Undecidability

23 April 2020
Where are we?
We’ve introduced our final model of a computer, the Turing machine.

We designed Turing machines at the level of individual states and transitions.

In your reading (and on Assignment 7), you’ve seen that you can also describe a Turing machine more abstractly, like writing pseudo-code.
The *Church–Turing thesis* states:

Every effective method of computation is either equivalent to or weaker than a Turing machine.
A language $L$ is **Turing-recognizable** if there is a TM $M$ such that

$$\forall w \in \Sigma^* . (M \text{ accepts } w \iff w \in L).$$

This is a *weak* notion of solving a problem:

- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ does not accept $w$.
  - It might reject or it might loop forever.

The class **RE** consists of all Turing-recognizable languages.
A language $L$ is **Turing-decidable** if there is a TM $M$ such that

$$
\forall w \in \Sigma^*. (M \text{ accepts } w \iff w \in L) \land (M \text{ halts on all inputs}).
$$

This is a **strong** notion of solving a problem:

- If $w \in L$, then $M$ accepts $w$.
- If $w \not\in L$, then $M$ rejects $w$.

The class $\mathcal{R}$ consists of all Turing-decidable languages.
If $Obj$ is an object, then $\langle Obj \rangle$ denotes some string representing $Obj$, similar to how it might be stored on disk or in memory on a real computer.

We can encode multiple objects as a single string. For example, if $M$ is a TM and $w$ is a string, then $\langle M, w \rangle$ is a string representing the pair of $M$ and $w$. 
There is a TM named $U$ that is a *universal Turing machine*.

$U$ takes as input a pair $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string.

$U$ does to $\langle M, w \rangle$ whatever $M$ does to $w$. 

![Diagram of Universal Turing Machine](image_url)
The language of $U$ is called $A_{\text{TM}}$:

$$A_{\text{TM}} = L(U) = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

$A_{\text{TM}}$ is the *acceptance language for Turing machines*. Because there is a Turing machine, $U$, that recognizes $A_{\text{TM}}$, we know $A_{\text{TM}} \in \text{RE}$.
Emergent property: Self-referentiality
PINOCCHIO

MY NAME ISN'T PINOCCHIO.

BRANSON REESE

WOW!
This is your brain on self-reference.
We can write programs that act on themselves.

As a fun case, we can write quines – programs that print out their own source code.

The way to do this varies by language, but it tends to be confusing!
E.g., a Scheme/Racket quine:

```scheme
((lambda (x)
   (list x (list 'quote x)))
 '((lambda (x) (list x (list 'quote x)))))
```

Less of a brain-teaser is to consider programs that read input files from disk (like a compiler does).

We can run such a program with its own source code file as input.
The fact that we can write quines isn’t a coincidence.

**THEOREM** (Kleene’s second recursion theorem). It is possible to construct TMs that perform arbitrary computations on their own “source code” (the string encoding of that TM).

In other words, any computing system that’s equal to a Turing machine possess some mechanism for self-reference.
If this is already making your brain hurt, it’s ok.

When we think about “real” computer programs, we can instead think of the program reading its own source code from disk.
If this is already making your brain hurt, it’s ok.

When we think about “real” computer programs, we can instead think of the program reading its own source code from disk.

selfie.py:

```python
def main(my_input):
    my_source = open("selfie.py").read()
    print(my_source)
```
Teaser: Self-reference lets machines compute on themselves. That will let them do cruel and unusual things.
A **self-defeating object** is an object whose essential properties ensure it doesn’t exist.

“I don’t know. I may be man’s best friend, but I’m my own worst enemy.”

Al Ross
The New Yorker
1991
**Question**: Why is there no largest integer?

**Answer**: Because if $n$ is the largest integer, what happens when we look at $n + 1$?
THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n + 1$.

Notice that $n < n + 1$.

But then $n$ isn’t the largest integer.

Contradiction!
THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer \( n \).

Consider the integer \( n + 1 \).

Notice that \( n < n + 1 \).

But then \( n \) isn’t the largest integer.

Contradiction!

We’re using \( n \) to construct something that undermines \( n \), hence the term “self-defeating”.
The general template for proving that $x$ is a self-defeating object is as follows:

- Assume that $x$ exists.
- Construct some object $f(x)$ from $x$.
- Show that $f(x)$ has some impossible property.
- Conclude that $x$ doesn’t exist.

The particulars of what $x$ and $f(x)$ are and why $f(x)$ has an impossible property depend on the specifics of the proof.
Beware

CLAIM: There is a largest integer.

PROOF SKETCH: Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there's no contradiction.
Beware

CLAIM: There is a largest integer.

PROOF SKETCH: Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there’s no contradiction.

Careful – we’re assuming what we’re trying to prove!
Beware

CLAIM: There is a largest integer.

PROOF SKETCH: Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there’s no contradiction.

Careful – we’re assuming what we’re trying to prove!

How do we know there’s no contradiction? We only checked one case!
You *cannot* show that a self-defeating object \( x \) exists by using this line of reasoning:

- Suppose that \( x \) exists.
- Construct some object \( g(x) \) from \( x \).
- Show that \( g(x) \) has no undesirable properties.
- Conclude that \( x \) exists.

The fact that \( g(x) \) has no bad properties doesn’t mean that \( x \) exists. It just means you didn’t look hard enough for a counterexample.
Teaser: Certain Turing machines can’t exist, as they’d be self-defeating objects.
Now we’ll see how the emergent properties of *universality* and *self-reference* bring us to the limits of decidability.
Undecidability
Suppose $M$ is a recognizer for some language. We have a string $w$, and we want to know if $w \in L(M)$. How could we check this?
If you want to know whether this is true…

\[ w \in L(M) \]

if and only if

\[ M \text{ accepts } w \]

…you can try to determine whether this is true.
Option 1: Run $M$ on $w$.

What could happen?

- $M$ could accept $w$. Great! We know $w \in L(M)$.
- $M$ could reject $w$. Great! We know $w \notin L(M)$.
- $M$ could loop on $w$. In this case we’ve learned nothing. 😞

So, this won’t always tell us whether $w \in L(M)$.

We’ll need a different strategy.
If you want to know whether this is true...

\[ w \in L(M) \]

if and only if

\[ M \text{ accepts } w \]

if and only if

\[ \langle M, w \rangle \in A_{TM} \]

...you can try to determine whether this is true.
Option 2: Use the universal Turing machine, which is a recognizer for $A_{TM}$!

Specifically, run $U$ on $\langle M, w \rangle$.

What could happen?

- $U$ could accept $\langle M, w \rangle$. Great! Then $w \in L(M)$.
- $U$ could reject $\langle M, w \rangle$. Great! Then $w \notin L(M)$.
- $U$ could loop on $\langle M, w \rangle$. In this case we’ve learned nothing. 😞

This won’t always tell us whether $w \in L(M)$ either. We’ll need a different strategy.
Option 2: Use the universal Turing machine, which is a recognizer for $A_{TM}$!

Specifically, run $U$ on $\langle M, w \rangle$.

What could happen?

- $U$ could accept $\langle M, w \rangle$. Great! Then $w \in L(M)$.
- $U$ could reject $\langle M, w \rangle$. Great! Then $w \notin L(M)$.
- $U$ could loop on $\langle M, w \rangle$. In this case we’ve learned nothing. 😞

This won’t always tell us whether $w \in L(M)$ either. We’ll need a different strategy.
**Option 3**: Build a *decider* for $A_{TM}$ rather than just a recognizer.

What could happen?

The decider could accept $\langle M, w \rangle$. Great! Then $w \in L(M)$.

The decider could reject $\langle M, w \rangle$. Great! Then $w \notin L(M)$.

How do we build this decider?
CLAIM. A decider for $A_{TM}$ is a self-defeating object. It therefore doesn’t exist.
Assume $A_{TM}$ is decidable, i.e., $A_{TM} \in \mathbb{R}$.

This means there’s a decider for $A_{TM}$. Let’s call it $D$:

![Diagram showing decider $D$ for $A_{TM}$ with transitions to Accept and Reject.]($M$, $w$) \rightarrow \text{Decider } D \text{ for } A_{TM} \rightarrow \text{Accept: } M \text{ accepts } w \rightarrow \text{Reject: } M \text{ does not accept } w

What can we do with $D$?
We’ve seen the idea that Turing machines can run other Turing machines as subroutines. This means we can write programs that use $D$ as a subroutine.

Since Turing machines are like programs, we can imagine that $D$ is a helper method like this:

```python
def will_accept(program, input):
    ...some implementation...
```
We’ve seen the idea that Turing machines can run other Turing machines as subroutines. This means we can write programs that use $D$ as a subroutine.

Since Turing machines are like programs, we can imagine that $D$ is a helper method like this:

```python
def will_accept(program, input):
    ...some implementation...
```
We’ve seen the idea that Turing machines can run other Turing machines as subroutines. This means we can write programs that use $D$ as a subroutine.

Since Turing machines are like programs, we can imagine that $D$ is a helper method like this:

```python
def will_accept(program, input):
    ...some implementation...
```
What can we do with the subroutine `will_accept`?

Ultimately, we’re trying to get a contradiction.

Specifically, we’re going to build a program that has some really broken behavior:

*It will accept its input if and only if it doesn’t accept its input!*
def will_accept(program, input):
    ...
some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True

Try running this on any input.

uh-oh.py:
What happens if the program accepts its input?

`uh-oh.py`:

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True
```
What happens if the program accepts its input?

**uh-oh.py:**

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True
```

It rejects the input!
What happens if the program rejects its input?

**uh-oh.py:**

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True
```
What happens if the program rejects its input?

**uh-oh.py:**

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True
```

*It accepts the input!*
def will_accept(program, input):  
  ...some implementation...

def main(my_input):
  my_source = open("uh-oh.py").read()
  if will_accept(my_source, my_input):
    return False
  else:
    return True

uh-oh.py:

The self-defeating object

Using that object against itself
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True

"The largest integer, n"

"The number n + 1."
"I cannot – yet I must! How do you calculate that? At what point on the graph do 'must' and 'cannot' meet?"

– Ro-Man in Robot Monster, 1953
THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n + 1$.

Notice that $n < n + 1$.

But then $n$ isn’t the largest integer.

Contradiction!

----------

**uh-oh.py:**

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True
```

---

THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n + 1$.

Notice that $n < n + 1$.

But then $n$ isn’t the largest integer.

Contradiction!
THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer $n$.

Consider the integer $n + 1$.

Notice that $n < n + 1$.

But then $n$ isn’t the largest integer.

Contradiction!

uh-oh.py:

```python
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
```

Assume there exists this object $x$ which has these properties that are too powerful to actually work.
THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer \( n \).

Consider the integer \( n + 1 \).

Notice that \( n < n + 1 \).

But then \( n \) isn’t the largest integer.

Contradiction!

Use the purported properties of \( x \) against itself to create a contradiction.

```
def will_accept(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("uh-oh.py").read()
    if will_accept(my_source, my_input):
        return False
    else:
        return True
```
THEOREM. There is no largest integer.

PROOF SKETCH. Suppose for the sake of contradiction that there is a largest integer. Call that integer \( n \).

Consider the integer \( n + 1 \).

Notice that \( n < n + 1 \).

But then \( n \) isn’t the largest integer.

Contradiction!

Thus the object \( x \) cannot exist!
THEOREM. $A_{TM} \not\in R$.

PROOF. By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$, which we can represent in software as a procedure `will_accept` that takes as input the source code of a program and an input, and returns true if the program accepts the input and false otherwise.

Given this, we could construct this program $P$:

```python
if will_accept(my_source, my_input):
    return False
else:
    return True
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$: If `will_accept(my_source, my_input)` returns true, this means that $P$ must accept its input $w$, but instead it rejects it. Otherwise, if `will_accept(my_source, my_input)` returns false, this means that $P$ must not accept its input $w$, but instead it accepts it.

In both cases, we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \not\in R$. □
THEOREM. $A_{TM} \notin R$.

PROOF. By contradiction; assume that $A_{TM} \in R$. Then there is some decider $D$ for $A_{TM}$. If this machine is given a TM/string pair, it will determine whether the TM accepts the string and report back the answer.

Given this, we could construct the following TM:

$$M = \text{“On input } w:$$

1. Have $M$ obtain its own description $\langle M \rangle$.
2. Run $D$ on $\langle M, w \rangle$ and see what it says.
3. If $D$ says that $M$ will accept $w$, reject.
4. If $D$ says that $M$ will not accept $w$, accept.”

Choose any string $w$ and trace through the execution of the machine, focusing on the answer given back by the machine $D$. If $D$ says that $M$ will accept $w$, notice that $M$ then proceeds to reject $w$, contradicting what $D$ says. Otherwise, if $D$ says that $M$ will not accept $w$, notice that $M$ then proceeds to accept $w$, contradicting what $D$ says.

In both cases, we reach a contradiction, so our assumption must have been wrong. Therefore, $A_{TM} \notin R$.
Regular languages

Context-free languages

All languages
What does this mean?

In one fell swoop, we’ve proven that

A decider for $A_{TM}$ is a self-defeating object.

$A_{TM}$ is undecidable, i.e., there is no general algorithm that can determine whether a Turing machine will accept a string.

$R \neq RE$, because $A_{TM} \notin R$ but $A_{TM} \in RE$

What do these three statements really mean? Why should you care?
1 Self-defeating objects

The fact that a decider for $A_{TM}$ is a self-defeating object is analogous to the philosophical question,

“If you know what you are fated to do, can you avoid your fate?”

If we have a decider for $A_{TM}$, we could use it to build a TM that determines what it’s supposed to do and then chooses to do the opposite!
The proof we’ve done says that there is no algorithm that can determine whether a program will accept an input.

Our proof just assumed that there was some decider for $A_{TM}$ and didn’t assume anything about how the decider worked.

No matter how you try to implement a decider for $A_{TM}$, you can never succeed!
What exactly does it mean for $A_{TM}$ to be undecidable?

*The only general way to find out what a program will do is to run it.*

This means that it’s provably impossible for computers to be able to answer most questions about what a program will do.
At a more fundamental level, the existence of undecidable problems tells us: *There is a difference between what is true and what we can discover is true.*

Given a Turing machine $M$ and a string $w$, one of these two statements is true:

- $M$ accepts $w$
- $M$ does not accept $w$

But since $A_{TM}$ is undecidable, there’s no algorithm that can always determine which of these statements is true!
This tells us *it is fundamentally harder to solve a problem than it is to check an answer.*

There are problems where, when the answer is “yes”, you can confirm it (build a recognizer), but where if you don’t have the answer, you can’t come up with it in a mechanical way (build a decider).
The Halting Problem
The most famous undecidable problem is the **Halting Problem**, which asks:

*Given a Turing machine $M$ and a string $w$, will $M$ halt when run on $w$?*

or, as a language,

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on } w\}$$

This is an **RE** language. (We’ll see why later.)

How do we know that it’s undecidable?
CLAIM: A decider for $HALT_{TM}$ is a self-defeating object. Therefore it doesn’t exist.
A decider for $\text{HALT}_{\text{TM}}$

Suppose we managed to build a decider for $\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on } w\}$:

We could represent this in software as a procedure `will_halt(program, input)`
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("halt.py").read()
    if will_halt(my_source, my_input):
        while True:
            # Infinite loop
            pass
    else:
        return True

halt.py:
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("halt.py").read()
    if will_halt(my_source, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True

Try running this on any input.
What happens if the program **halts on its input?**

### halt.py:

```python
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("halt.py").read()
    if will_halt(my_source, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True
```
What happens if the program **halts on its input?**

**halt.py:**

```python
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("halt.py").read()
    if will_halt(my_source, my_input):
        while True:
            pass  # Infinite loop
    else:
        return True
```

It loops on the input!
What happens if the program loops on its input?

`halt.py`:

```python
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("halt.py").read()
    if will_halt(my_source, my_input):
        while True:
            # Infinite loop
            pass
    else:
        return True
```
What happens if the program loops on its input?

halt.py:

def will_halt(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("halt.py").read()
    if will_halt(my_source, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True  # It halts on the input!
def will_halt(program, input):
    ...some implementation...

def main(my_input):
    my_source = open("halt.py").read()
    if will_halt(my_source, my_input):
        while True:  # Infinite loop
            pass
    else:
        return True

halt.py:
The self-defeating object

Using that object against itself
THEOREM. $HALT_{TM} \notin R$.

PROOF. By contradiction; assume that $HALT_{TM} \in R$. Then there is some decider $D$ for $HALT_{TM}$, which we can represent in software as a procedure `will_halt` that takes as input the source code of a program and an input, and returns true if the program halts on the input and false otherwise.

Given this, we could construct this program $P$:

```python
if will_halt(my_source, my_input):
    while True: pass
else:
    return True
```

Choose any string $w$ and trace through the execution of program $P$ on input $w$: If $will_halt(my_source, my_input)$ returns true, this means that $P$ must halt on its input $w$, but instead it loops on it. Otherwise, if $will_halt(my_source, my_input)$ returns false, this means that $P$ must not halt on its input $w$, but instead it halts and accepts it.

In both cases, we reach a contradiction, so our assumption must have been wrong. Therefore, $HALT_{TM} \notin R$. ■
THEOREM. $HALT_{TM} \not\in R$.

PROOF. By contradiction; assume that $HALT_{TM} \in R$. Then there is some decider $D$ for $HALT_{TM}$. If this machine is given any TM/string pair, it will then determine whether the TM halts on the string and report back the answer.

Given this, we could construct the following TM:

$M = “On input w:"

1. Have $M$ obtain its own description $\langle M \rangle$.
2. Run $D$ on $\langle M, w \rangle$ and see what it says.
3. If $D$ says that $M$ halts on $w$, go into an infinite loop.
4. If $D$ says that $M$ loops on $w$, accept.”

Choose any string $w$ and trace through the execution of the machine, focusing on the answer given back by machine $D$. If $D$ says that $M$ will halt on $w$, notice that $M$ then proceeds to loop on $w$, contradicting what $D$ says. Otherwise, if $D$ says that $M$ will loop on $w$, notice that $M$ then proceeds to accept $w$, so $M$ halts on $w$, contradicting what $D$ says.

In both cases, we reach a contradiction, so our assumption must have been wrong. Therefore, $HALT_{TM} \not\in R$. ■
CLAIM. $\text{HALT}_\text{TM} \in \text{RE}$.

IDEA. If you were certain that a Turing machine $M$ halted on a string $w$, could you convince me of that?

Yes – just run $M$ on $w$ and see what happens!
Moral: This isn’t necessarily Microsoft’s fault.
Regular languages
Context-free languages

$R$

$A_{TM}$

$HALT_{TM}$

RE

All languages
Acknowledgments

This lecture incorporates material from:

Nancy Ide
Keith Schwarz
Michael Sipser