# Recursion Theorem

Vassos Hadzilacos

## Motivation

As programmers you know that recursion, the ability of a program to call itself, is a powerful technique. Unfortunately, this feature is not directly available in the Turing machine model: There is no direct way for a Turing machine to obtain its own code and then use that code in its computation. The Recursion Theorem is an extremely useful result which asserts that we can, in fact, do this — albeit indirectly. Informally, it says that when defining a Turing machine $M$ we can imagine that we have available the instruction "$\langle M \rangle :=$ GetSelf", which assigns to $\langle M \rangle$ the code of $M$, the machine being defined, and then allows $M$ to use its code $\langle M \rangle$ in its computation!

For example, consider the following Turing machine, enhanced with this feature. Let's think of the input $x$ as an natural number, although in reality it is a string representing a natural number. This way we can use more common and intuitive notation for applying operations to numbers:

```
    F := on input x:
1       if x = 0 then output 1
2       else
3           ⟨F⟩ := GetSelf
4           a := output of F on input x − 1
5           b := x ∗ a
6           output b
```

This "enhanced" Turing machine uses recursion to compute the factorial of its input. Note that recursion is a form of self-reference, and as we have been seeing since the first lecture, self-reference can be the source of trouble. Later we will see how this ability for self-reference can be exploited to show that certain problems are undecidable or unrecognizable.

## Statement of the Recursion Theorem

The technical statement of the Recursion Theorem is somewhat abstract, so it takes some careful examination to absorb its meaning and understand how it relates to the above stated goal — of allowing a Turing machine to effectively obtain, and use, its own code. First let us establish some notation and conventions that will be useful.

In this document we will be thinking of Turing machines mostly as computers of partial functions, rather than as language recognizers – i.e., computers of partial functions with only binary output. (Note that a function is a special case of a partial function, namely one that is defined on all inputs.) If $M$ is a Turing machine we will use the corresponding lower-case character $m$ to denote the partial function that $M$ computes. That is, $m(x)$ is the output of $M$ on input $x$, if $M$ halts on $x$; if $M$ loops on $x$, $m(x)$ is undefined and we denote this as $m(x) = \perp$.

So far we have been consideringTuring machines as computers of functions that take one input. It will be useful here to also consider Turing machines with two inputs — that is, computers of partial functions with two arguments. This is not really a departure from the single-input Turing machine model since we can encode a pair of strings by a single string. So a Turing machine with two inputs can be thought of as a regular Turing machine that takes a single string as its input, "decodes" this string as a pair of strings,

and treats the first component of the pair as a first input and the second component of the pair as a second input. We can now state the Recursion Theorem:

**Theorem 5.2** (The Recursion Theorem, Kleene 1938): *Let $T$ be a (two-input) Turing machine that computes the partial function $t$: $\Sigma^* \times \Sigma^* \to \Sigma^*$. Think of the first input of $T$ as the code of a Turing machine, encoded by a string in $\Sigma^*$. There exists a (one-input) Turing machine $R$ that computes the partial function $r$: $\Sigma^* \to \Sigma^*$ such that, for every $x \in \Sigma^*$, $r(x) = t(\langle R \rangle, x)$. Furthermore, the mapping $\langle T \rangle \mapsto \langle R \rangle$ is computable.*

Let us unpack this: $T$ is a two-input Turing machine whose first input is to be interpreted as the code of a Turing machine $M$. $T$ describes how $M$ is to be used on $T$'s second input $x$. For example, recall the factorial-computing Turing machine (enhanced with the GetSelf instruction) discussed earlier. Consider the following two-input Turing machine $T$:

```
    T := on inputs ⟨M⟩ and x:
1       if x = 0 then output 1
2       else
3           a := output of M on input x − 1
4           b := x ∗ a
5           output b
6       halt
```

Note that this is a normal Turing machine — it does not use the magic instruction GetSelf. It describes how to use a Turing machine $M$ to manipulate input $x$: It applies $M$ to $x - 1$, multiplies the result of this by $x$, and outputs the product. What this actually does of course depends on $M$: Plugging different Turing machine codes for $\langle M \rangle$ results in $T$ computing different functions. For example,

- If $\langle M \rangle$ is a Turing machine that, on any input $x$, outputs 3, then $T$ computes the function

$$f_1(x) = \begin{cases} 1, & \text{if } x = 0 \\ 3x, & \text{otherwise} \end{cases}$$

- If $\langle M \rangle$ is a Turing machine that, on input $x$, outputs $x + 1$, then $T$ computes the function

$$f_2(x) = \begin{cases} 1, & \text{if } x = 0 \\ x^2, & \text{otherwise} \end{cases}$$

- If $\langle M \rangle$ is a Turing machine that loops on any input $x$, then $T$ computes the partial function

$$f_3(x) = \begin{cases} 1, & \text{if } x = 0 \\ \perp, & \text{otherwise} \end{cases}$$

The Turing machine $R$ of the Recursion Theorem that corresponds to the above Turing machine $T$ is precisely the Turing machine $F$ shown earlier, that on input $x$ outputs $x!$, with the line "$\langle F \rangle :=$ GetSelf" replaced by a construction that we will see in the proof of the Recursion Theorem — specifically, the Turing machine that computes $\langle R \rangle$ from $\langle T \rangle$.

The Turing machine $T$ is not limited to using its first input $\langle M \rangle$ only in the simple manner that the above example illustrates. Here is another example where $\langle M \rangle$ is used multiple times on multiple inputs obtained from $x$:

```
    T' := on inputs ⟨M⟩ and x:
1       if x = 0 or x = 1 then output 1
2       else
3           a := output of M on input x − 1
4           b := output of M on input x − 2
5           c := a + b
6       halt
```

$T'$ uses its first input $\langle M \rangle$ twice: First it applies $M$ on $x - 1$ and then on $x - 2$. It then returns the sum of the results of these two applications of $M$. As an exercise, determine the Turing machine $R$ that corresponds to this $T'$ according to the Recursion Theorem. (It computes a well-known function!)

## Applications

Next we will present three applications of the Recursion Theorem. We will assume that Turing machines have access to the GetSelf instruction to obtain their code, and then use this ability to create machines with self-contradictory behaviour thereby proving, by contradiction, that various problems are undecidable or unrecognizable.

### A. Alternative proof of the undecidability of Univ

Suppose, for contradiction, that Univ is decidable, and let $D_U$ be decider for it. Now consider the following Turing machine:

```
    R := on input x:
1       ⟨R⟩ := GetSelf
2       run D_U on ⟨R, x⟩        ▶ and then do the opposite
3       if D_U accepts then reject
4       else accept
```

We have:

- If $R$ accepts $x$, $D_U$ rejects $\langle R, x \rangle$, which means that $R$ does not accept $x$.
- If $R$ rejects $x$, $D_U$ accepts $\langle R, x \rangle$, which means that $R$ accepts $x$.

Therefore both cases lead to contradiction, which means that the original assumption, namely that Univ is decidable, is false.

### B. Alternative proof of Rice's theorem

Let $P$ be any non-trivial property of recognizable languages. We will prove that $T_P = \{\langle M \rangle : \mathcal{L}(M) \in P\}$ is undecidable.

Suppose, for contradiction, that $T_P$ is decidable, and let $D_P$ be a decider for it. Since $P$ is non-trivial, there are Turing machines $M_Y$ and $M_N$ such that $\mathcal{L}(M_Y) \in P$ and $\mathcal{L}(M_N) \notin P$. Consider the following Turing machine:

```
    R := on input x:
1       ⟨R⟩ := GetSelf
2       run D_P on R
3       if D_P accepts ⟨R⟩ then      ▶ behave like M_N
4           run M_N on x
5           if M_N accepts then accept
6           else reject
7       else                         ▶ behave like M_Y
8           run M_Y on x
9           if M_Y accepts then accept
10          else reject
```

We have:

$$\mathcal{L}(R) = \begin{cases} \mathcal{L}(M_N) \notin P, & \text{if } D_P \text{ accepts } \langle R \rangle \Leftrightarrow \mathcal{L}(R) \in P \\ \mathcal{L}(M_Y) \in P, & \text{if } D_P \text{ does not accept } \langle R \rangle \Leftrightarrow \mathcal{L}(R) \notin P \end{cases}$$

Therefore $\mathcal{L}(R) \in P$ if and only if $\mathcal{L}(R) \notin P$, which is a contradiction. So the original assumption, namely that $T_P$ is decidable, is false.

### C. A new unrecognizability result

For each recognizable language $L$ there are infinitely many Turing machines that recognize it: we can add any number of useless states or symbols to the description of a machine without changing its essential functionality. If we order the machines that recognize $L$ by the length of their encoding, the shortest ones are called ***minimal*** for $L$. Let MIN be the set of codes of minimal Turing machines; that is,

$$\text{MIN} = \{\langle M \rangle : \text{for every Turing machine } M' \text{ such that } \mathcal{L}(M') = \mathcal{L}(M),\ |\langle M' \rangle| \geq |\langle M \rangle|.$$

**Theorem 5.3:** MIN *is unrecognizable.*

PROOF. Suppose, for contradiction, that MIN is recognizable. Then there is an enumerator $E_{\text{MIN}}$ for MIN (see Week 3 Tutorial, Question 1). Consider the following Turing machine:

```
    R := on input x:
1       ⟨R⟩ := GETSELF
2       repeat
3           ⟨M⟩ := next element of MIN output by E_MIN
4       until |⟨M⟩| > |⟨R⟩|
5       ▶ Behave like M
6       run M on x
7       if M accepts then accept
8       else reject
```

The loop in lines 2-4 will terminate because $E_{\text{MIN}}$ has an infinite number of strings to output (since there are infinitely many recognizable languages), so it must output arbitrarily long Turing machine codes. So by lines 6-8 $\mathcal{L}(R) = \mathcal{L}(M)$. By the exit condition of the loop $|\langle M \rangle| > |\langle R \rangle|$, so $\langle M \rangle \notin$ MIN. This contradicts the fact that $\langle M \rangle$ was output by an enumerator for MIN. □

Note that the above proof actually shows a stronger result than the statement of Theorem 5.3: Not only is MIN unrecognizable, but so is every infinite subset of it!

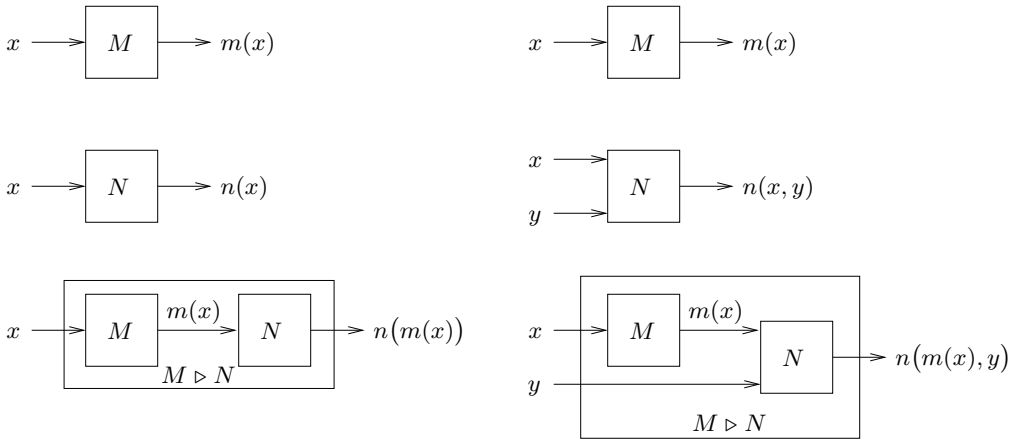
## Some preliminary results

If $M$ and $N$ are Turing machines, $M \triangleright N$ denotes the Turing machine that "pipes" $M$'s output to $N$'s input (see Figure 1(a)). That is, $M \triangleright N$ first runs $M$ on its input and then, if $M$ halts, it runs $N$ on $M$'s output. Thus, on input $x$, $M \triangleright N$ computes the partial function $n\big(m(x)\big)$.

If $N$ is a two-input Turing machine, $M \triangleright N$ denotes the two-input Turing machine that "pipes" $M$'s output to $N$'s first input (see Figure 1(b)): $M \triangleright N$ first runs $M$ on its first input; if and when $M$ halts, $M \triangleright N$ runs $N$ using $M$'s output as the first input and its second input as $N$'s second input. Thus, in this case, on inputs $x$ and $y$, $M \triangleright N$ computes the two-argument partial function $n\big(m(x), y\big)$.

**Lemma 5.4:** *There is a two-input Turing machine* PIPE *that takes inputs* $\langle M \rangle$ *and* $\langle N \rangle$*, and outputs* $\langle M \triangleright N \rangle$.

PROOF. Consider the case where $N$ is a one-input Turing machine. The initial state of $M \triangleright N$ is the initial state of $M$. Each state transition of $M$ that leads to the halt state is modified to lead to a special




(a) Piping to 1-input TM

(b) Piping to 2-input TM

Figure 1: The "piping" operation ▷ on TMs

state in which the tape head keeps moving left until it reaches the leftmost cell, at which point $M \triangleright N$ enters the initial state of $N$ and continues the computation from there, halting if and when $N$ does.

The case where $N$ is a two-input Turing machine is similar with some straightforward additional manipulations to save the second input to $M \triangleright N$ so that it can be presented as the second input to $N$ when $M$ completes its computation, the output of which is presented as the first input to $N$.

Given the preceding description of how $M \triangleright N$ operates, it is clear that given $\langle M \rangle$ and $\langle N \rangle$, a Turing machine PIPE can construct $\langle M \triangleright N \rangle$. □

Note that PIPE is not the Turing machine $M \triangleright N$; it is a Turing machine that produces the code of $M \triangleright N$, given the codes of $M$ and $N$.

**Lemma 5.5:** *There is a Turing machine* PRINT *that, on input $x$, outputs $\langle P_x \rangle$, where $P_x$ is a Turing machine that outputs $x$.*

PROOF. PRINT hard-codes its input string $x = a_1 a_2 \ldots a_k$ into the state transition function of the Turing machine $P_x$ whose code it must output. It can do so by defining a set of states that includes a state $q_i$ for each $i \in [1..k+1]$, where, for $i \in [1..k]$, $q_i$ "remembers" that the $i$-th symbol of $x$ is $a_i$ and $q_{k+1}$ is the halt state. The transition function causes $P_x$ to behave as follows: $P_x$ starts by erasing its input and returning the tape head to the leftmost cell; it then enters state $q_1$, and for each $i \in [1..k]$, the transition function of $P_x$ specifies that, if $P_x$ is in state $q_i$ and the current symbol is $\sqcup$, $P_x$ replaces the $\sqcup$ by $a_i$, enters state $q_{i+1}$, and moves to the right. PRINT then returns the code $\langle P_x \rangle$ of the Turing machine $P_x$ described above. □

Note that $P_x$ takes no input, since it starts by erasing whatever is initially on its tape, so it computes a function of zero arguments.

## Warm-up exercise: A Quine

As a preliminary step towards proving the Recursion Theorem we show how to construct a Turing machine that outputs its own code.[1] This is trickier than it may first appear. Note that $P_{\langle M \rangle}$ is not such a machine: its output is $\langle M \rangle$ and not its own code, which is $\langle P_{\langle M \rangle} \rangle$.

[1] A program in any programming language that just prints its own code is called a ***Quine***, after the American philosopher and logician Willard Van Orman Quine who, among other things, was interested in self-reference and the paradoxes to which it leads.

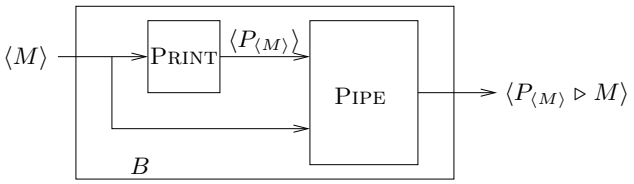



Figure 2: The TM $B$

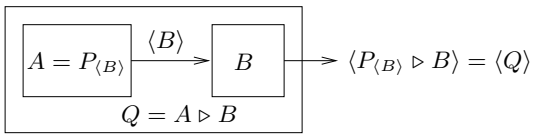



Figure 3: The Quine $Q$

**Theprem 5.6:** *There is a Turing machine $Q$ that (ignores its input and) outputs its own code $\langle Q \rangle$.*

PROOF. Let $B$ be a Turing machine that takes as input $\langle M \rangle$ and outputs $\langle P_{\langle M \rangle} \triangleright M \rangle$. That is, $B$ takes as input the code of a Turing machine $M$, and outputs the code of the Turing machine that runs $M$ on input $\langle M \rangle$. $B$ can be built using the Turing machines PIPE and PRINT of Lemmas 5.4 and 5.5, as follows (also depicted diagrammatically in Figure 2):

$B :=$ **on input** $\langle M \rangle$:
1 $a :=$ output of PRINT on input $\langle M \rangle$ (i.e., $a = \langle P_{\langle M \rangle} \rangle$)
2 $b :=$ output of PIPE on input $a$ and $\langle M \rangle$ (i.e., $b = \langle P_{\langle M \rangle} \triangleright M \rangle$)
3 **output** $b$
4 **halt**

This Turing machine has a code $\langle B \rangle$ and, by Lemma 5.4, there is a Turing machine $P_{\langle B \rangle}$ that outputs $\langle B \rangle$. Let $A = P_{\langle B \rangle}$.

We claim that $Q = A \triangleright B$ has the desired property: it outputs $\langle Q \rangle$. We have $Q = A \triangleright B = P_{\langle B \rangle} \triangleright B$ (see Figure 3). That is, $Q$ is the Turing machine that runs $B$ on $B$'s own code, $\langle B \rangle$. By the specification of $B$, the output of $B$ on $\langle B \rangle$ is $\langle P_{\langle B \rangle} \triangleright B \rangle = \langle A \triangleright B \rangle = \langle Q \rangle$. That is, the output of $Q$ is $\langle Q \rangle$, as wanted. □

## Proof of the Recursion Theorem

Though a program that prints its own code has a self-referential flavour, it is not quite what we want: The Recursion Theorem does not only say that a program can print its own code, but that it can use that code in arbitrary (computable) ways, as specified by the Turing machine $T$. As we will see, however, the proof of the Recursion Theorem is a somewhat more elaborate version of the proof of Theorem 5.6. For convenience we restate the Recursion Theorem below.

**Theorem 5.2** (The Recursion Theorem, Kleene 1938): *Let $T$ be a (two-input) Turing machine that computes the partial function $t$: $\Sigma^* \times \Sigma^* \to \Sigma^*$. There exists a (one-input) Turing machine $R$ that computes the partial function $r$: $\Sigma^* \to \Sigma^*$ such that, for every $x \in \Sigma^*$, $r(x) = t(\langle R \rangle, x)$. Furthermore, the mapping $\langle T \rangle \mapsto \langle R \rangle$ is computable.*

PROOF OF THEOREM 5.2. Let $B$ be the same Turing machine as in the proof of Theorem 5.6, except that the input $\langle M \rangle$ is interpreted as the code of a two-input Turing machine. As we have seen this simply means that the Turing machine decodes its input string as if it encoded a pair of strings, and we think of the two components of that pair as its two inputs. So, on input the code $\langle M \rangle$ of a two-input Turing machine $M$, $B$ outputs $\langle P_{\langle M \rangle} \triangleright M \rangle$ — that is, the code a Turing machine that outputs the code $\langle M \rangle$ of $M$ and uses it as the first input of $M$.

Next, let $A$ be the Turing machine $P_{\langle B \triangleright T \rangle}$. That is, $A$ is a Turing machine that, without any input, outputs the code of a two-input Turing machine that first runs $B$ on its input and then uses $B$'s output as the first input of $T$.

Finally, let $R = A \triangleright (B \triangleright T)$ — see Figure 4. We claim that this Turing machine has the desired property: It computes the partial function $r$ such that $r(x) = t(\langle R \rangle, x)$. To see this first note that the output of $A$ is $\langle B \triangleright T \rangle$. When we run $B$ on this Turing machine code, by the definition of $B$, its output

is $\langle P_{\langle B \triangleright T\rangle} \triangleright (B \triangleright T)\rangle$, and this becomes the first input of $T$ in $R$. Therefore, on input $x$, $R$ outputs $r(x) = t\big(\langle \underbrace{P_{\langle B \triangleright T\rangle}}_{A} \triangleright (B \triangleright T)\rangle, x\big) = t\big(\langle \underbrace{A \triangleright (B \triangleright T)}_{R}\rangle, x\big) = t(\langle R\rangle, x)$, as wanted.

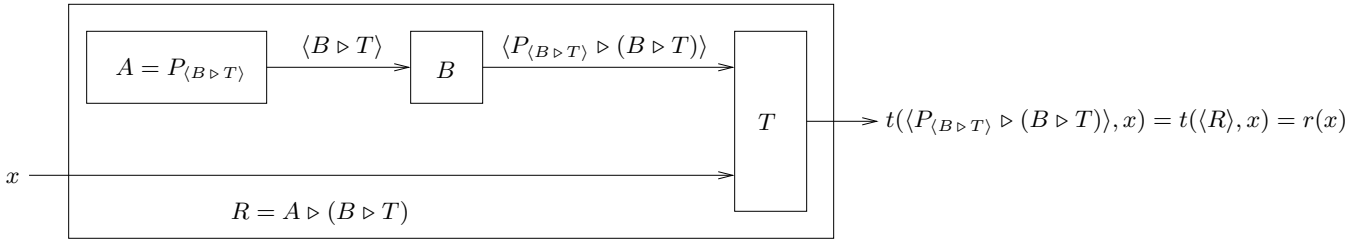



Figure 4: The Turing machine $R$

It remains to show that the mapping $\langle T\rangle \mapsto \langle R\rangle$ is computable; that is, there is a Turing machine RECURSIFY that takes as input $\langle T\rangle$, where $T$ is a two-input Turing machine, and outputs the code $\langle R\rangle$ of the one-input Turing machine $R$ described above. To see this we first note that, by Lemma 5.5, there is a Turing machine, namely $P_{\langle B\rangle}$, that outputs $\langle B\rangle$. Using this machine to produce $\langle B\rangle$, its input $\langle T\rangle$, and the Turing machine PIPE of Lemma 5.4, RECURSIFY can then produce $\langle B \triangleright T\rangle$. Using this as input and the Turing machine PRINT of Lemma 5.5, RECURSIFY can construct $A = P_{\langle B \triangleright T\rangle}$. Finally, using $A$, the previously produced $\langle B \triangleright T\rangle$, and PIPE again, RECURSIFY can output $\langle A \triangleright (B \triangleright T)\rangle$, i.e., $\langle R\rangle$. □