Undecidability and Reductions

Lecture 21

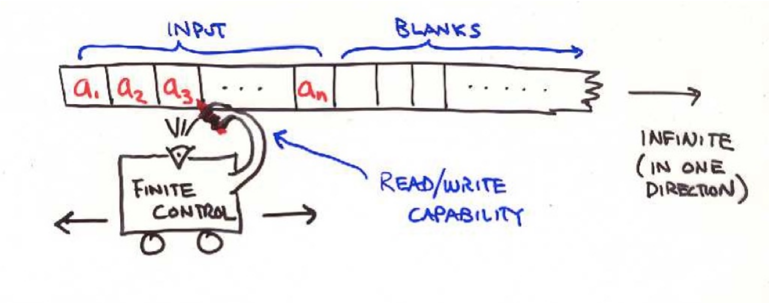
April 13, 2023

Part I

TM Recap and Recursive/Decidable Languages

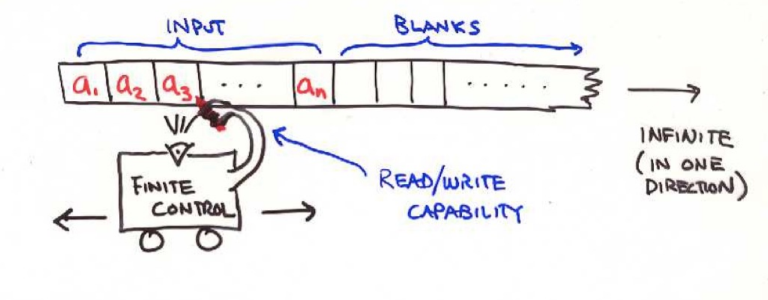
Turing Machine

- DFA with infinite tap
- One move: read, write, move one cell, change state



Turing Machine

- DFA with infinite tap
- One move: read, write, move one cell, change state



On a given input string w a TM M does one of the following:

- halt and accept w
- halt and reject w
- go into an infinite loop (not halt)
- crash in which case we think of it as rejecting w

Recursive and Recursively Enumerable

Definition

Given TM M, $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
We say M accepts L.

Caveat: A language L can be accepted by many different TMs.

Recursive and Recursively Enumerable

Definition

Given TM M, $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
We say M accepts L.

Caveat: A language L can be accepted by many different TMs.

Definition

M is an **algorithm** if it halts on every input and accepts/rejects.

Recursive and Recursively Enumerable

Definition

Given TM M, $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
We say M accepts L.

Caveat: A language L can be accepted by many different TMs.

Definition

M is an **algorithm** if it halts on every input and accepts/rejects.

Definition

A language L is **decidable (or recursive)** if there is an algorithm M such that $L = L(M)$.

Recursive and Recursively Enumerable

Definition

Given TM M, $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
We say M accepts L.

Caveat: A language L can be accepted by many different TMs.

Definition

M is an **algorithm** if it halts on every input and accepts/rejects.

Definition

A language L is **decidable (or recursive)** if there is an algorithm M such that $L = L(M)$.

Definition

A language L is **recursively enumerable** if there is a TM M such that $L = L(M)$.

Recursive and Recursively Enumerable

- If L is recursive then $\bar{L} = \Sigma^* - L$ is also recursive
- If L is recursive then L is r.e.

Recursive and Recursively Enumerable

- If L is recursive then $\bar{L} = \Sigma^* - L$ is also recursive
- If L is recursive then L is r.e.
- Suppose L is r.e. $L = L(M)$ for some M.
  - If $w \in L$ then M halts and accepts w.

Recursive and Recursively Enumerable

- If L is recursive then $\bar{L} = \Sigma^* - L$ is also recursive
- If L is recursive then L is r.e.
- Suppose L is r.e. $L = L(M)$ for some M.
  - If $w \in L$ then M halts and accepts w.
  - If $w \not\in L$ then

Recursive and Recursively Enumerable

- If L is recursive then $\bar{L} = \Sigma^* - L$ is also recursive
- If L is recursive then L is r.e.
- Suppose L is r.e. $L = L(M)$ for some M.
  - If $w \in L$ then M halts and accepts w.
  - If $w \not\in L$ then M may or may not halt! If M halts then it rejects w.

Recursive and Recursively Enumerable

- If L is recursive then $\bar{L} = \Sigma^* - L$ is also recursive
- If L is recursive then L is r.e.
- Suppose L is r.e. $L = L(M)$ for some M.
  - If $w \in L$ then M halts and accepts w.
  - If $w \notin L$ then M may or may not halt! If M halts then it rejects w.

Question: Are r.e languages interesting? And why?

- Technical/mathematical reasons
- Pragmatic reasons. We are used to programs that are correct, but are willing to give up on efficiency/halting.

Recursive and Recursively Enumerable

- If L is recursive then $\bar{L} = \Sigma^* - L$ is also recursive
- If L is recursive then L is r.e.
- Suppose L is r.e. $L = L(M)$ for some M.
  - If $w \in L$ then M halts and accepts w.
  - If $w \notin L$ then M may or may not halt! If M halts then it rejects w.

Question: Are r.e languages interesting? And why?

- Technical/mathematical reasons
- Pragmatic reasons. We are used to programs that are correct, but are willing to give up on efficiency/halting.

Definition

L is **undecidable** if there is no algorithm M such that $L = L(M)$. L is **not r.e** if there is no TM M such that $L = L(M)$.

Universal TM

A single TM that can simulate other TMs. Basis of modern computers. Single computer that runs many different programs.

- UTM takes as input $\langle M \rangle$ (encoding of a TM M) and a string w. Typically written as $\langle M, w \rangle$.

Universal TM

A single TM that can simulate other TMs. Basis of modern computers. Single computer that runs many different programs.

- UTM takes as input $\langle M \rangle$ (encoding of a TM M) and a string w. Typically written as $\langle M, w \rangle$.
- UTM **simulates** M on w.
    - If M accepts w then UTM accepts its input $\langle M, w \rangle$.
    - If M halts and rejects w then UTM rejects its input $\langle M, w \rangle$.
    - If M does not halt on w then UTM also does not halt on input $\langle M, w \rangle$ and hence does not accept its input.

Universal TM

A single TM that can simulate other TMs. Basis of modern computers. Single computer that runs many different programs.

- UTM takes as input $\langle M \rangle$ (encoding of a TM M) and a string w. Typically written as $\langle M, w \rangle$.
- UTM **simulates** M on w.
  - If M accepts w then UTM accepts its input $\langle M, w \rangle$.
  - If M halts and rejects w then UTM rejects its input $\langle M, w \rangle$.
  - If M does not halt on w then UTM also does not halt on input $\langle M, w \rangle$ and hence does not accept its input.
- What is the language of UTM? Special name called Universal Language denote by L_u.

$$L_u = \{\langle M, w \rangle \mid M \text{ accepts } w.\}.$$

Encoding TMs

Observation

There is a fixed encoding such that every TM M can be represented as a unique binary string.

Equivalently we think of a TM as simply a program which is a string.

For each string that is not a valid encoding we associate a *dummy* TM that does not accept any string. Why?

Encoding TMs

Observation

There is a fixed encoding such that every TM M can be represented as a unique binary string.

Equivalently we think of a TM as simply a program which is a string.

For each string that is not a valid encoding we associate a *dummy* TM that does not accept any string. Why?

One-to-one correspondence between binary strings and TMs.

M_i is the the TM associate with integer i

How many TMs?

One-to-one correspondence between integers and TMs.

Proposition

The number of TMs is countably infinite.

How many TMs?

One-to-one correspondence between integers and TMs.

Proposition

The number of TMs is countably infinite.

Easy but important corollaries:

- Hence, countably infinite number of r.e (hence also recursive) languages
- Number of languages is uncountably infinite! Hence there must be languages that are not r.e/recursive and hence undecidable! In fact, most langauges are undecidable!

How many TMs?

One-to-one correspondence between integers and TMs.

Proposition

The number of TMs is countably infinite.

Easy but important corollaries:

- Hence, countably infinite number of r.e (hence also recursive) languages
- Number of languages is uncountably infinite! Hence there must be languages that are not r.e/recursive and hence undecidable! In fact, most langauges are undecidable!

Question: Which *interesting* languages are undecidable/not r.e?

Part II

Undecidable Languages and Proofs via Reductions

Undecidable Languages

Counting argument shows that too many languages and too few TMs/programs hence most languages are not decidable.

What "real-world" and "natural" languages are undecidable?

Short answer: reasoning about general programs is difficult.

Undecidable Languages

Counting argument shows that too many languages and too few TMs/programs hence most languages are not decidable.

What "real-world" and "natural" languages are undecidable?

Short answer: reasoning about general programs is difficult.

Theorem (Turing)

Following languages are undecidable.

- $L_{HALT} = \{\langle M \rangle \mid M \text{ halts on blank input}\}$
- $L_{HALT,w} = \{\langle M, w \rangle \mid M \text{ halts on input } w\}$
- $L_u = \{\langle M, w \rangle \mid M \text{ accepts } w\}$

Recall that languages are problems. Jeff's notes calls Halting problem $HALT$ (the second version)

What else is undecidable?

Via (sometimes highly non-trivial) reductions one can show

- Essentially many questions about sufficiently general programs are undecidable
- Many problems in mathematical logic are undecidable
- Posts correspondence problem which is a string problem
- Tiling problems
- Problems in mathematics such as Diophantine equation solution (Hilbert's 10th problem)

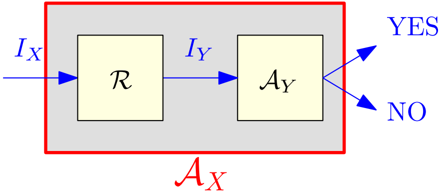
Undecidablity connects computation to mathematics/logic and proofs

What do we want you to know?

- The core undecidable problems (***HALT*** and $\boldsymbol{L_u}$)
- Ability to do simple reductions that prove undecidability of program behaviour

Reductions

1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y:
3. $\implies$ New algorithm for X:



We write $X \leq Y$ if X reduces to Y

Lemma

If $X \leq Y$ **and** *X is undecidable then Y is undecidable.*

CS 125 assignment

Write a program that prints "Hello World"

```
main() {
    print(''Hello World'')
}
```

CS 125 assignment

Write a program that prints "Hello World"

```
main() {
    print(''Hello World'')
}
```

Question: Can we create an autograder?

CS 125 assignment

Write a program that prints "Hello World"

```
main() {
    print(''Hello World'')
}
```

Question: Can we create an autograder? No! Why?

```
main() {
    stealthcode()
    print(''Hello World'')
}
stealthcode() {
    do this
    do that
    viola
}
```

Reducing Halting to Autograder

- **Halting problem:** given *arbitrary* program foo(), does it halt?

Reducing Halting to Autograder

- **Halting problem:** given *arbitrary* program foo(), does it halt?
- **Reduction to CS125Autograder**: given foo() output foobar()

```
main() {
    foo()
    print(‘‘Hello World’’)
}
foo() {
    line 1
    line 2
    ...
}
```

Note: Reduction only needs to add a few lines of code to foo()

Reducing Halting to Autograder

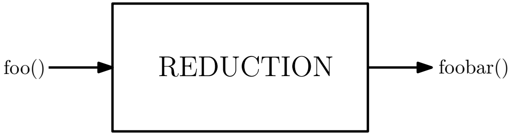
- **Halting problem:** given *arbitrary* program foo(), does it halt?
- **Reduction to CS125Autograder**: given foo() output foobar()

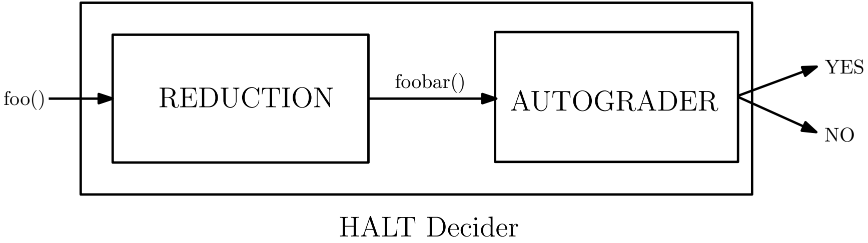
```
main() {
    foo()
    print(``Hello World'')
}
foo() {
    line 1
    line 2
    ...
}
```

Note: Reduction only needs to add a few lines of code to foo()

- foobar() prints "Hello World" **if and only if** foo() halts!
- If we had CS125Autograder then we can solve Halting. But Halting is hard according to Turing. Hence ...

Reducing Halting to Autograder





Connection to proofs

Goldbach's conjecture: Every *even* integer ≥ 4 can be written as sum of two primes. Made in 1742, still open.

Connection to proofs

Goldbach's conjecture: Every *even* integer ≥ 4 can be written as sum of two primes. Made in 1742, still open.

If Halting can be solved then can solve Goldbach's conjecture. How? Can write a program that halts if and only if conjecture is *false*.

```
golbach() {
    n = 4
    repeat
        flag = FALSE
        for (int i = 2, i < n; i++) do
            If (i and (n - i) are both prime)
                flag = TRUE; Break
        If (!flag) return ``Goldbach's Conjecture is False''
        n = n + 2
    Until (TRUE)
}
```

More reduction about languages

We will show following languages about program behaviour are undecidable.

- $L_{374} = \{\langle M \rangle \mid L(M) = \{0^{374}\}\}$
- $L_{\neq\emptyset} = \{\langle M \rangle \mid L(M) \neq \emptyset\}$
- a template to show that essentially checking whether a given program's language satisfies some non-trivial property is undecidable

Same proof technique as the one for autograder

Undecidability of L_{374}

Understanding: What is the problem of deciding L_{374}?

Given an arbitrary program ***boo(str w)*** *does* ***boo()*** *accept only the string* 0^{374} *and nothing else?*

Undecidability of L_{374}

Understanding: What is the problem of deciding L_{374}?

Given an arbitrary program **boo(str w)** *does* **boo()** *accept only the string* 0^{374} *and nothing else?*

Seems harder than autograder for printing "Hello World"!

Undecidability of L_{374}

Understanding: What is the problem of deciding L_{374}?

Given an arbitrary program $\textbf{boo}(\textbf{str } w)$ *does* $\textbf{boo}()$ *accept only the string* 0^{374} *and nothing else?*

Seems harder than autograder for printing "Hello World"!

Prove that if we had a decider DecideL_{374} for L_{374} then we can create a decider for HALT.

Undecidability of L_{374}

Understanding: What is the problem of deciding $\boldsymbol{L}_{374}$?

Given an arbitrary program $\boldsymbol{boo}(\boldsymbol{str}\ \boldsymbol{w})$ *does* $\boldsymbol{boo}()$ *accept only the string* 0^{374} *and nothing else?*

Seems harder than autograder for printing "Hello World"!

Prove that if we had a decider Decide$\boldsymbol{L}_{374}$ for $\boldsymbol{L}_{374}$ then we can create a decider for HALT.

Recall: Decider for HALT takes an arbitrary program $\boldsymbol{foo}()$ and needs to check if $\boldsymbol{foo}()$ halts.

Undecidability of L_{374}

Understanding: What is the problem of deciding L_{374}?

*Given an arbitrary program **boo**(**str w**) does **boo**() accept only the string* 0^{374} *and nothing else?*

Seems harder than autograder for printing "Hello World"!

Prove that if we had a decider DecideL_{374} for L_{374} then we can create a decider for HALT.

Recall: Decider for HALT takes an arbitrary program ***foo***() and needs to check if ***foo***() halts.
Reduction should transform ***foo***() into a program ***fooboo***() such that answer to ***fooboo***() from DecideL_{374} will let us know if ***foo***() halts.

Undecidability of L_{374}

A simple program ***simpleboo*(*str w*)**

```
simpleboo(str w) {
    if (w = 0^374) then return YES
    return NO
}
```

Easy to see that $\boldsymbol{L(simpleboo())} = \{0^{374}\}$.

Undecidability of L_{374}

A simple program $\textit{simpleboo}(\textit{str } w)$

```
simpleboo(str w) {
    if (w = 0^374) then return YES
    return NO
}
```

Easy to see that $L(\textit{simpleboo}()) = \{0^{374}\}$.

Given arbitrary program $\textit{foo}()$ reduction creates $\textit{fooboo}(\textit{str } w)$:

```
fooboo(str w) {
    foo()
    if (w = 0^374) then Return YES
    return NO
}
foo () {
code of foo ...
}
```

Undecidability of L_{374}

Lemma

Language of ***fooboo***() *is* $\{0^{374}\}$ *if* ***foo***() *halts. Language of* ***fooboo***() *is* $\emptyset$ *if* ***foo***() *does not halt.*

Undecidability of L_{374}

Lemma

Language of $\textbf{fooboo}()$ *is* $\{0^{374}\}$ *if* $\textbf{foo}()$ *halts. Language of* $\textbf{fooboo}()$ *is* $\emptyset$ *if* $\textbf{foo}()$ *does not halt.*

Corollary

$\textbf{fooboo}()$ *in* $\boldsymbol{L}_{374}$ *if and only if* $\textbf{foo}() \in \boldsymbol{L_{HALT}}$.

Corollary

If $\boldsymbol{L}_{374}$ *is decidable then* $\boldsymbol{L_{HALT}}$ *is decidable. Since* $\boldsymbol{L_{HALT}}$ *is undecidable* $\boldsymbol{L}_{374}$ *is undecidable.*

Undecidability of $L_{\neq\emptyset}$

Understanding: What is the problem of deciding $L_{\neq\emptyset}$?

Given an arbitrary program $\textbf{boo}(\textbf{str } \textbf{w})$ *does* $\textbf{boo}()$ *accept any string?*

Undecidability of $L_{\neq\emptyset}$

Understanding: What is the problem of deciding $L_{\neq\emptyset}$?

Given an arbitrary program ***boo***(***str w***) *does* ***boo***() *accept any string?*

Reduce from HALT: given arbitrary program ***foo***() create ***fooboo***() such that ***fooboo***() accepts some string iff ***foo***() halts.

Undecidability of $L_{\neq\emptyset}$

A simple program $\boldsymbol{simpleboo(str\ w)}$

```
simpeboo(str w) {
return YES
}
```

Easy to see that $L(\boldsymbol{simpleboo()}) = \Sigma^*$ and hence not empty.

Undecidability of $L_{\neq\emptyset}$

A simple program ***simpleboo***(***str w***)

```
simpeboo(str w) {
return YES
}
```

Easy to see that $L(\textit{simpleboo}()) = \Sigma^*$ and hence not empty.

Given arbitrary program ***foo***(), reduction creates ***fooboo***(***str w***):

```
fooboo(str w) {
    foo()
    return YES
}
foo () {
code of foo ...
}
```

Undecidability of $L_{\neq\emptyset}$

Lemma

Language of **fooboo**() *is* Σ^* *if* **foo**() *halts. Language of* **fooboo**() *is* $\emptyset$ *if* **foo**() *does not halt.*

Undecidability of $L_{\neq \emptyset}$

Lemma

Language of **fooboo**() *is* Σ^* *if* **foo**() *halts. Language of* **fooboo**() *is* $\emptyset$ *if* **foo**() *does not halt.*

Corollary

fooboo() *in* $L_{\neq \emptyset}$ *if and only if* **foo**() $\in L_{HALT}$.

Beyond r.e

Lemma

If L is recursive then $\bar{L} = \Sigma^* - L$ is recursive.

Beyond r.e

Lemma

If L is recursive then $\bar{L} = \Sigma^* - L$ is recursive.

Lemma

Suppose L and $\bar{L}$ are both r.e. Then L is recursive.

Beyond r.e

Lemma

If L is recursive then $\bar{L} = \Sigma^* - L$ is recursive.

Lemma

Suppose L and $\bar{L}$ are both r.e. Then L is recursive.

Proof.

We have TMs M, M' such that $L = L(M)$ and $\bar{L} = L(M')$. Construct new TM M^* that on input w simulates *both* M and M' on w in *parallel*. One of them has to halt and give right answer. □

Beyond r.e

Lemma

If L is recursive then $\bar{L} = \Sigma^* - L$ is recursive.

Lemma

Suppose L and $\bar{L}$ are both r.e. Then L is recursive.

Proof.

We have TMs M, M' such that $L = L(M)$ and $\bar{L} = L(M')$. Construct new TM M^* that on input w simulates *both* M and M' on w in *parallel*. One of them has to halt and give right answer. □

Corollary

Suppose L is r.e but not recursive. Then $\bar{L}$ is not r.e.

Beyond r.e

Corollary

Suppose L is r.e but not recursive. Then $\bar{L}$ is not r.e.

Thus $\overline{L_{HALT}}$ and $\overline{L_u}$ are not even r.e. What does this mean?

Beyond r.e

Corollary

Suppose L is r.e but not recursive. Then $\bar{L}$ is not r.e.

Thus $\overline{L_{HALT}}$ and $\overline{L_u}$ are not even r.e. What does this mean?

What problem is $\overline{L_{HALT}}$? Given code/program $\langle M \rangle$ does it *not* halt on blank input? How can we tell?

We can simulate M using a UTM. How long? If M halts during simulation, UTM can reject $\langle M \rangle$. But if it does not halt after a billion steps can we stop simulation and say for sure that M will not halt? Perhaps there are other ways of figuring this out? Proof says no.

Part III

Undecidablity of Halting

Turing's Theorem

Theorem (Turing)

Following languages are undecidable.

- $L_{HALT} = \{\langle M \rangle \mid M \text{ halts on blank input}\}$
- $L_{HALT,w} = \{\langle M, w \rangle \mid M \text{ halts on input } w\}$
- $L_u = \{\langle M, w \rangle \mid M \text{ accepts } w\}$

Exercise: Prove that the above languages can be reduced to each other.

Turing's Theorem

Theorem (Turing)

Following languages are undecidable.

- $L_{HALT} = \{\langle M \rangle \mid M \textit{ halts on blank input}\}$
- $L_{HALT,w} = \{\langle M, w \rangle \mid M \textit{ halts on input } w\}$
- $L_u = \{\langle M, w \rangle \mid M \textit{ accepts } w\}$

Exercise: Prove that the above languages can be reduced to each other.

Two proofs

- A two step one based on Cantor's diagonalization
- A slick one but essentially the same idea in a different fashion

Diagonalization based proof

TMs can be put in 1-1 correspondence with integers: M_i is i'th TM

Definition

$L_d = \{\langle i \rangle \mid M_i \text{ does } \textbf{not} \text{ accept } \langle i \rangle\}$. Same as
$L_d = \{\langle M_i \rangle \mid M_i \text{ does } \textbf{not} \text{ accept } \langle i \rangle\}$.

Understanding L_d

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	...
M_0	no	no	no	no	no	no	no	no	no	no	...
M_1	yes	no	no	yes	no	yes	yes	yes	yes	no	...
M_2	no	yes	yes	no	no	yes	no	yes	no	no	...
M_3	no	yes	no	yes	no	yes	no	yes	no	yes	...
M_4	yes	yes	yes	yes	no	no	no	no	no	no	...
M_5	no	no	no	no	no	no	no	no	no	no	...
M_6	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	...
M_7	yes	yes	no	no	yes	yes	yes	no	no	yes	...
M_8	no	yes	no	no	yes	no	yes	yes	yes	no	...
M_9	no	no	no	yes	yes	no	yes	no	yes	yes	...
...	...	...	...	...	...	...	...	...	...	...	...

Understanding L_d

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	...
M_0	no	no	no	no	no	no	no	no	no	no	...
M_1	yes	no	no	yes	no	yes	yes	yes	yes	no	...
M_2	no	yes	yes	no	no	yes	no	yes	no	no	...
M_3	no	yes	no	yes	no	yes	no	yes	no	yes	...
M_4	yes	yes	yes	yes	no	no	no	no	no	no	...
M_5	no	no	no	no	no	no	no	no	no	no	...
M_6	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	...
M_7	yes	yes	no	no	yes	yes	yes	no	no	yes	...
M_8	no	yes	no	no	yes	no	yes	yes	yes	no	...
M_9	no	no	no	yes	yes	no	yes	no	yes	yes	...
...	...	...	...	...	...	...	...	...	...	...	...

Understanding L_d

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	...
M_0	yes	no	no	no	no	no	no	no	no	no	...
M_1	yes	yes	no	yes	no	yes	yes	yes	yes	no	...
M_2	no	yes	no	no	no	yes	no	yes	no	no	...
M_3	no	yes	no	no	no	yes	no	yes	no	yes	...
M_4	yes	yes	yes	yes	yes	no	no	no	no	no	...
M_5	no	no	no	no	no	yes	no	no	no	no	...
M_6	yes	yes	yes	yes	yes	yes	no	yes	yes	yes	...
M_7	yes	yes	no	no	yes	yes	yes	yes	no	yes	...
M_8	no	yes	no	no	yes	no	yes	yes	no	no	...
M_9	no	no	no	yes	yes	no	yes	no	yes	no	...
...	...	...	...	...	...	...	...	...	...	...	...

L_d is not r.e

$L_d = \{\langle i \rangle \mid M_i$ does **not** accept $\langle i \rangle\}$.

Theorem

L_d is not r.e.

L_d is not r.e

$L_d = \{\langle i \rangle \mid M_i \text{ does } \textbf{not} \text{ accept } \langle i \rangle\}$.

Theorem

L_d is not r.e.

Proof by contradiction. Suppose it is. Then there is some i^* such that $L_d = L(M_{i^*})$.

L_d is not r.e

$L_d = \{\langle i \rangle \mid M_i$ does **not** accept $\langle i \rangle\}$.

Theorem

L_d is not r.e.

Proof by contradiction. Suppose it is. Then there is some i^* such that $L_d = L(M_{i^*})$. Does $\langle i^* \rangle \in L_d$?

L_d is not r.e

$L_d = \{\langle i \rangle \mid M_i$ does **not** accept $\langle i \rangle\}$.

Theorem

L_d *is not r.e.*

Proof by contradiction. Suppose it is. Then there is some i^* such that $L_d = L(M_{i^*})$. Does $\langle i^* \rangle \in L_d$?

- If yes then M_{i^*} accepts $\langle i^* \rangle$ since $L_d = L(M_{i^*})$. But this is a contradiction since $\langle i^* \rangle \not\in L_d$ by definition of L_d.

L_d is not r.e

$L_d = \{\langle i \rangle \mid M_i \text{ does } \textbf{not} \text{ accept } \langle i \rangle\}$.

Theorem

L_d is not r.e.

Proof by contradiction. Suppose it is. Then there is some i^* such that $L_d = L(M_{i^*})$. Does $\langle i^* \rangle \in L_d$?

- If yes then M_{i^*} accepts $\langle i^* \rangle$ since $L_d = L(M_{i^*})$. But this is a contradiction since $\langle i^* \rangle \not\in L_d$ by definition of L_d.
- If no then M_{i^*} does not accept $\langle i^* \rangle$ since $L_d = L(M_{i^*})$. But this is a contradiction since $\langle i^* \rangle \in L_d$ by definition of L_d.

L_d is not r.e

$L_d = \{\langle i \rangle \mid M_i$ does **not** accept $\langle i \rangle\}$.

Theorem

L_d is not r.e.

Proof by contradiction. Suppose it is. Then there is some i^* such that $L_d = L(M_{i^*})$. Does $\langle i^* \rangle \in L_d$?

- If yes then M_{i^*} accepts $\langle i^* \rangle$ since $L_d = L(M_{i^*})$. But this is a contradiction since $\langle i^* \rangle \not\in L_d$ by definition of L_d.
- If no then M_{i^*} does not accept $\langle i^* \rangle$ since $L_d = L(M_{i^*})$. But this is a contradiction since $\langle i^* \rangle \in L_d$ by definition of L_d.

Thus we obtain a contradiction in both cases which implies that L_d is **not** r.e.

L_d is not r.e implies L_u is not decidable

Lemma

$\mathbf{L_d} \leq \bar{\mathbf{L}}_{\mathbf{u}}$. *That is, if there is an algorithm for* $\bar{\mathbf{L}}_{\mathbf{u}}$ *then there is an algorithm for* $\mathbf{L_d}$. *Equivalently, if there is an algorithm for* $\mathbf{L_u}$ *then there is an algorithm for* $\mathbf{L_d}$.

L_d is not r.e implies L_u is not decidable

Lemma

$L_d \leq \bar{L_u}$. *That is, if there is an algorithm for* $\bar{L_u}$ *then there is an algorithm for* L_d. *Equivalently, if there is an algorithm for* L_u *then there is an algorithm for* L_d.

Algorithm for L_d from an algorithm for L_u:

- Given $\langle i \rangle$ we simply feed $\langle M_i, i \rangle$ to algorithm for L_u
- If algorithm for L_u says NO return YES Else return NO

L_d is not r.e implies L_u is not decidable

Lemma

$\boldsymbol{L_d} \leq \bar{\boldsymbol{L_u}}$. *That is, if there is an algorithm for* $\bar{\boldsymbol{L_u}}$ *then there is an algorithm for* $\boldsymbol{L_d}$. *Equivalently, if there is an algorithm for* $\boldsymbol{L_u}$ *then there is an algorithm for* $\boldsymbol{L_d}$.

Algorithm for $\boldsymbol{L_d}$ from an algorithm for $\boldsymbol{L_u}$:

- Given $\langle \boldsymbol{i} \rangle$ we simply feed $\langle \boldsymbol{M_i}, \boldsymbol{i} \rangle$ to algorithm for $\boldsymbol{L_u}$
- If algorithm for $\boldsymbol{L_u}$ says NO return YES Else return NO

Corollary

$\boldsymbol{L_u}$ *is undecidable.*

Corollary

$\boldsymbol{L_{HALT}}$ *is undecidable.*

The Big Picture