Universal Machines and Simulation

We know now that there are sets which are not computable. But we do not know of any particular ones except the rather peculiar diagonal set defined in the theorem that declared that there are some sets which Turing machines cannot accept. We must remedy this if we are to make anyone believe that noncomputability is a practical issue.

To find some uncomputable sets, we would like to actually build them, not just know they exist. To accomplish this we need formal construction tools and methods. That is what we shall immediately begin working upon. At the same time, we shall expand our arsenal of theoretical tools and discover more about the nature of computation.

The first thing we must learn how to do formally is very simple and called substitution. Suppose we have the Turing machine $M_k$ which computes:

$$M_k(x, a, b) = ax + b$$

and we wish to change it into a machine which always computes $6x + 57$. What we need to do is substitute the integer 6 for the variable $a$ and 57 for $b$. Or, suppose we changed our mind and wanted to compute $26.5x - 3$ instead. That is straightforward too. But it would be nice if there was a general procedure for modifying our $ax + b$ machine to produce another where we have substituted for $a$ and $b$ with particular values we select. And, after we choose values for $a$ and $b$, we would like to have the new machine built automatically. Furthermore, we want to know how it is done. In other words, the process must be effective.

Another way to formulate this is to ask:

‘Given a machine $M_k(x, a, b)$ which computes $ax + b$ and specific values $u$ and $v$ to be substituted for $a$ and $b$, is there a computable substitution function $s(k, u, v)$ which provides an index for a new machine in the standard enumeration that always computes $ux + v$?’

The two examples of the use of this substitution function $s(k, a, b)$ from our previous $ax + b$ example are:

$$M_{s(k,6,57)}(x) = M_k(x, 6, 57) = 6x + 57$$

$$M_{s(k,26.5,-3)}(x) = M_k(x, 26.5, -3) = 26.5x - e$$

and of course, there are many more.
What the function \( s(k, a, b) \) does is to provide a new machine which performs \( M_k \)'s computation for the specified values of \( a \) and \( b \). Also, note that \( a \) and \( b \) are now fixed and do not appear as input any more in the new machine \( M_s(k,a,b) \).

Actually doing this with a program is very easy. We just remove the variables \( a \) and \( b \) from the header and insert the pair of assignments: \( a=m \) and \( b=n \) at the beginning of the program code.

Theorem 1 precisely states the general case for substitution. (This is also an example of how a very simple concept is sometimes quite difficult to express precisely and formally!)

**Theorem 1 (Substitution).** There is a computable function \( s \) such that for all \( i, x_1, \ldots, x_n, y_1, \ldots, y_m \):

\[
M_{s(i, y_1, \ldots, y_m)}(x_1, \ldots, x_n) = M_i(x_1, \ldots, x_n, y_1, \ldots, y_m)
\]

Proof. We must design an Turing computable algorithm for the substitution function \( s(i, y_1, \ldots, y_m) \) which, when given specific values for \( i \) and \( y_1, \ldots, y_m \) generates the index (or description) of a machine which computes:

\[
M_i(x_1, \ldots, x_n, y_1, \ldots, y_m)
\]

The machine with index \( s(i, y_1, \ldots, y_m) \) operates exactly as described in the following algorithm.

- **Move to the right end of the input** (to the right of \( x_n \))
- **Write values of** \( y_1, \ldots, y_m \) **on the input tape.**
- **Return to the beginning of the input tape** (to the left of \( x_1 \))
- **Commence processing** as \( M_i \)

If we are provided with the instructions for \( M_i \), we know how to write down the Turing machine instructions which accomplish all of the above steps. Thus we know exactly what \( M_{s(i, y_1, \ldots, y_m)} \) looks like. We shall now appeal to Church's Thesis and claim that there is a Turing machine which will can add the instructions for the above preprocessor to \( M_i \).
In fact, the machine can not only generate the instructions for a machine which computes the above algorithm, but also find it in our standard enumeration. (This is clear if we recall our arithmetization of Turing machines and how we can effectively go between machine descriptions and indices in the standard enumeration.)

The machine which locates the index of $M_{s(i,y_1,\ldots,y_m)}$ is exactly the machine which computes $s(i, y_1, \ldots, y_m)$.

This result is well-known in recursive function theory as the s-m-n theorem because the function $s$ usually appears as $s^{m}_n$. It is an extremely useful result and we shall use it almost every time we build Turing machines from others. It is also very important since there are results in recursion theory which state that all nice enumerations of the computable functions must possess an s-m-n or substitution theorem.

So far we have spoken of Turing machines as special purpose devices. Thus they may have seemed more like silicon chips than general-purpose computers. Our belief that Turing machines can compute everything that is computable indicates that they can do what general-purpose computers do. That is, if programmed correctly, they can be operating systems, compilers, servers, or whatever we wish. As a step in this direction we shall examine their ability to simulate each other - just like real computers! So, we shall design what is always called a universal Turing machine. This will also come in handy in the future to prove several strong results about computation.

Let's begin by calling the universal machine $M_u$. It receives the integers $i$ and $x$ as inputs and carries out the computation of $M_i$ on the input $x$. It is actually what we in computer science have always called an interpreter. The formal specification for this machine is:

**Definition.** $M_u$ is a universal Turing machine if and only if for all integers $i$ and $x$: $M_u(i, x) = M_i(x)$.

In our design, we shall give this machine two tapes, one above the machine and one below as in the figure 1 below. The top tape shall be used for performing or interpreting the computation of $M_i(x)$. The bottom tape is used to store the machine description (or the program) for $M_i$ and thus provides all of the information we need to know exactly how the simulation should take place.

The universal Turing machine, $M_u(i, x)$ receives $i$ and $x$ as input on its upper tape and begins its computation reading both tape endmarkers in a configuration which looks like that of figure 1.
Immediately, the universal machine proceeds to execute the following steps.

   a) Write down the description of \( M_i \) on the lower tape.
   b) Copy the input \( x \) at the left of the input (upper) tape.
   c) Compute \( M_i(x) \) on the upper tape.

After step (a) it has written a description of \( M_i \) on its lower tape. This results in a configuration such as the following.

Before continuing, let us examine the description of \( M_i \)'s instructions that must be written on the lower tape. As an example, suppose instructions I73 and I74 of the machine \( M_i \) are:

\[
\begin{array}{cccc}
\text{I73} & 0 & 1 & \text{right same} \\
 & 1 & b & \text{left next} \\
 & b & 0 & \text{right I26} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{I74} & 0 & 1 & \text{left next} \\
 & 1 & b & \text{halt} \\
 & b & b & \text{right I46} \\
\end{array}
\]
The universal machine's lower tape could hold a straightforward transcription of the instruction tables. A fragment of a three-track description tape that holds the above instructions might look like the following.

```
* 1 7 3 * 0 1 → s  * 1 7 4 * 0 1
... * 1 ← n * 1 ...
* 0 → 1 2 6 *
```

Exactly how a Turing machine could have done this is left as an exercise. It is not too difficult to design a procedure for doing this though. Recall the arithmetization process where machines were mapped into decimal numbers? First, Mu could jot down the decimal numbers in order, count how many of these are actually Turing machine encodings, and save the decimal number that is the encoding of the i\textsuperscript{th} one.

Translation from the decimal number description to a tape like that shown above is almost automatic. The portion of the decimal number that contains the above two instructions is just:

\[
\ldots 99013691247920381101099004791259223810111099 \ldots
\]

and this quickly translates into:

\[
\ldots **01→s*1b←n*b0→l26**00←n*1b↓* \ldots
\]

which in turn is easily written on the three-track tape.

(N.B. We need to pause here and note an important assumption that was made when we copied the instructions of Mi on the three-track description tape. We assumed that Mi was a one-tape, one-atrack machine that used only 0, 1, and blank as its symbols. This is allowable because we showed earlier that this class of machines computes everything that is computable and thus is equivalent to any other class of Turing machines, or even programs.

So, if there is a universal machine which simulates these machines, we are simulating the entire class of computable functions.)

At this point the universal machine overprints the integer i and copys x at the left end of the top (or simulation) tape. If we recall that the selection machine which we designed earlier to carry out the operation M(i, x) = x, performing this step is not difficult to imagine. After resetting the tape heads to the left, we arrive at a configuration like that which follows.
At this point the universal machine is ready to begin its simulation of $M_i(x)$. In the sequel we shall look at a simulation example of one step during a computation. Consider the following configuration of our universal machine.

The squares that the universal machine is examining are shaded in blue. This indicates that in the simulation, $M_i$ is reading a 1 and about to execute instruction $I_{73}$.

In the simulation, the universal machine merely applies the instructions written on the lower tape to the data on the top tape. More precisely, a simulation step consists of:

a) Reading the symbol on the top tape.
b) Writing the appropriate symbol on the top tape.
c) Moving the input head (on the top tape).
d) Finding the next instruction on the description (bottom) tape.

Instead of supplying all of the minute details of how this simulation progresses, we shall explain the process with an example of one step and resort once again to our firm belief in Church's thesis to assert that there is indeed a universal Turing machine $M_u$. We should be able easily to write down the instructions of $M_u$, or at least write a program which emulates it.
In figure 2, our universal machine locates the beginning or read portion of the instruction to be executed and finds the line of the instruction (on the description tape) that corresponds to reading a 1 on the top tape. Moving one square over on the description tape, it discovers that it should write a blank, and so it does.

Next \( M_i \) should move and find the next instruction to execute. In figure 3, this is done. The universal machine now moves its simulation tape head one square to the right, finds that it should move the tape head of \( M_i \) to the left and does so. Another square to the right is the command to execute the next instruction (174), so the universal machine moves over to that instruction and prepares to execute it.
Several of the details concerning the actual operation of the universal machine have been omitted. Some of these will emerge later as exercises. So, we shall assume that all of the necessary work has been accomplished and state the famous Universal Turing Machine Theorem without formal proof.

**Theorem 2. There is a universal Turing machine.**

We do however need another note on the above theorem. The universal machine we described above is a two-tape machine with a three-track tape and as such is not included in our standard enumeration. But if we recall the results about multi-tape and multi-track machines being equivalent to ordinary one-tape machines, we know that an equivalent machine exists in our standard enumeration. Thus, with heroic effort we could have built a binary alphabet, one-tape, one-track universal Turing machine. This provides an important corollary to the universal Turing machine theorem.

**Corollary. There is a universal Turing machine in the standard enumeration.**
At various times previously we mentioned that the universal Turing machine and s-m-n theorems were very important and useful. Here at last is an example of how we can use them to prove a very basic closure property of the computable sets.

**Theorem 3.** The class of computable sets is closed under intersection.

**Proof.** Given two arbitrary Turing machines $M_a$ and $M_b$, we must show that there is another machine $M_k$ that accepts exactly what both of the previous machines accept. That is for all $x$:

$$M_k(x) \text{ halts if and only if both } M_a(x) \text{ and } M_b(x) \text{ halt.}$$

or if we recall that the set which $M_a$ accepts is named $W_a$, another way to state this is that:

$$W_k = W_a \cap W_b.$$ 

The algorithm for this is quite simple. Just check to see if $M_a(x)$ and $M_b(x)$ both halt. This can be done with the universal Turing machine. An algorithm for this is:

```
Intersect(x, a, b)
run $M_u(a, x)$, and diverge if $M_u$ diverges
if $M_u(a, x)$ halts then run $M_u(b, x)$
if $M_u(b, x)$ halts then halt (accept)
```

Appealing once more to Church's thesis, we claim that there is a Turing machine which carries out the above algorithm. Thus this machine exists and has a place in our standard enumeration. We shall call this machine $M_{int}$. And, in fact, for all $x$:

$$M_{int}(x, a, b) \text{ halts if and only if both } M_a(x) \text{ and } M_b(x) \text{ halt.}$$

At this point we have a machine with three inputs ($M_{int}$) which halts on the proper $x$'s. But we need a machine with only one input which accepts the correct set. If we recall that the s-m-n theorem states that there is a function $s(int, a, b)$ such that for any integers $a$ and $b$:

$$M_{s(int, a, b)}(x) = M_{int}(x, a, b).$$
we merely need to look at the output of $s(\text{int, a, b})$ and then set the index: $k = s(\text{int, a, b})$.

Thus we have designed a Turing machine $M_k$ which satisfies our requirements and accepts the intersection of the two computable sets $W_a$ and $W_b$. 