Undecidability

Zeyu Mi

2019-12-13

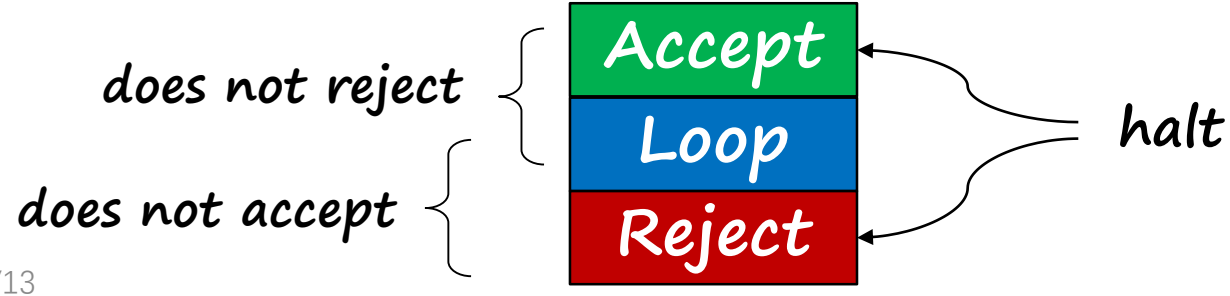
Adapted From:

IALC 9.1 9.2 9.3

Stanford CS103 Undecidability

Review: Very Important Terminology

- Let M be a Turing machine and let s be a string.
  - M **accepts** s if it enters an accept state when run on s
  - M **rejects** s if it enters a reject state when run on s.
  - M **loops infinitely** on s (or just **loops** on w) if when run on s it enters neither an accept nor a reject state.
  - M **does not accept** s if it either rejects s or loops infnitely on s.
  - M **does not reject** s if it either accepts s or loops on s.
  - M **halts** on s if it accepts w or rejects s.



Review: R & RE

- A language $\boldsymbol{L}$ is **recursive enumerable(递归可枚举的)** if there's a TM $\boldsymbol{M}$ such that:
  - If $w \in L, M$ accepts w in finite steps.
  - If $w \notin L, M$ **does not accept** w, it means M rejects w or loop forever
- A language $\boldsymbol{L}$ is **recursive(递归的)** if there's a TM $\boldsymbol{M}$ such that:
  - If $w \in L, M$ accepts w in finite steps.
  - If $w \notin L, M$ **rejects** w in finite steps.
- **R ⊆ RE**

Review: R & RE

- Union/Intersection of two **RE** language is **RE**.
- Union/Intersection of two **R** language is **R**.
- Complements of Language
  - If $\boldsymbol{L}$ is recursive, so is $\overline{\boldsymbol{L}}$
  - If both $\boldsymbol{L}$ and $\overline{\boldsymbol{L}}$ is RE, then $\boldsymbol{L}$ is recursive

Review: Universal Turing Machine

- Encode Turing machine and input with binary
- There's a Universal Turing machine U_{TM} that accepts $\langle M, w \rangle$ if and only if M accepts w
- The universal language $L_u = \{\langle M, w \rangle | M \ accepts \ w\}$ is **RE**

How we investigate computability?

2.1 Discussion on Problems

Part1. Intro & Set theory(I) : Basics & Formal Language.

Part2. Set Theory(II): Axiom system & Cardinality.

Part3. Capture Structures: Binary Relation & Function

2.2 Discussion on Computation

Part1. Turing Machine Basics.

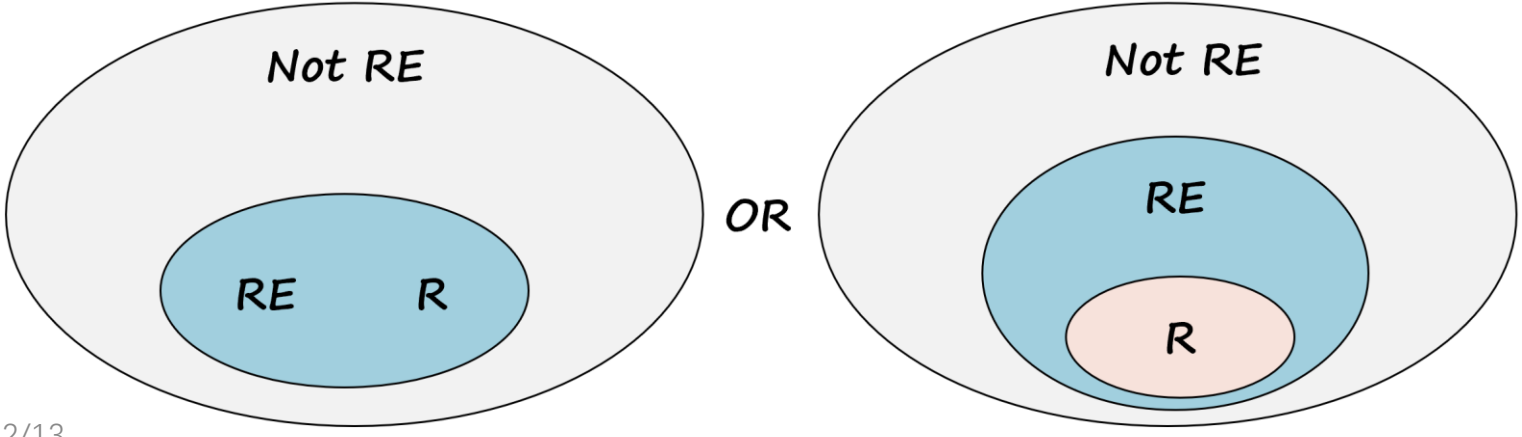
Part2. Variants of Turing Machine. Church-Turing Thesis.

2.3 Discussion on computability

Part1. The Language of Turing Machine. R & RE

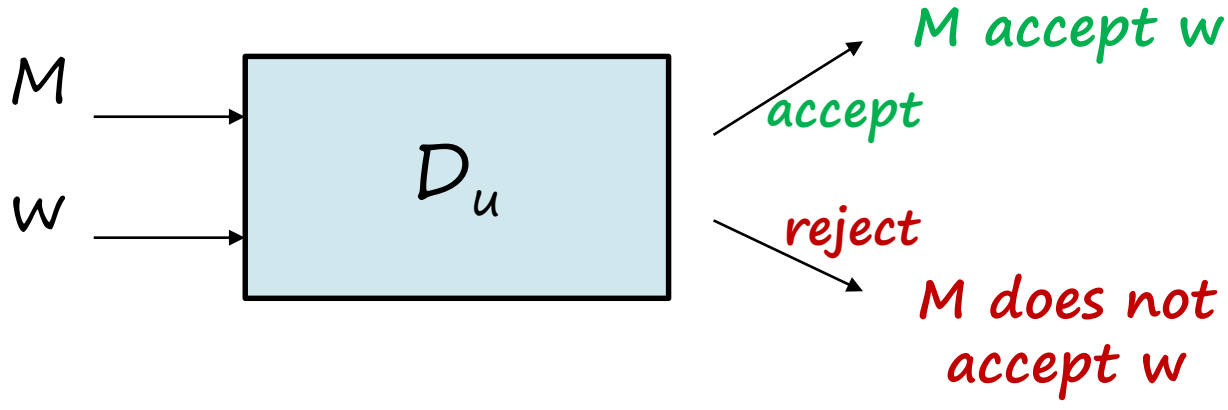
Part2. Undecidability.

Question:
RE=R?



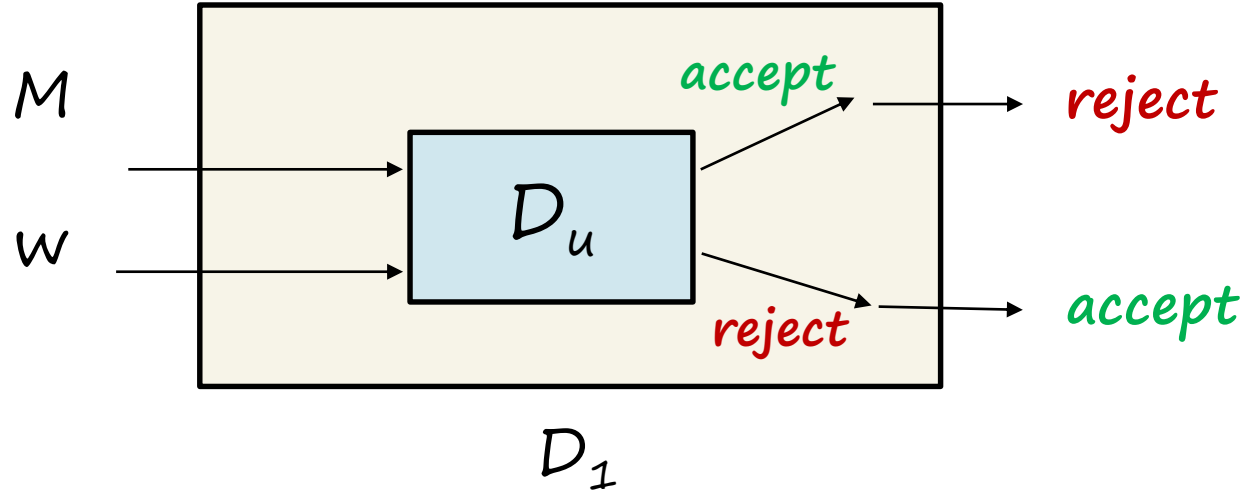
A Decider for L_u

- We know that $L_u \in \mathbf{RE}$, but whether $L_u \in \mathbf{R}$?
- That is, whether there is a decider D_u for L_u?



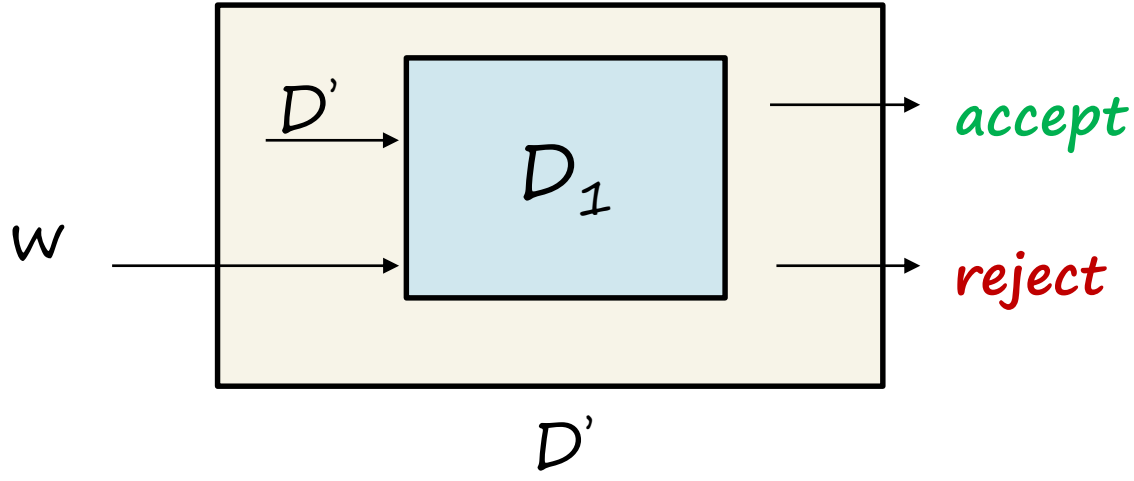
A Decider for L_u

- If D_u exists, we can build a new TM D_1 based on D_u
- D_1 takes a TM M and string w as inputs, and feeds them to D_u, then outputs the **reverse result** of D_u.



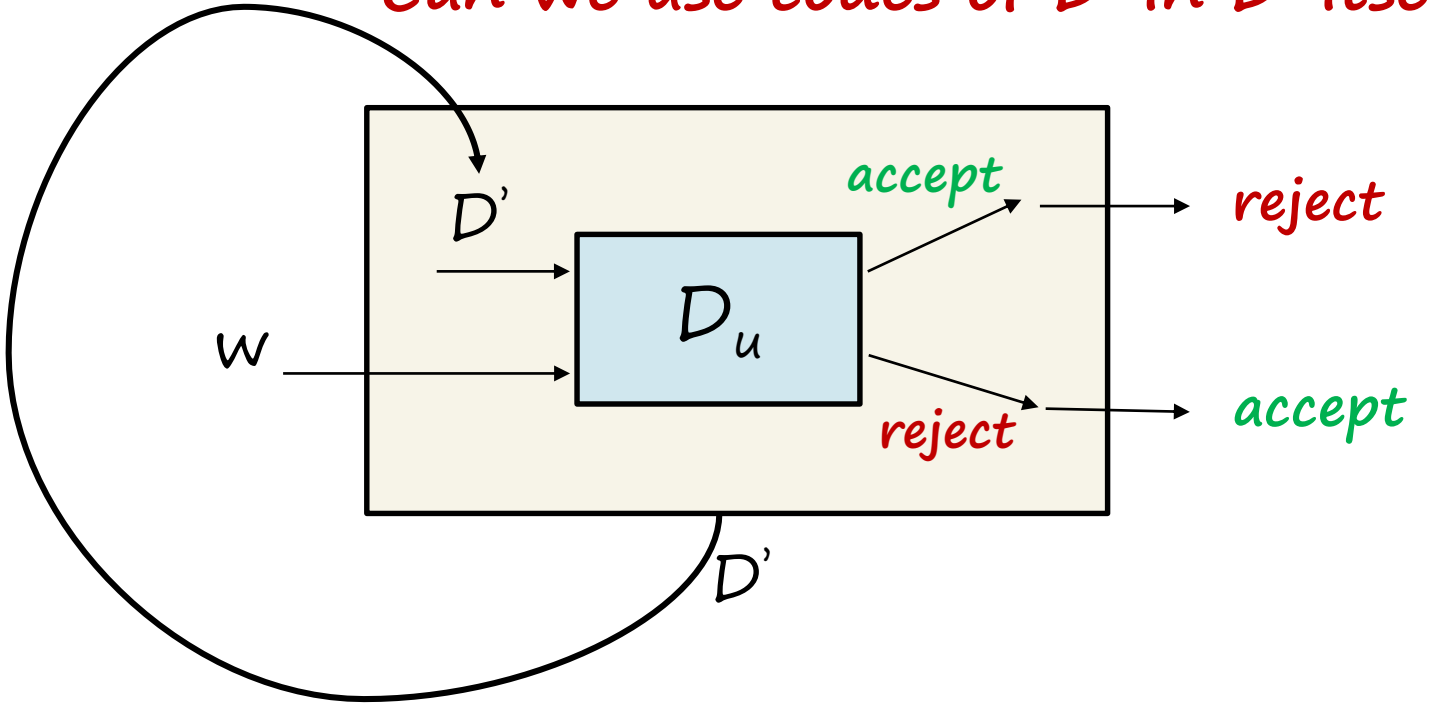
A Decider for L_u

- Then we can build another TM D' based on D_1
- D' only takes a string w as input, and feeds **its own code** and w to D_1, then outputs the result of D_1.

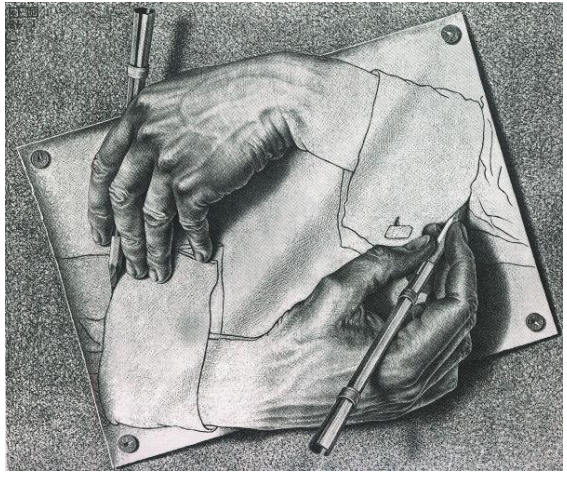


A Decider for L_u

Can we use codes of D' in D' itself?



Self-Reference



Sets that do not contain itself.

"This sentence is wrong."

Self-Referential Software

- A **Quine** is a program that, when running, prints its own source code.
- Quines aren't allowed to just read the file containing their source code and print it out; that's cheating (and technically incorrect if someone changes that file!)
- How would you write such a program?

Example: Quine

```
#include <stdio.h>
char*f="#include <stdio.h>
char*f=%c%s%c;
main(){printf(f,34,f,34,10);}%c";
main(){printf(f,34,f,34,10);}
```

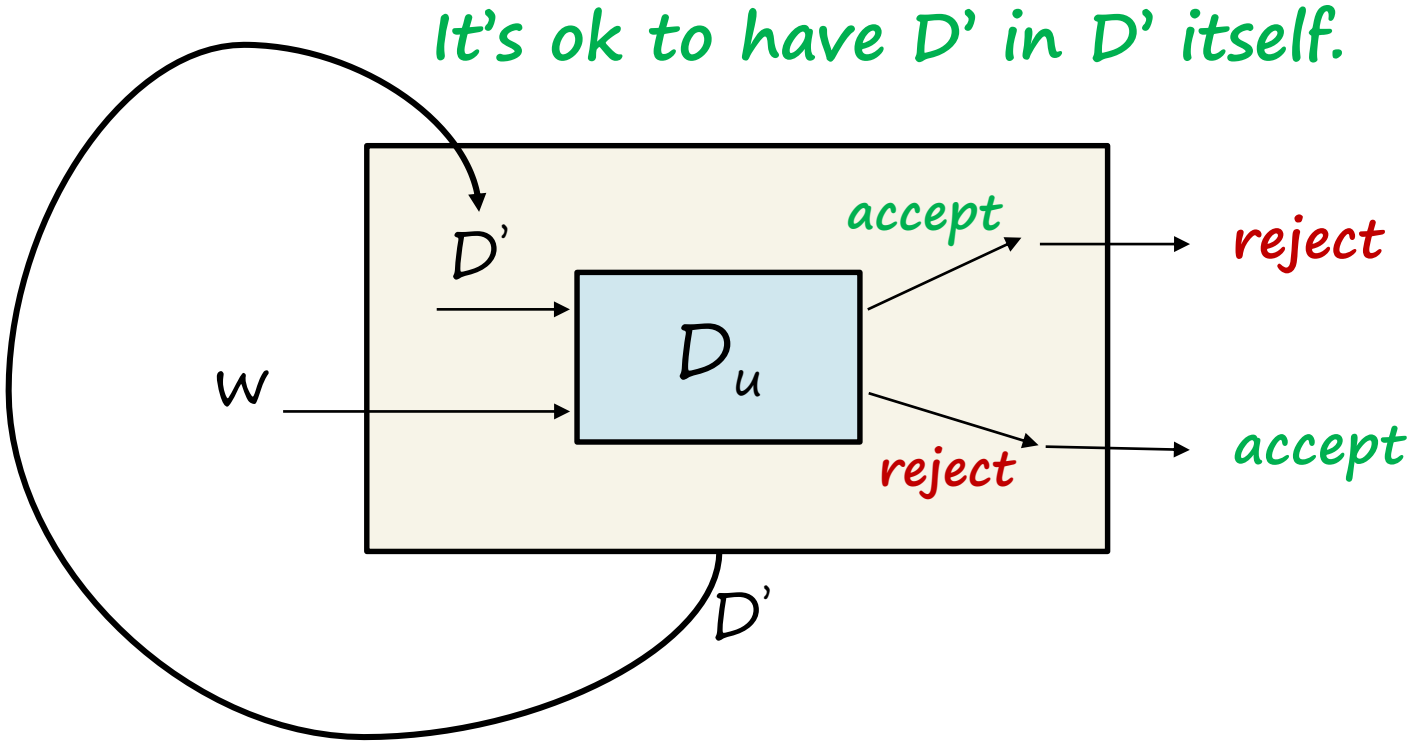
http://www.nyx.net/~gthompso/quine.htm

Self-Referential Software

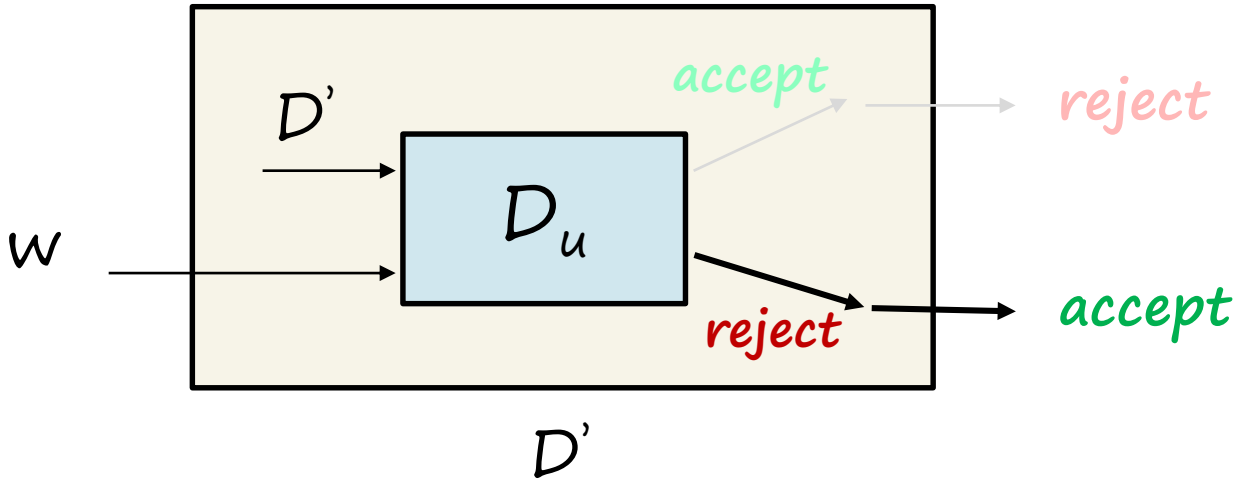
- Going forward, assume that any program can be augmented to include a method called *mySource()* that returns a string representation of its source code.
- Then we can take the whole program codes as an input though we don’t know what the whole program codes are, and even what we write now will change the final code itself.

```
char * mySource(){…}
int main(){
  char * s = mySource();
  ......
}
```

A Decider for L_u

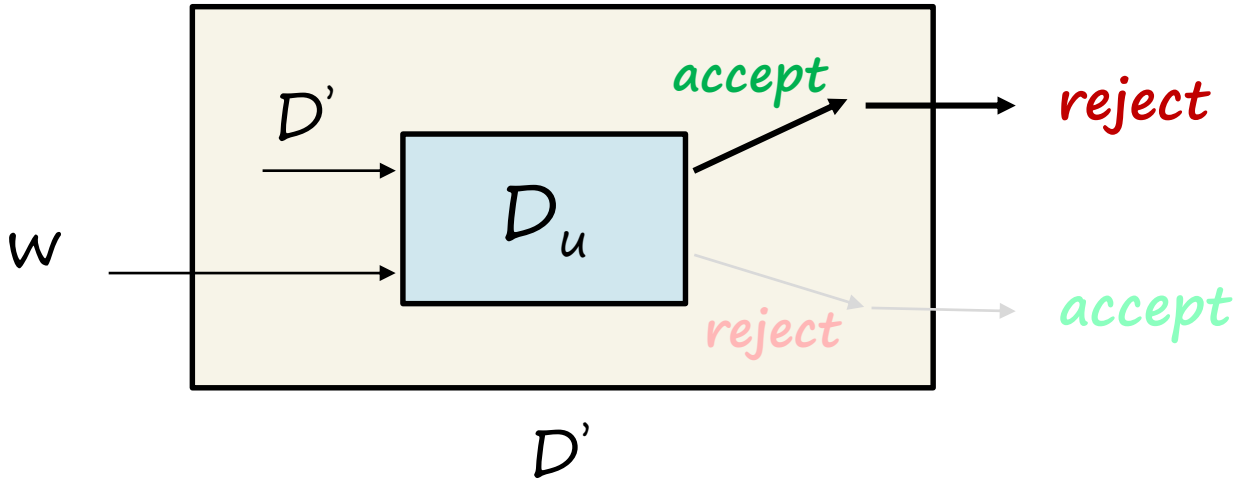


A Decider for L_u



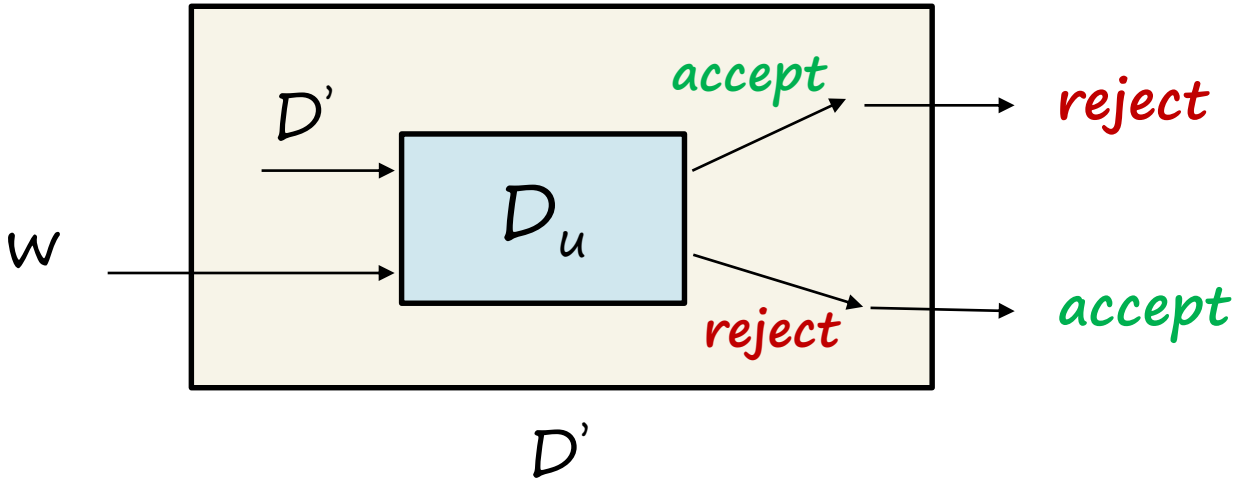
- If D' accepts w.
  - Then D_u rejects $\langle D', w \rangle$, which means that D' rejects w

A Decider for L_u



- If D' accepts w.
  - Then D_u rejects $\langle D', w \rangle$, which means that D' rejects w
- If D' rejects w.
  - Then D_u accepts $\langle D', w \rangle$, which means that D' accepts w

A Decider for L_u



D' accepts $w \Leftrightarrow D'$ rejects w

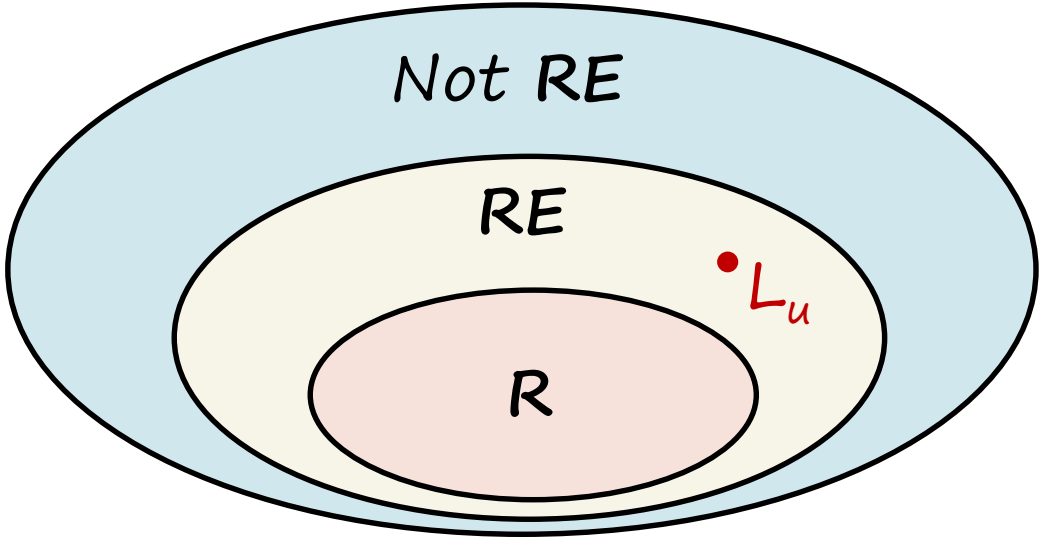
D_u can not exist!

$L_u \notin \mathbf{R}$

- As there are no deciders for L_u, L_u is not in the class of **R.**
- This also tells us that there is **no algorithm** that can determine whether a program will accept an input, or the determination of whether a program will accept an input is **undecidable(不可判定)**.
- Intuition: The only general way to find out what a program will do is to run it.

$\mathbf{R} \neq \mathbf{RE}$

- We found a language $L_u \in \mathbf{RE}$, but $L_u \notin \mathbf{R}$, which means $\mathbf{R} \neq \mathbf{RE}$



R ≠ RE

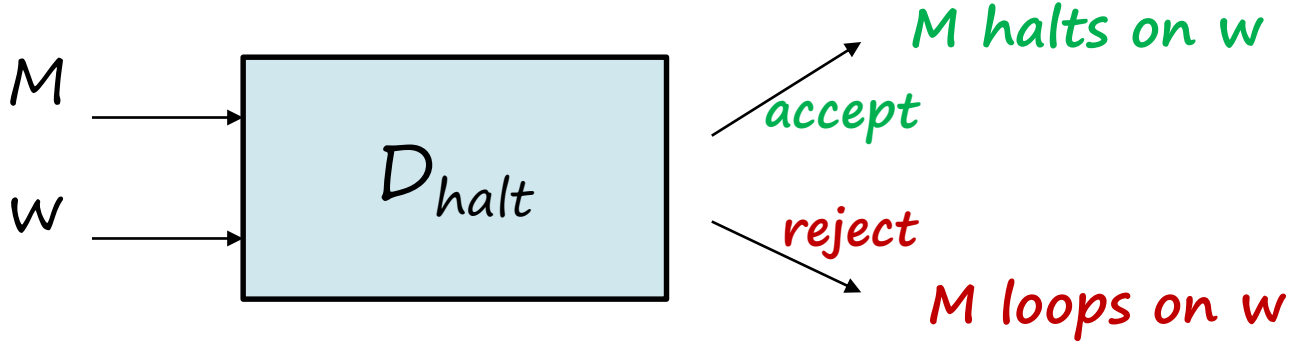
- Because **R ≠ RE**, there is a difference between decidability and recognizability:
- In some sense, it is fundamentally harder to solve a problem than it is to check an answer.

Halting Problem:

Determine whether a program will halt.

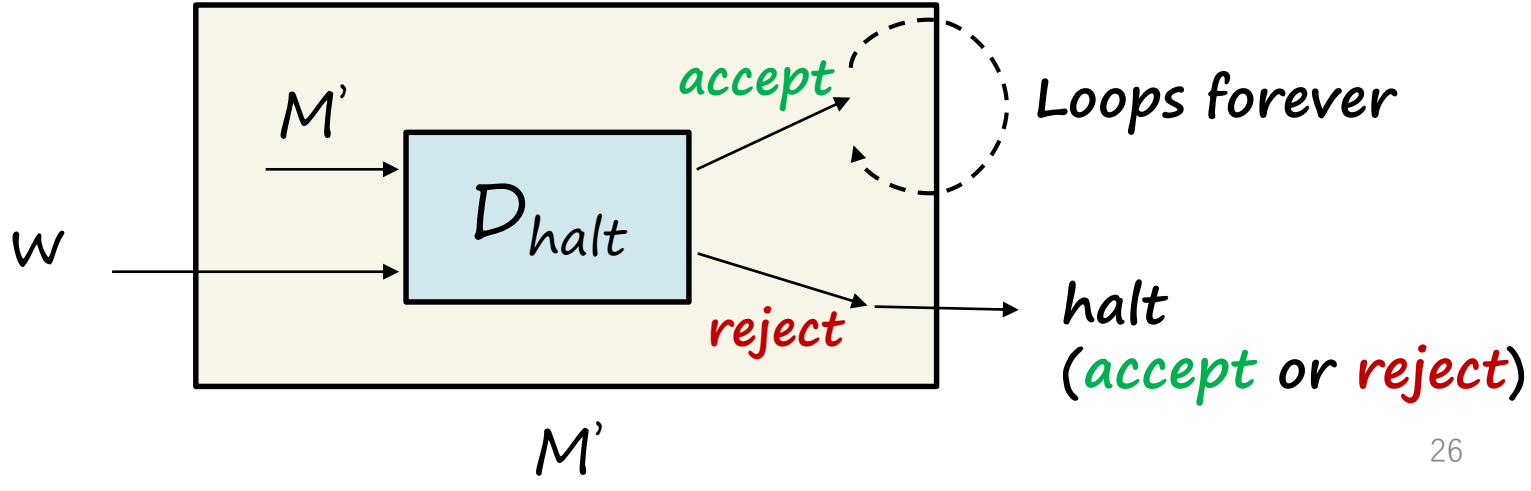
Turing Halting Problem

- Does there exist a TM to **decide** whether a program will finally halt(reject or accept) on a certain input?
- Or whether $L_{halt} = \{\langle M, w\rangle | M \ halts \ on \ w\}$ is in $\mathbf{R}$?

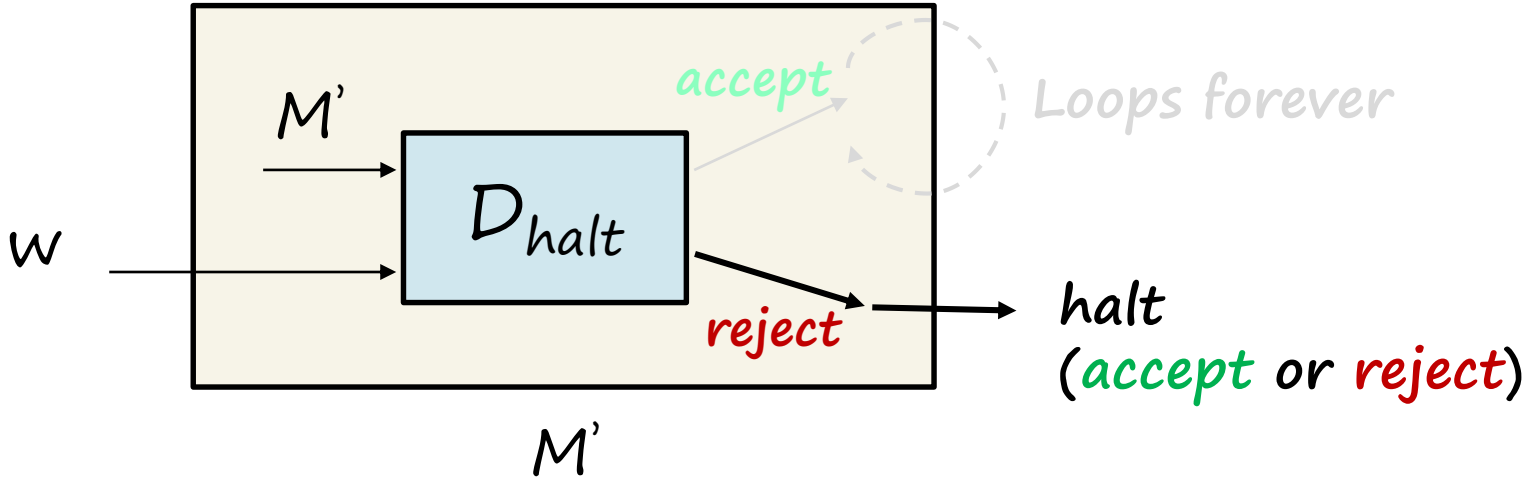


Turing Halting Problem

- Similarly we can build a new TM M' based on D_{halt}.
- M' takes a string w as input, and feed its own code and w to D_{halt}. If D_{halt} rejects the input, then M' halts(accept or reject) on w, otherwise M' loops forever and never stop.

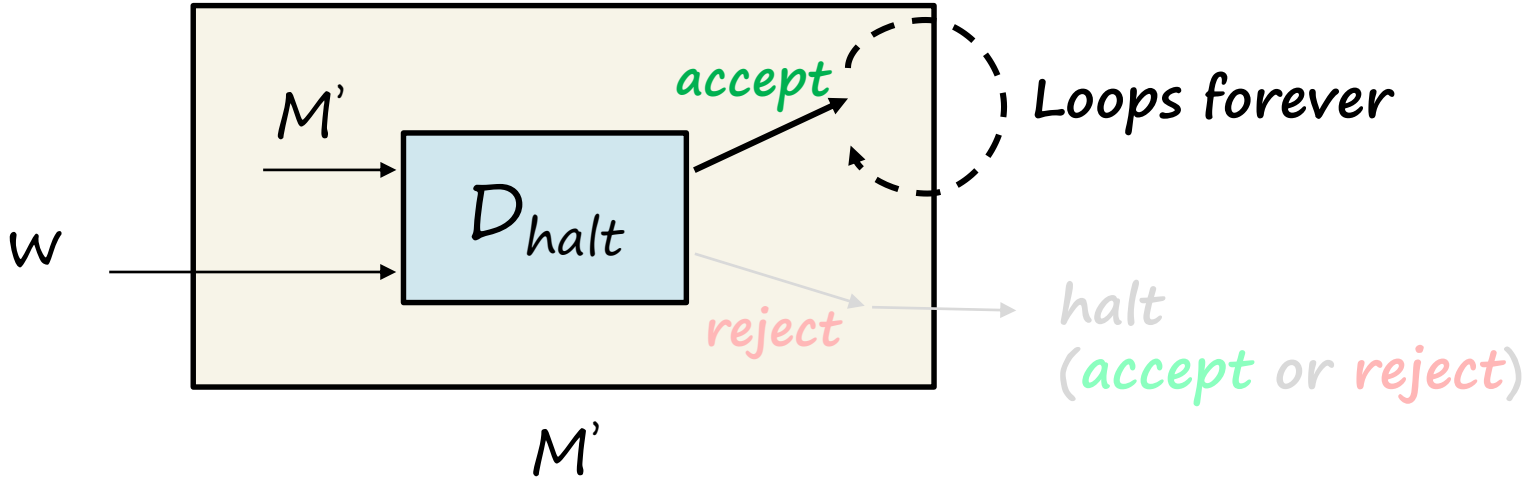


Turing Halting Problem



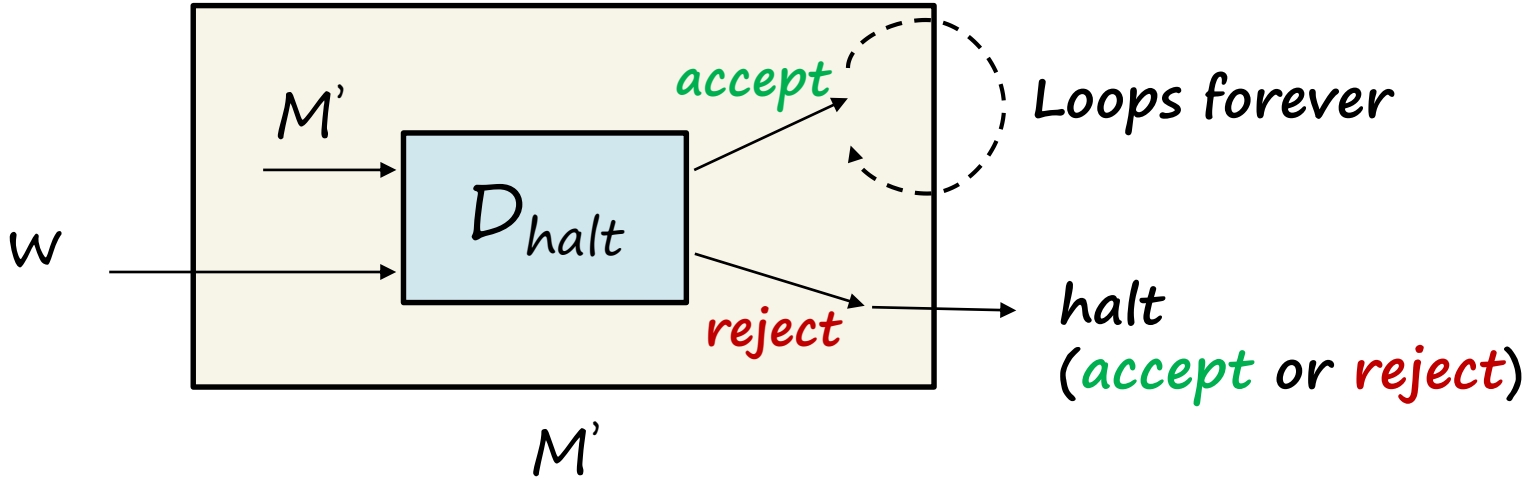
- If M' halts on w
  - Then D_{halt} must reject $\langle M', w \rangle$, which means M' loops on w

Turing Halting Problem



- If M' halts on w
  - Then D_{halt} must reject $\langle M', w \rangle$, which means M' loops on w
- If M' loops on w
  - Then D_{halt} must accept $\langle M', w \rangle$, which means M' halts on w

Turing Halting Problem



M' halts on $w \Leftrightarrow M'$ loops on w

D_{halt} can not exist!

Language That is Non-RE

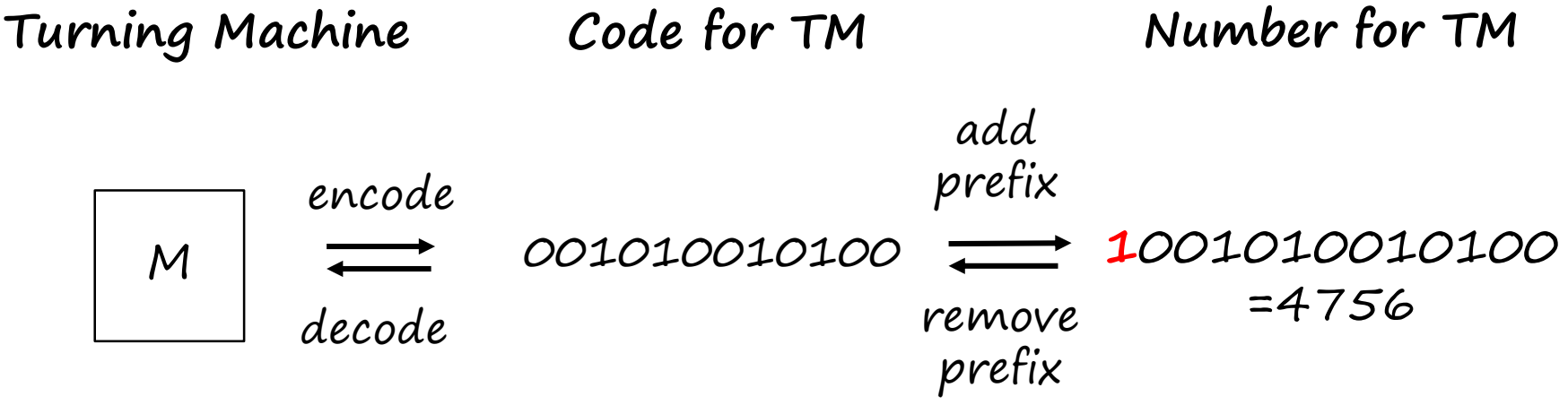
Beyond $\mathbf{RE}$

- The $\mathbf{RE}$ languages represent languages whose membership can be proven
  - If $w \in L$, we can prove that by a TM
  - So what does a non-RE language look like?
- This language cannot be recognized by any Turing machines

Numbering TMs

- Recall that a TM $\boldsymbol{M}$ can be encoded as a binary. In return, a binary can be interpreted as a TM.
  - Some binary may not be interpreted as a TM (illegal format), we interpret that binary as a TM without any transition function and accepting no string
  - TM code begins with 0's, and we can add a prefix 1 to the code of TM as the number of that TM
    - So the numerical values of different TMs are different.

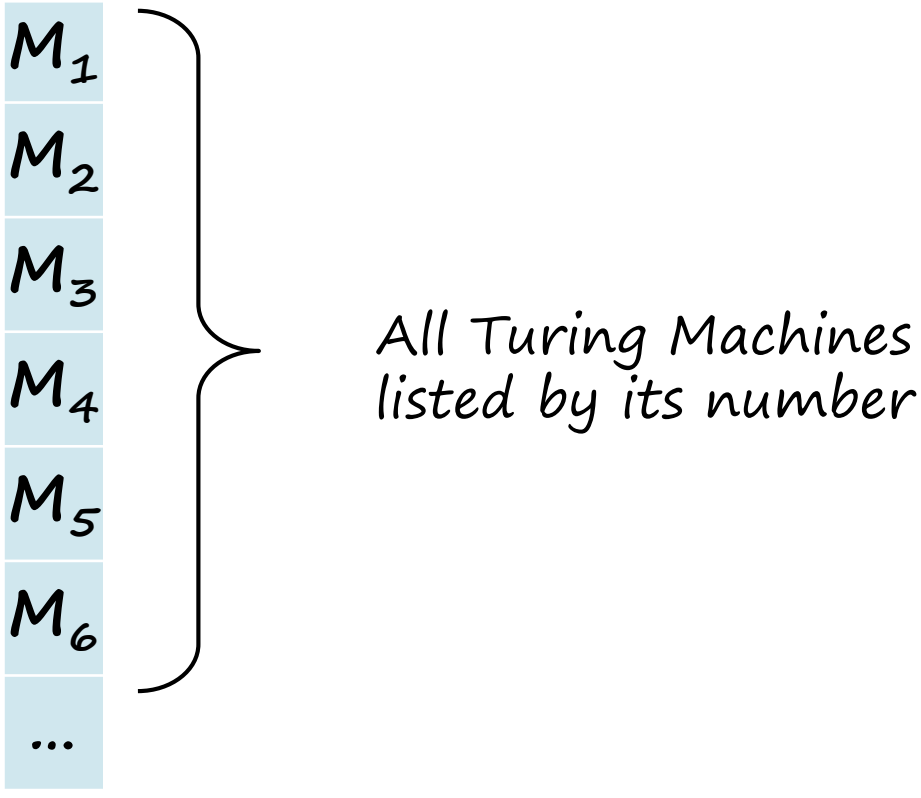
Numbering TMs



M=4756

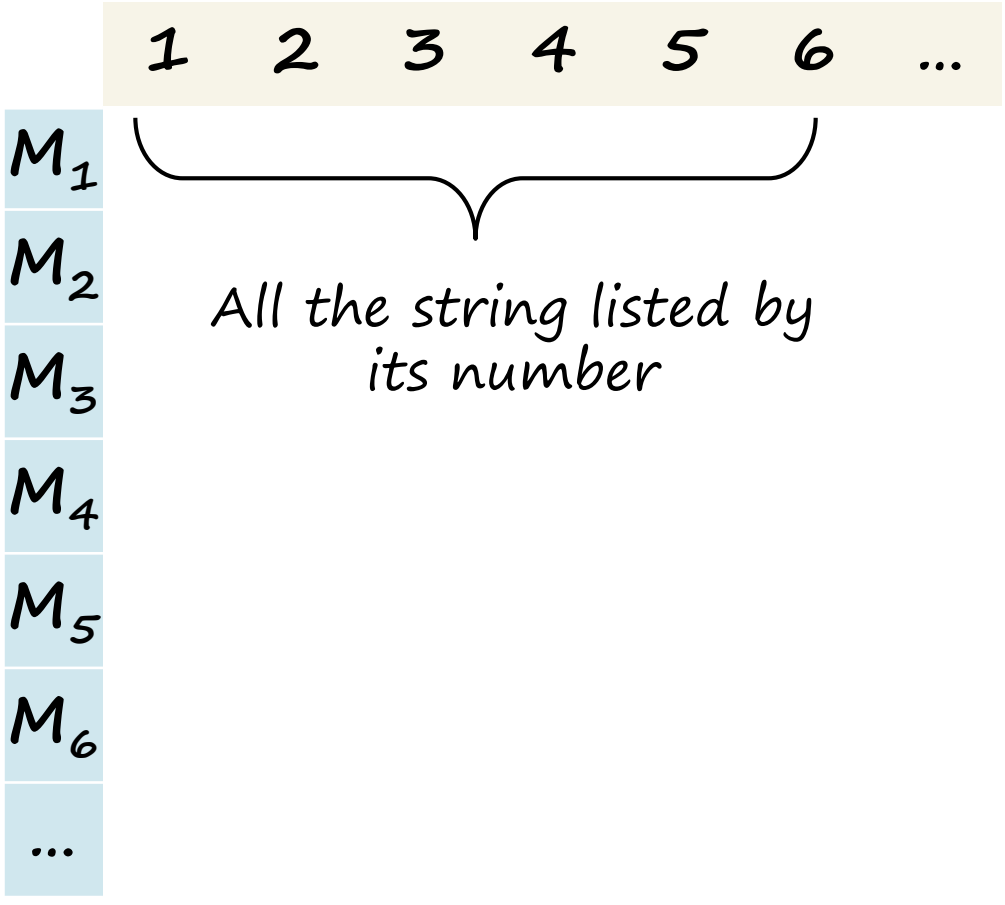
Numbering TMs

- We assign each TM a positive integer, and each positive integer has one TM corresponding to it.
  - We say that the $\boldsymbol{i}$th TM $\boldsymbol{M_i}$ is the TM whose number is $\boldsymbol{i}$.
- So we've build a bijection between the set of all the TMs and the set of positive integer. Then we can list all the TMs one by one.


M_1
M_2
M_3
M_4
M_5
M_6
...
All Turing Machines
listed by its number

Numbering All the String

- Each binary string represents a number
- We add a prefix 1 to each binary string to prevent the following condition
  - “01” and “001” are two different binary strings, but they represent the same number (1)
- w_i denotes the string with number i.
  - $w_1 = \epsilon, w_2 = 0, w_3 = 1, w_4 = 00$


1 2 3 4 5 6 ...
M1
M2
M3
M4
M5
M6
...
All the string listed by its number

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

1 in row i column j means that M_i accepts w_j.

0 means "not accept".

M_3 accepts w_4 (00)

M_6 does not accept w_6(10)

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

Row i represents the language of M_i

$L(M_4)=\{w_1,w_2,w_4,w_6\}$

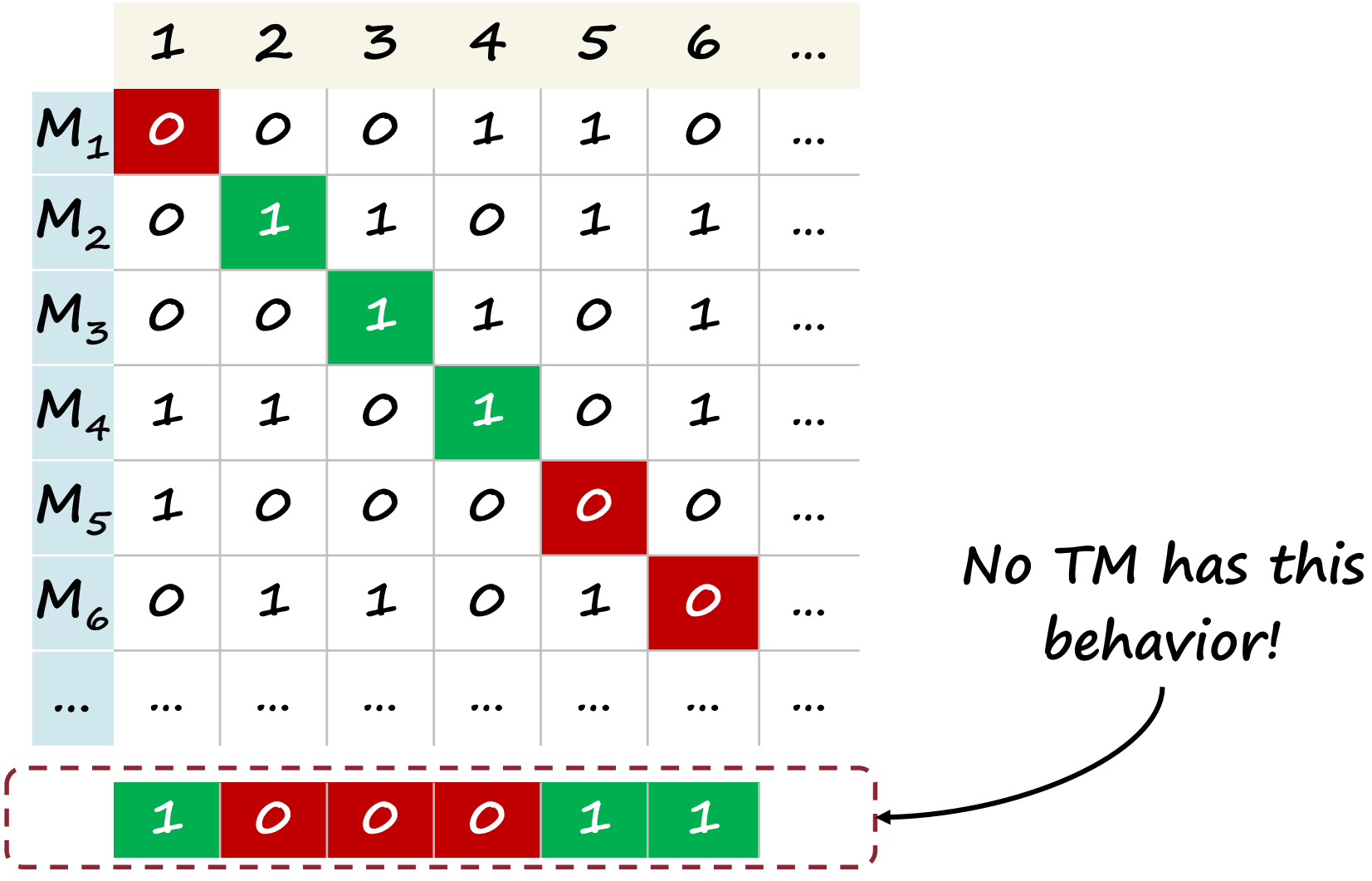
	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

0	1	1	1	0	0

	1	2	3	4	5	6	…
M_1	0	0	0	1	1	0	…
M_2	0	1	1	0	1	1	…
M_3	0	0	1	1	0	1	…
M_4	1	1	0	1	0	1	…
M_5	1	0	0	0	0	0	…
M_6	0	1	1	0	1	0	…
…	…	…	…	…	…	…	…

1	0	0	0	1	1	

Flip this language.
Swap what's included
and what's excluded


1 2 3 4 5 6 ...
M1 0 0 0 1 1 0 ...
M2 0 1 1 0 1 1 ...
M3 0 0 1 1 0 1 ...
M4 1 1 0 1 0 1 ...
M5 1 0 0 0 0 0 ...
M6 0 1 1 0 1 0 ...
... ...
1 0 0 0 1 1
No TM has this behavior!

Diagonalization Language

- The language L_d, the diagonalization language, is the set of string w_i such that w_i is not in $L(M_i)$

$$L_d = \{w_i | w_i \notin M_i\}$$

- Notice that the code for M_i is just w_i, so we can write L_d in a more interesting way:

$$L_d = \{M | M \notin L(M)\}$$

- The diagonalization language is all the codes of Turing machine which does not accept its own code.

$L_d \notin \mathbf{RE}$

- On a deeper philosophical level, the fact that non-**RE** languages exist supports the following claim:

  There are statements that are true but not provable.

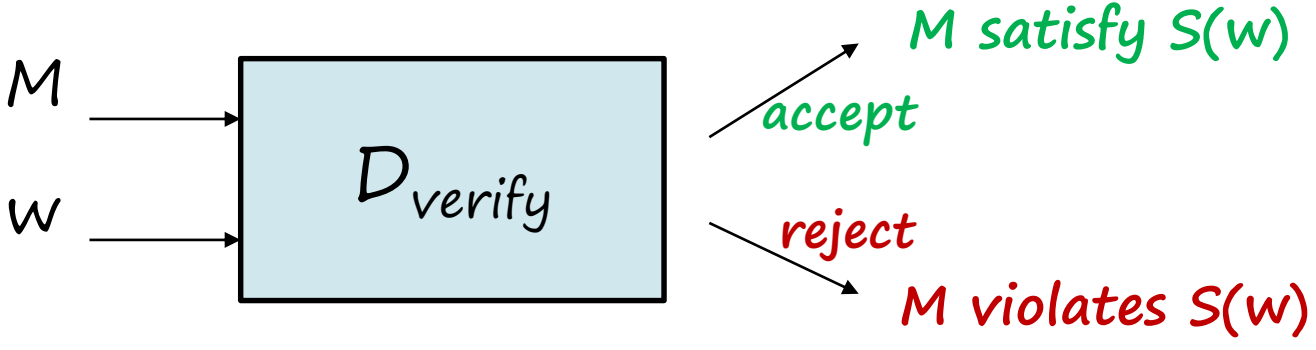
- Intuitively, given any non-**RE** language, there will be some strings in the language that cannot be proven to be in the language.
- This result can be formalized as a result called **Gödel's incompleteness theorem(哥德尔不完备定理)**, one of the most important mathematical results of all time.

Universal Program Verifying

Verify the correctness of every program.

Universal Program Verifying

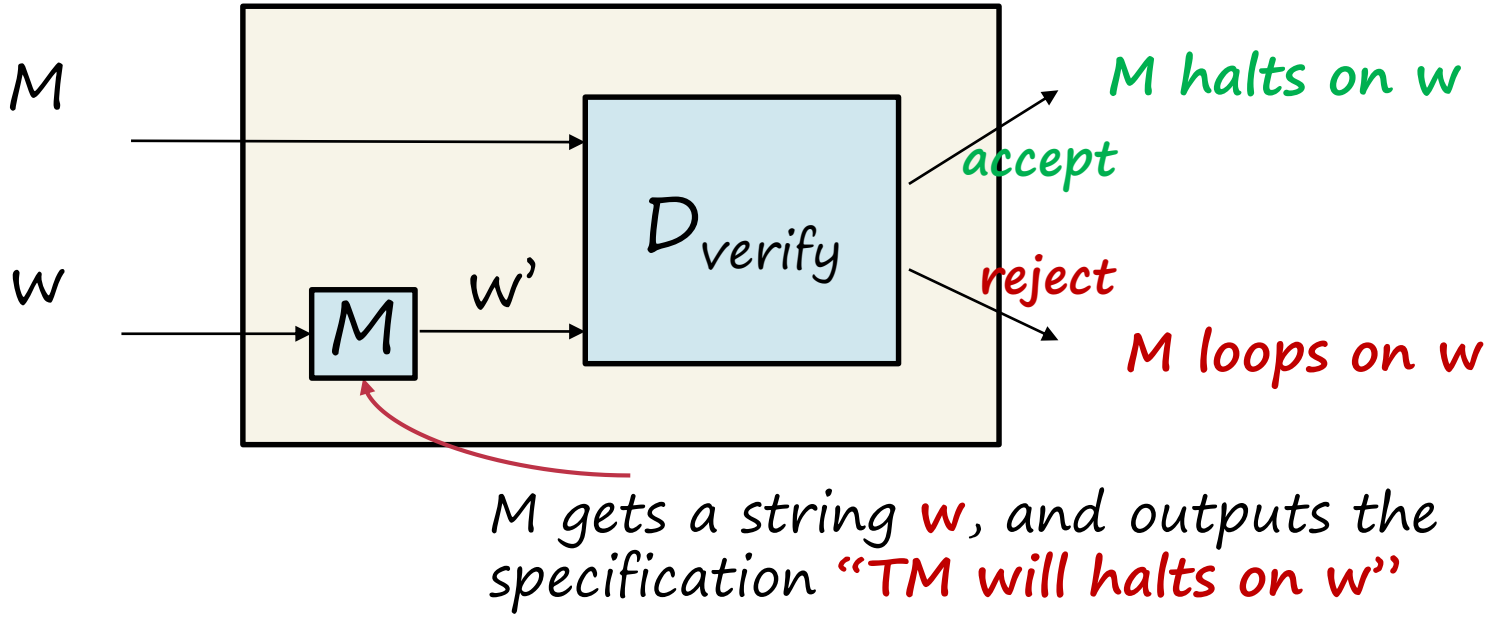
- Does there exist a TM to **decide** whether a TM satisfies a specification (some property of TM, encoded as binary string) given by an input?
- Or whether $L_{verify} = \{\langle M, w\rangle | M\ satisfies\ S(w)\}$, where $S(w)$ is the specification given by w, is in $\mathbf{R}$?



Universal Program Verifying

- We can reduce a halting Program to this problem.
- Or if D_{verify} exists, we can use it to solve the Halting Problem.

Universal Program Verifying

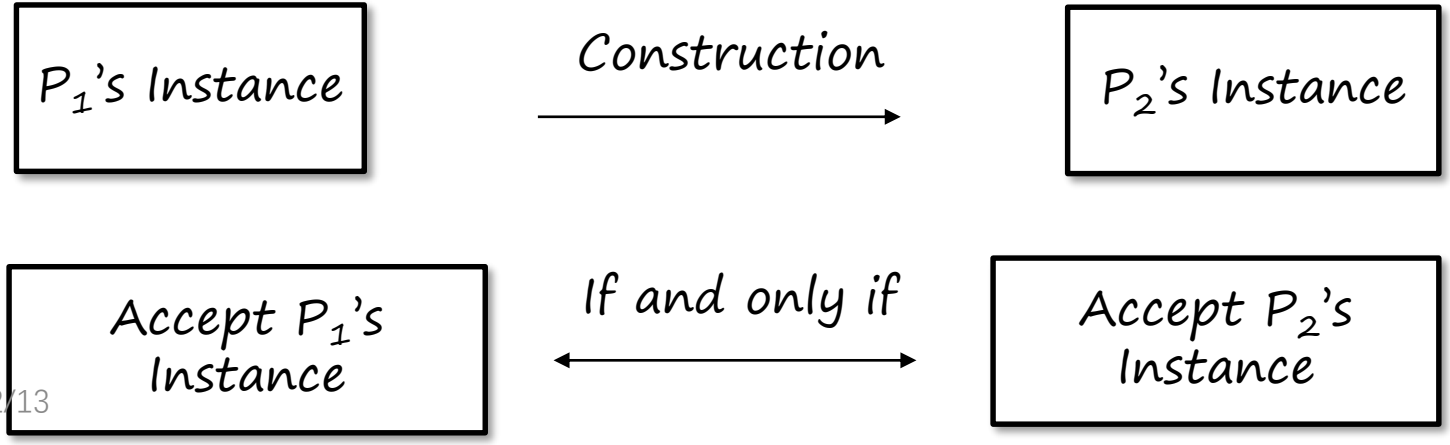


Universal Program Verifying

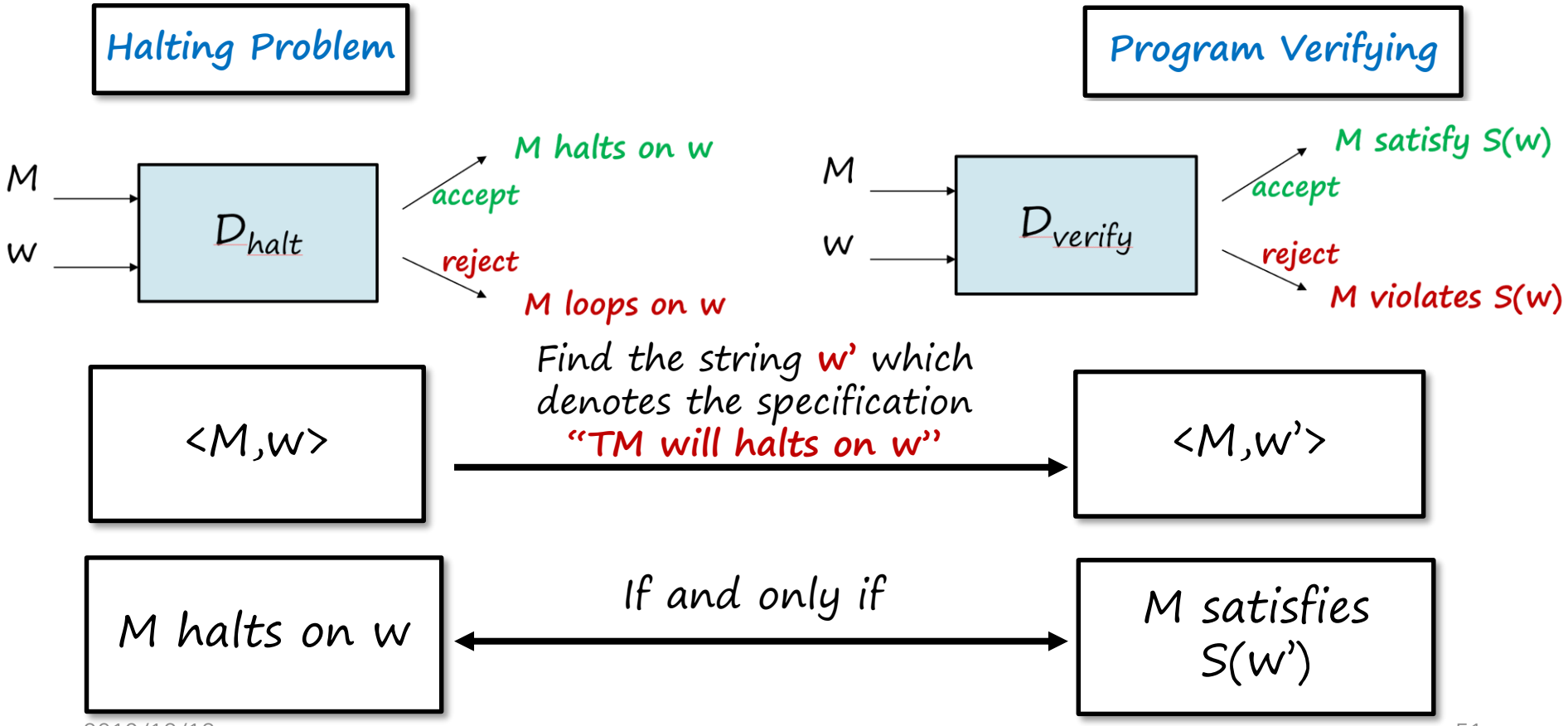
- We can reduce a halting program to this problem.
- Or if D_{verify} exists, we can use it to solve the halting problem.
- But we've proved halting problem is undecidable.
- So D_{verify} does not exists.
- **Universal Program Verifying is undecidable.**

Reduction

- We prove that universal program verifying problem is undecidable by reducing the halting problem to it.
- Reducing from problem P_1 to problem P_2 means:
  - An algorithm to convert P_1's instance to P_2's instance.
  - P_1's instance is accepted in P_1 if and only if its converted P_2's instance is accepted in P_2



Reductions


Halting Problem
Program Verifying
M
w
D_{halt}
M halts on w
accept
reject
M loops on w
M
w
D_{verify}
M satisfy S(w)
accept
reject
M violates S(w)
<M,w>
Find the string w' which denotes the specification "TM will halts on w"
<M,w'>
M halts on w
If and only if
M satisfies S(w')

Reduction

- If we can reduce a problem P_1 to another problem P_2, we intend to use the solution of P_2 to solve P_1, which means that P_2 is “harder” than P_1.
  - If P_1 is undecidable then so is P_2.
  - If P_1 is non-**RE**, then so is P_2.
- So if we want to prove some problem P is undecidable, we only need to reduce some already known undecidable problem to P
  - L_u, L_{halt}, etc.

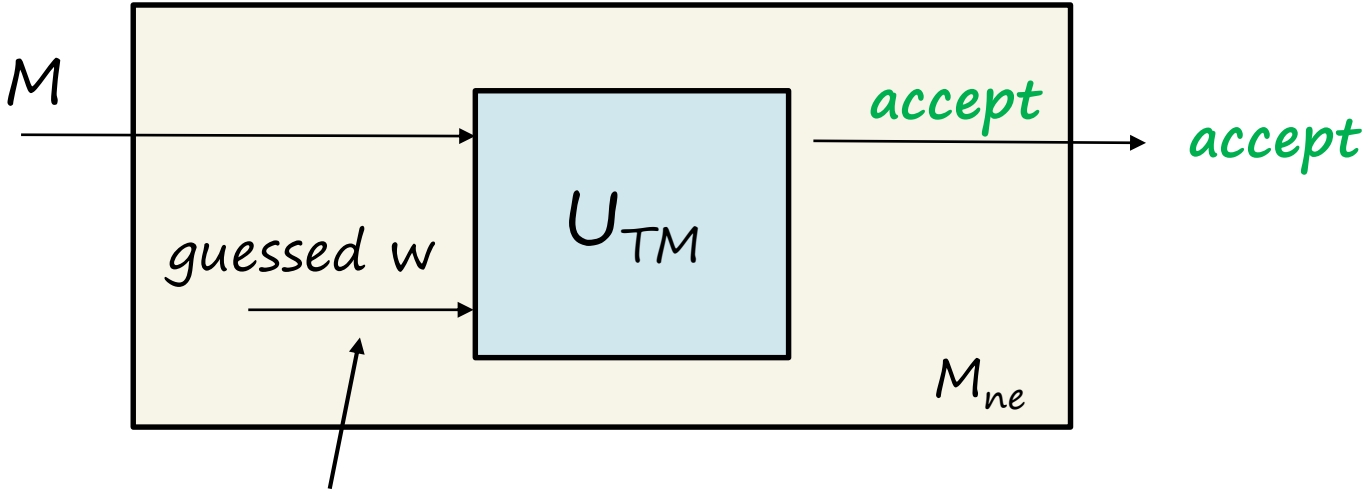
TM that Accepts the Empty Language

- As an example of reductions involving TMs, let us investigate two languages called L_e and L_{ne}.
  - If $L(M) = \emptyset$, that is, M does not accept any input, then the code of M is in L_e
  - Otherwise, if $L(M)$ is not the empty language, then the code of M is in L_{ne}

$$L_e = \{M | L(M) = \emptyset\}$$

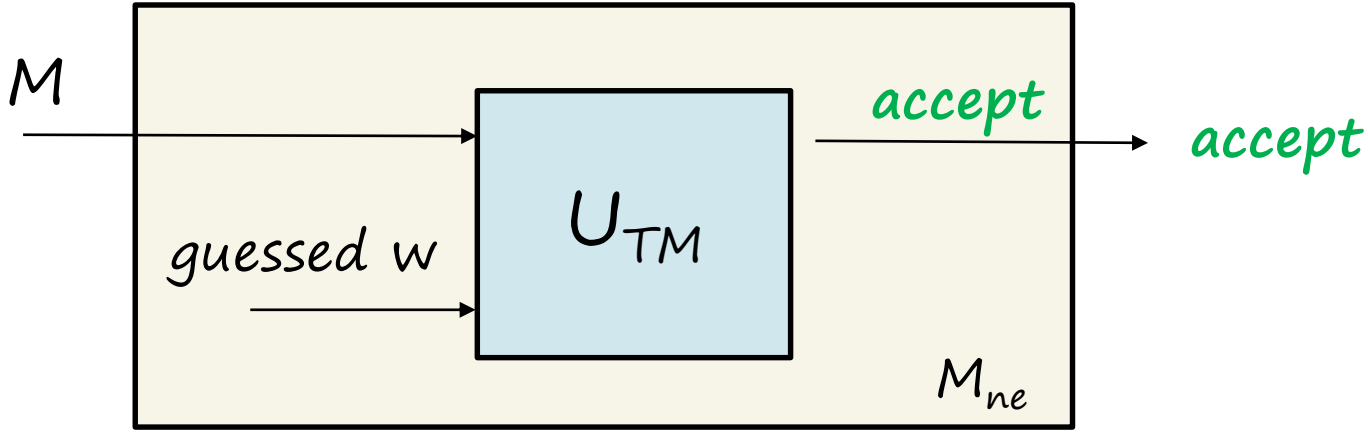
$$L_{ne} = \{M | L(M) \neq \emptyset\}$$

$L_{ne} \in$ RE



We guess all the w simultaneously instead of guessing w one by one in case getting into a forever loop (For more information, ref to 8.4.4 Nondeterministic TM)

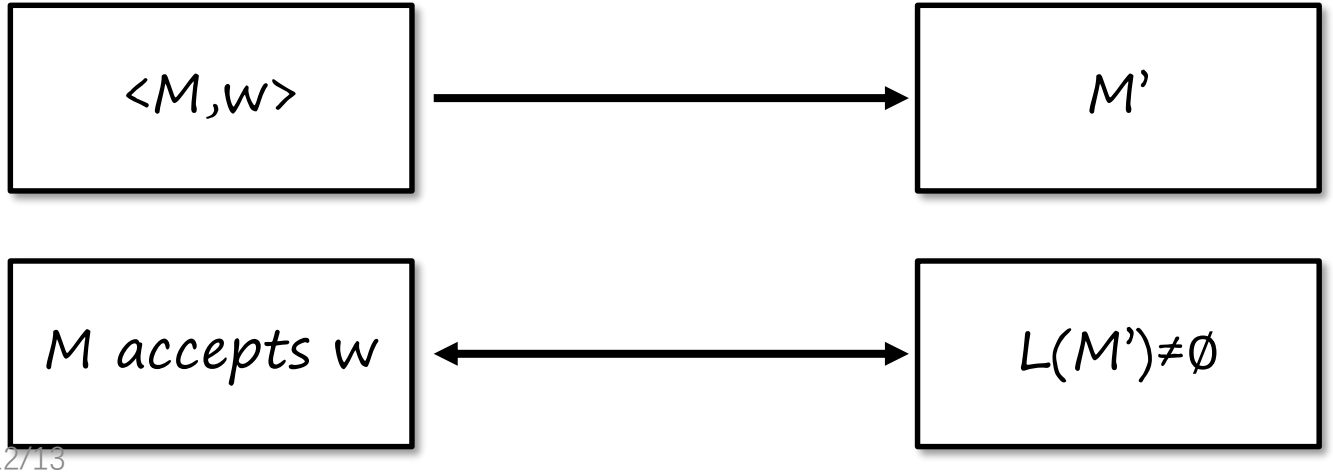
$L_{ne} \in \mathbf{RE}$



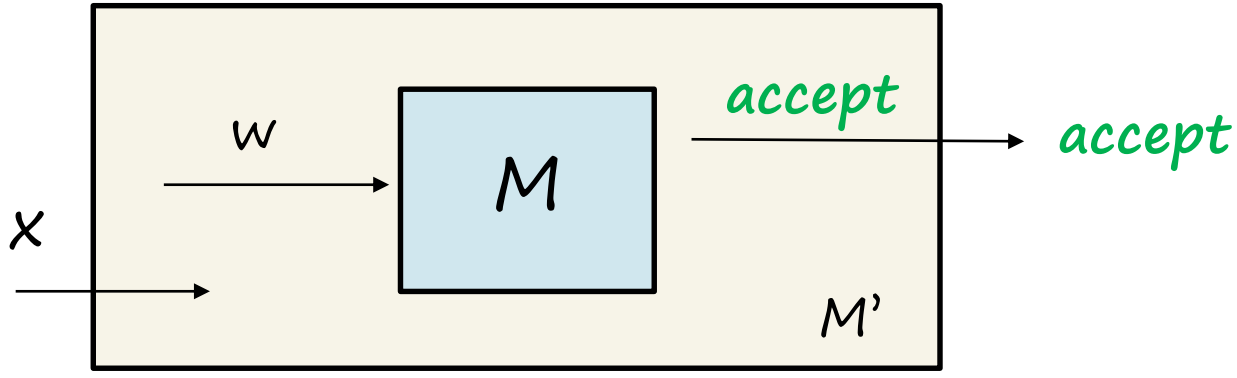
- If $M \in L_{ne}$, which means M accepts some string w, we'll finally guess w, then M_{ne} accepts M.
- If $M \notin L_{ne}$, which means M accepts no string, then we will guess a string that does not exist forever, then M_{ne} does not accept M.

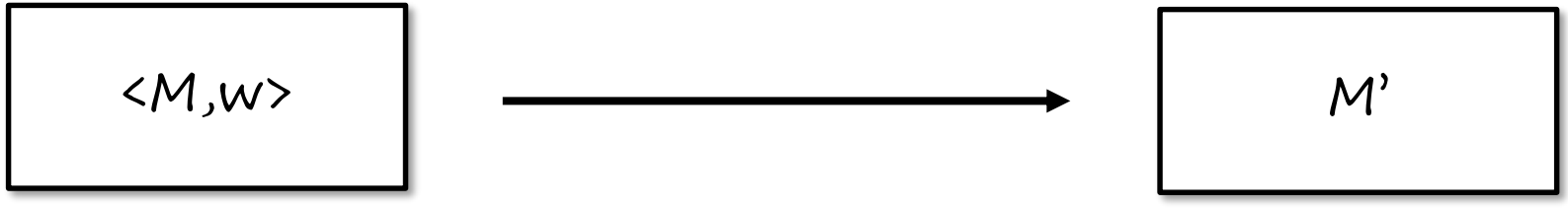
$L_{ne} \notin \mathbf{R}$

- We'll prove this by reducing L_u to L_{ne}.
- We need to design an algorithm that converts an input that is a pair $\langle \boldsymbol{M}, \boldsymbol{w} \rangle$ into a TM $\boldsymbol{M'}$ such that $\boldsymbol{L}(\boldsymbol{M'}) \neq \emptyset$ if and only if $\boldsymbol{M}$ accepts the input $\boldsymbol{w}$.

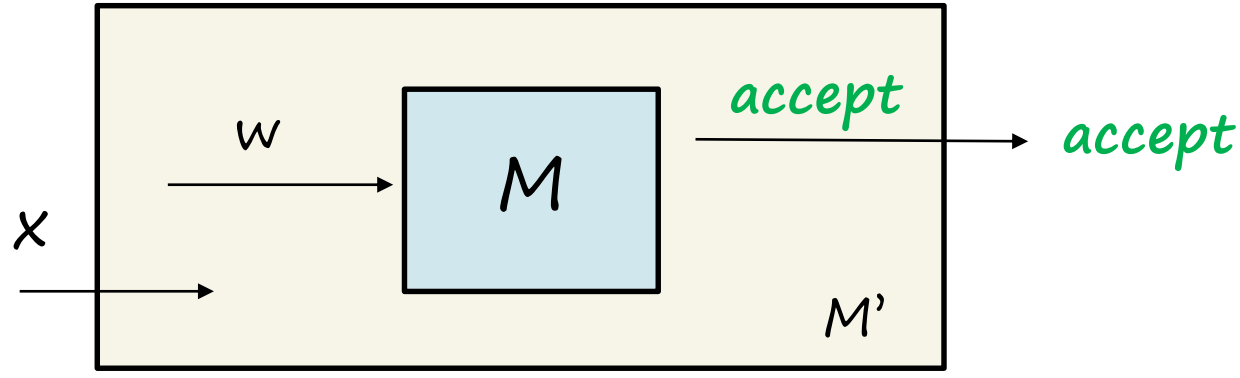


$L_u \rightarrow L_{ne}$



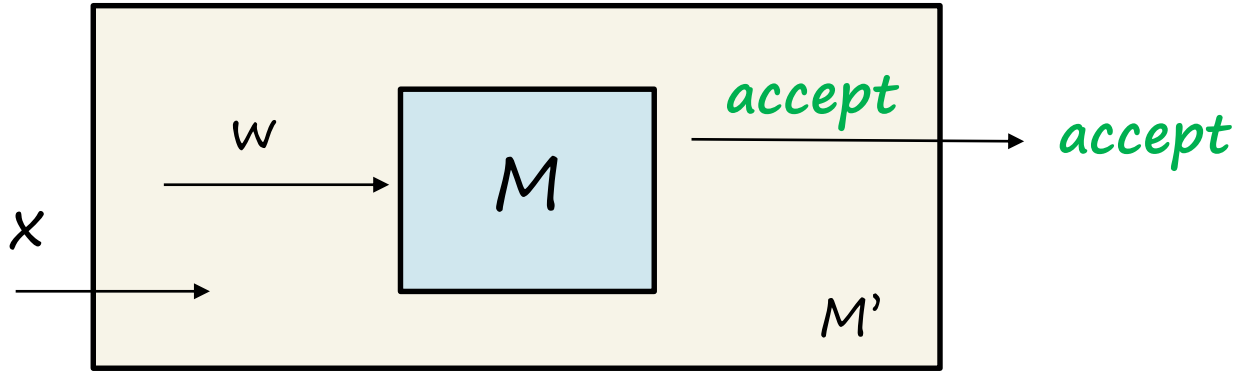


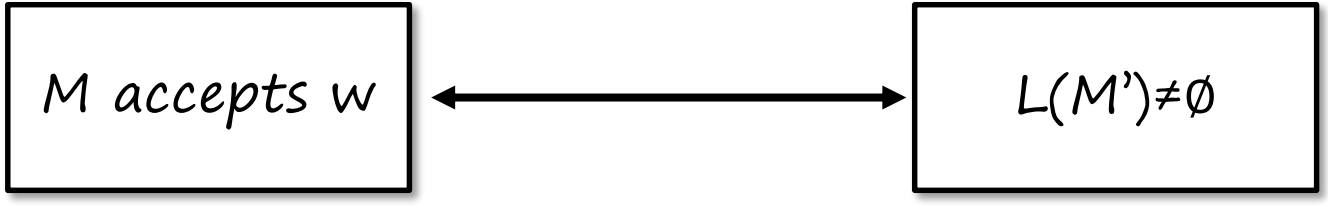
$L_u \rightarrow L_{ne}$



- If M accepts w
  - Then M' accepts any strings, which means $L(M') \neq \emptyset$
- If $L(M') \neq \emptyset$
  - Then M must accept w

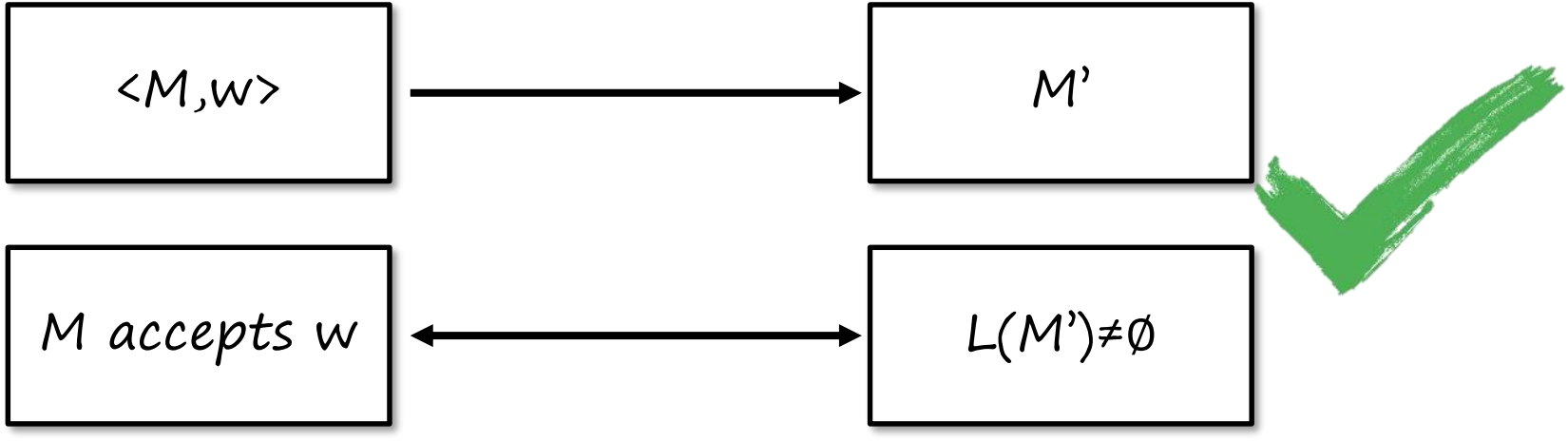
$L_u \rightarrow L_{ne}$





$L_{ne} \notin \mathbf{R}$

- We'll prove this by reduce L_u to L_{ne}.
- We need to design an algorithm that converts an input that is a pair $\langle \boldsymbol{M}, \boldsymbol{w} \rangle$ into a TM $\boldsymbol{M'}$ such that $\boldsymbol{L(M')} \neq \emptyset$ if and only if $\boldsymbol{M}$ accepts input $\boldsymbol{w}$.



TM that Accepts the Empty Language

- We now know that $L_{ne} \in \mathbf{RE}$ and $L_{ne} \notin \mathbf{R}$.
- As $L_e = \overline{L_{ne}}$, so if $L_e \in \mathbf{RE}$, then $L_{ne} \in \mathbf{R}$. (why?)
- But $L_{ne} \notin \mathbf{R}$. So we have $\boldsymbol{L_e \notin \mathbf{RE}}$

Properties of RE

- L_e and L_{ne} are the set of codes for certain TMs, they describe the TMs which have certain properties.
- More generally, the **property** of RE language is simply a set of RE languages.
  - The property of being regular is formally the set of all regular languages.
  - The property of being empty is the set $\{\emptyset\}$ consisting of only the empty language.

Properties of RE

- A property is **trivial** if it is either empty or is all RE languages. Otherwise, it is **nontrivial**.
  - Note that the empty property, $\emptyset$, is different from the property of being an empty language $\{\emptyset\}$
- Let $\mathcal{P}$ denote a property of RE languages, then the language $L_{\mathcal{P}}$ is the set of codes for TM M such that $L(M) \in \mathcal{P}$

$$L_{\mathcal{P}} = \{M | L(M) \in \mathcal{P}\}$$

Rice's Theorem

Every nontrivial property of RE languages is undecidable.
Or given a nontrivial property $\mathcal{P}$, $L_{\mathcal{P}} \notin \mathbf{R}$.

- It can be similarly proved by reducing L_u to $L_{\mathcal{P}}$.
  - Proof ignored, feel free to read contents in IALC 9.3.3

Rice's Theorem

- By applying Rice's Theorem, we can prove the undecidability of some problem by simply choosing the property.
  - Let $\mathcal{P} = \{\emptyset\}$, then $L_{\mathcal{P}} = \{M | L(M) \in \{\emptyset\}\} = \{M | L(M) = \emptyset\} = L_e$ is undecidable.

Where We Are

- Now can we answer the question:

  What problems can a computer solve?

- Unfortunately we can only tell there exist some unsolvable(undecidable) programs. But we cannot know exactly what programs can be solved.
  - Actually there's little understanding about the sufficient condition for a problem to be computable.

Where We Are

- Knowing some problem is undecidable means:
  - Hold back our ambition of finishing an impossible task. (The universal program verifier does not exist.)
  - Then try to find another decidable problem to solve. (But we can build a verifier for specific programs.)
  - And it is interesting itself.

How we investigate computability?

2.1 Discussion on Problems

Part1. Intro & Set theory(I) : Basics & Formal Language.

Part2. Set Theory(II): Axiom system & Cardinality.

Part3. Capture Structures: Binary Relation & Function

2.2 Discussion on Computation

Part1. Turing Machine Basics.

Part2. Variants of Turing Machine. Church-Turing Thesis.

2.3 Discussion on computability

Part1. The Language of Turing Machine. R & RE

Part2. Undecidability.