-cases by "calls" to a TM that computes c_H):

$$d(x) = \begin{cases} 42 & \text{if } c_H(x) = 1 \\ \uparrow & \text{if } c_H(x) = 0 \end{cases} \tag{6}$$

We want to show that—**under assumption (4)**—$d \in \mathcal{P}$.

It suffices to give an informal algorithm—we do so in pseudo-C-like code below—and then invoke CT:

proc $d(x)$
{
if $c_H(x) = 1$ **then return**(42)

while (1) /* A deliberate "infinite loop" for the "else" case */
  {
  }
}

Well, the above computes d. By CT, there is a TM that does so too, let it be M_e. Thus,

$$d = \phi_e \tag{7}$$

Let us figure out the response of d to input e:

It is easier to work with (5) than (6) below:

$$\phi_e(e)\downarrow \underset{\text{By (7)}}{\equiv} d(e)\downarrow \underset{\text{By (5)}}{\equiv} \phi_e(e)\uparrow^{\dagger}$$

A glorious (!) contradiction. Thus, (4) (and hence (3)) is false. We are done. □

6. More on unsolvability. Reducibility. Semi-decidable relations

Here we will explore a couple more undecidable problems by a "reducibility" technique. We already know that the halting problem is unsolvable (i.e., H is not recursive).

Suppose we are studying a new problem

$$x \in A \tag{1}$$

If we can show that (1) is "at least as hard as" "$x \in H$", knowing that the latter is unsolvable should make (1) unsolvable too.

More precisely, the reducibility argument (showing that a problem is "at least as hard as" "$x \in H$") is this:

Suppose I can show that existence of a TM, M, that solves (1) can assist me to write a program—using M as a subroutine (subprogram)—to decide $x \in H$.

I can conclude then that such an M does not exist—and therefore $A \notin \mathcal{R}_*$ ((1) says that "$x \in A$" is unsolvable)—since otherwise CT would give me a TM to solve $x \in H$.

Let us also define for the benefit of this section *semi-recursive* or *semi-decidable* sets (relations, predicates):

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

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

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

$$Q(\vec{x}_n) \equiv f(\vec{x}_n)\downarrow \tag{2}$$

Sipser [4] calls these relations ***recognizable***. Some others call them ***recursively enumerable***. □

[†] You remember that I said that d is chosen to behave oppositely than ϕ_x with respect to being defined? Here is where this helped!

6.2 Remark. Yet another way to say (2) is:

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

$$Q(\vec{x}_n) \equiv \phi_e^{(n)}(\vec{x}_n) \downarrow \tag{3}$$

Recall Corollary 3.4!

More unsolvability.

6.3 Theorem. *The problem $x \in A$, where*

$$A = \{x : \phi_x \text{ is a constant function}\}$$

is unsolvable.

Proof. As we know, this statement asks us to prove that $A \notin \mathcal{R}_*$. The steps are standard, so we might as well enumerate them,

(i) We assume the opposite:

$$A \in \mathcal{R}_* \tag{4}$$

(ii)

The task: Using (4) solve: "$a \in H$", for the arbitrary $a \in \mathbb{N}$

(iii) (This step is "***Art***"[†]—all the rest is "Science".) Define a function f by

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

(iv) (Argue that f of the previous step is intuitively computable.) Well, here is a convincing pseudo-program:

proc $f(a, x)$

{

if $x = 0$ **then**

{

call $\phi_a(a)$ /* That is, find TM M_a, and then call it with input a */

if M_a halts as called, **then return**(0) /* Otherwise "loop forever" */

}

else return(0)

}

[†]Specifically, the part of knowing what is a good choice for f. This only experience (= a lot of practice) can help us do with ease. So what's new!

(v) By CT, step (iv) gave us a partial (Turing-)computable function $\lambda ax.f(a,x)$. By Corollary 3.4,

$$f(a,x) = \phi^{(2)}_e(a,x), \text{ for some } \textit{fixed } e \text{ and all } a, x$$[†]

(vi) By the parametrization theorem (4.1), there is a **total**[‡] recursive h such that

$$f(a,x) = \phi^{(2)}_e(a,x) = \phi_{h(a)}(x)$$

Hey! What happened to "e"? Why not "$h(e,a)$"?

(vii) By (4), and CT, since $h \in \mathcal{R}$, I can solve (by a TM) the problem

$$h(a) \in A \tag{5}$$

Indeed, given a, we first compute $h(a)$. Next, we call a TM (that (4) guarantees to exist) that solves "$x \in A$". For input "x" we provide the number $h(a)$ that we have just computed.

(viii) Does (5) solve something it should not? Well, if $h(a) \in A$, then $\phi_{h(a)}$ is a constant function. That *necessitates* that $\phi_a(a)\downarrow$ (See item (iii)!) If now $h(a) \notin A$, this means that $\phi_{h(a)}$ is NOT a constant. This *necessitates* that $\phi_a(a)\uparrow$ (See item (iii) again!) Thus, (5) is equivalent to

$$a \in H$$

Hey! We have just solved the halting problem! This being impossible, we reject (4).

□

6.4 Remark. The great degree of pedantry above was meant to do three things:

(1) Describe all steps of a reduction argument in detail in this first encounter of such arguments.

(2) Emphasize the routine nature of all steps, except the (iii)rd. This involves some fruitful speculation every time, to come up with an f that works.

(3) A reducibility argument really just does this in the end:

We find a **recursive** h such that

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

This "reduces" the problem at the left to the the problem at the right: If I can solve the latter, then I can solve the former. We say that "$x \in A$ is more unsolvable than $x \in H$". We often write this as

$$H \leq A$$

[†]Why "fixed"?

[‡]Shouting, just for emphasis.

This "reducibility" can be defined for any two sets A and B. One can define

$$A \leq B \text{ iff, for some } h \in \mathcal{R}, x \in A \equiv h(x) \in B$$

(4) One last thing: In (iii) of the proof we wrote "$0 \cdot \phi_a(a)$" and gave the semantics (compare with the pseudo-C implementation) that, essentially, "$0 \cdot \uparrow = \uparrow$". The same is true if any input of a total function is undefined: the result is undefined (e.g., also "$0 + \uparrow = \uparrow$"). This is the usual "call-by-value" semantics we use in the theory. That is, all the inputs are evaluated before used. Thus, if one (or more) is undefined, we are stuck.

6.5 Theorem. *The set $E = \{\langle x, y\rangle : \phi_x = \phi_y\}$ is not recursive.*

We say that "the equivalence (of programs) problem, namely, '$\langle x, y\rangle \in E$', is unsolvable".

Proof. Suppose that

$$\langle x, y\rangle \in E \text{ is solvable} \tag{1}$$

Then a TM M solves it.

Using M we will now solve

$$a \in H \tag{2}$$

To this end define f as in Theorem 6.3, and obtain the same recursive function h, as it was done there.

Consider also the constant function

$$g = \lambda x.0$$

As all constant functions are computable (really?), let $g = \phi_r$ for some fixed r.

Since we can solve (1), we feed the pair $h(a)$ and r to machine M. Thus we can solve

$$\langle h(a), r\rangle \in E$$

But this is equivalent to (2)—impossible!—since $\phi_{h(a)} = \phi_r$ is fancy for "For any x, $f(a, x)$ always returns 0". But this is so iff "$\phi_a(a) \downarrow$". □

More examples.

6.6 Example. Prove that $A = \{x \mid \mathrm{ran}(\phi_x) = \{1, 13\}\}$ is not recursive.

Proof. As always, by contradiction (reduction argument).

Suppose that I have a TM, M, that solves the problem "$x \in A$".

We show that we can then solve "$a \in H$" for arbitrary $a \in \mathbb{N}$.

Define (**this "f" is unrelated to the one used before. I am just running out of letters!**)

$$f(a, x) = \begin{cases} 1 & \text{if } x = 0 \wedge \phi_a(a) \downarrow \\ 13 & \text{if } x = 1 \wedge \phi_a(a) \downarrow \\ \uparrow & \text{in all other cases} \end{cases} \tag{1}$$

f is intuitively computable: Indeed, given the input a, x, if x has value other than 0 or 1, then get into an infinite loop. Otherwise,

(1) go and get machine M_a.

(2) Once found (by Theorem 3.2), run it on input a (i.e., start computing $\phi_a(a)$). If this ever halts, then output 1 and halt if $x = 0$; otherwise output 13 and halt.

By CT, $f \in \mathcal{P}$, thus, for some fixed $i \in \mathbb{N}$

$$f(a, x) = \phi_i^{(2)}(a, x), \text{ for all } a, x \tag{2}$$

By the index theorem (4.1), there is a recursive function k such that

$$f(a, x) = \phi_{k(a)}(x), \text{ for all } a, x \tag{3}$$

Pause. What happened to i?

We now feed this $k(a)$ to the supposed to exist TM M, that solves "$x \in A$".

If the answer is "yes" (i.e., $k(a) \in A$), then this means that $\text{ran}(\phi_{k(a)}) = \{1, 13\}$. Because of (3) and (1), $\phi_a(a) \downarrow$—that is, $a \in H$—in this case. If the answer is "no" (i.e., $k(a) \notin A$) this means that $\text{ran}(\phi_{k(a)}) \neq \{1, 13\}$. Looking at (1) we see that the only alternative is $\text{ran}(\phi_{k(a)}) = \emptyset$. Because of (3) and (1), $\phi_a(a) \uparrow$—that is, $a \notin H$—in this case. So, we have solved "$a \in H$" ! A contradiction. $\square$

6.7 Example. Prove that the problem "$x \in B$", where $B = \{x \mid 0 \in \text{ran}(\phi_x)\}$, is unsolvable.

By contradiction, let

$$x \in B$$

be solvable by some TM M.

We then solve "$a \in H$" using M as a subroutine:

Define (**this "f" is unrelated to the ones used before. I am just running out of letters!**)

$$f(a, x) = \text{ if } x = 0 \text{ then } 0 \cdot \phi_a(a) \text{ else } 1$$

This is clearly algorithmic. By CT, let j be such that $f(a, x) = \phi_j^{(2)}(a, x)$ for all a, x.

By Theorem 4.1, there is a recursive t such that $f(a, x) = \phi_{t(a)}(x)$ for all a, x.

Clearly,

$$\text{ran}(\phi_{t(a)}) = \begin{cases} \{0, 1\} & \text{if } \phi_a(a) \downarrow \\ \{1\} & \text{if } \phi_a(a) \uparrow \end{cases}$$

Thus, feeding $t(a)$ to the TM that solves "$x \in B$", we get "yes" if $\phi_a(a) \downarrow$ and "no" otherwise. We have then solved "$a \in H$". A contradiction. $\square$

Bibliography

[1] M. Davis. *Computability and Unsolvability.* McGraw-Hill, New York, 1958.

[2] J. E. Hopcroft and Ullman J. D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

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

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

[5] G. Tourlakis. *Computability.* Reston Publishing Company, Inc., Reston, Virginia, 1984.