Turing Machines and Decidability

Recap

The Church–Turing thesis claims:
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 $\forall w \in \Sigma^* . M$ 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 $\textbf{RE}$ consists of all Turing-recognizable languages.

A language $L$ is **Turing-decidable** if there is a TM $M$ such that $\forall w \in \Sigma^* . M$ accepts $w \iff w \in L$ and $M$ halts on all inputs.

This is a “strong” notion of solving a problem:
- If $w \in L$, then $M$ accepts $w$.
- If $w \notin L$, then $M$ rejects $w$.

The class $\textbf{R}$ consists of all Turing-decidable languages.

**Object encodings**

Think about files on your computer:
- Each file represents some data.
- Each files is encoded purely as a sequence of 0s and 1s.

If $\text{Obj}$ is an object, then $\langle \text{Obj} \rangle$ denotes some string representing $\text{Obj}$.
- Think of it as how you’d store Obj on disk.

We can encode multiple objects as a single string. E.g., if $M$ is a TM and $w$ is a string, then $\langle M, w \rangle$ is a string representing the pair of $M$ and $w$.
The Turing machine that runs other Turing machines is $U$, the *Universal Turing machine*.

When $U$ is run on an input of the form $\langle M, w \rangle$, where $M$ is a Turing machine and $w$ is a string, $U$ simulates $M$ running on $w$ and does whatever $M$ does on $w$.

- If $M$ accepts $w$, then $U$ accepts $\langle M, w \rangle$.
- If $M$ rejects $w$, then $U$ rejects $\langle M, w \rangle$.
- If $M$ loops on $w$, then $U$ loops on $\langle M, w \rangle$. 
A universal machine

An encoding of some Turing machine $M$ we want to run

A universal machine

The input to that program is some string

A universal machine

The input has the form $\langle M, w \rangle$ where $M$ is some TM and $w$ is some string

A universal machine

$\langle M, w \rangle$
The language of \( U \)

Recall that the language of a Turing machine is the set of all strings that the Turing machine accepts.

\( U \), when run on a string \( \langle M, w \rangle \), where \( M \) is a TM and \( w \) is a string, will

- accept \( \langle M, w \rangle \) if \( M \) accepts \( w \)
- reject \( \langle M, w \rangle \) if \( M \) rejects \( w \), and
- loop on \( \langle M, w \rangle \) if \( M \) loops on \( w \).

\[ L(U) = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \} = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in L(M) \} \]
The language of the $U$ is called $A_{\text{TM}}$:

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

$A_{\text{TM}}$ is the acceptance language for Turing machines.

Because there is a Turing machine, $U$, that recognizes $A_{\text{TM}}$, we know $A_{\text{TM}} \in \text{RE}$.

**Teaser:** This language, $A_{\text{TM}}$, has some interesting properties beyond what we’ve seen.

Why do we care about universality?

The existence of a universal Turing machine has both theoretical and practical significance.
Reason 1: It’s of practical significance

What happens if we replace the Turing machine input with a normal computer program?

Programs simulating programs

The fact that there’s a universal Turing machine combined with the fact that computers can simulate TMs and vice versa means that it’s possible to write a program that simulates other programs.

These programs go by many names:

- An interpreter like the Java Virtual Machine or Python.
- A virtual machine like VMWare or VirtualBox that simulates an entire computer.
Reason 2: It’s philosophically interesting

Can computers think?

On 15 May 1951, Alan Turing delivered a radio lecture on the BBC on the topic of whether computers can think.

Here’s what he had to say about whether a computer can be thought of as an electric brain:

The key idea behind the universal TM is that TMs can be fed as inputs into other TMs.

Similarly, an interpreter is a program that takes other programs as inputs.

Similarly, an emulator is a program that takes entire computers as inputs.

This hits at the core idea that computing devices can perform computations on other computing devices.
“In fact I think they [computers] could be used in such a manner that they could be appropriately described as brains. I should also say that

‘If any machine can be appropriately described as a brain, then any digital computer can be so described.’

This last statement needs some explanation. It may appear rather startling, but with some reservations it appears to be an inescapable fact.

It can be shown to follow from a characteristic property of digital computers, which I will call their universality. A digital computer is a universal machine in the sense that it can be made to replace any machine of a certain very wide class. It will not replace a bulldozer or a steam-engine or a telescope, but it will replace any rival design of calculating machine, that is to say any machine into which one can feed data and which will later print out results. In order to arrange for our computer to imitate a given machine it is only necessary to programme the computer to calculate what the machine in question would do under given circumstances, and in particular what answers it would print out. The computer can then be made to print out the same answers.

If now some machine can be described as a brain we have only to programme our digital computer to imitate it and it will also be a brain.”

Emergent property: Self-referentiality

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 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.

A self-defeating object is an object whose essential properties ensure it doesn’t exist.

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!
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.

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.

**Beware**

*Theorem:* There is a largest integer.

*Proof sketch:* Assume $x$ is the largest integer.

Notice that $x > x - 1$.

So there’s no contradiction.

*Teaser:* Certain Turing machines can’t exist, as they’d be self-defeating objects.
We stand at the foot of the mountains of undecidability.

Now we'll see how the emergent properties of universality and self-reference bring us to the limits of computation.

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

$M$ 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.

**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.

---

If you want to know whether this is true…

\( w \in L(M) \) if and only if
\( M \) accepts \( w \) if and only if
\( \langle M, w \rangle \in A_{TM} \)

…you can try to determine whether this is true.

---

What if we used a decider, not a recognizer?

Both \( U \) and \( M \) could accept \( \langle M, w \rangle \). Great! Then \( w \in L(M) \).
Both \( U \) and \( M \) could reject \( \langle M, w \rangle \). Great! Then \( w \notin L(M) \).
Both \( U \) and \( M \) 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 $A_{\text{TM}}$ 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?

Assume $A_{\text{TM}}$ is decidable, i.e., $A_{\text{TM}} \in \mathbb{R}$.

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

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...
```
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
```

Try running this on any 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
```

What happens if the program accepts 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
```

It rejects the input!

What happens if the program rejects 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
```

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

"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: $A_{TM} \notin R$.

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 $P$:

```
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, $A_{TM} \notin R$.

Note: This proof implicitly uses the diagonalization method. We’ll take a closer look at diagonalization soon.
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 $A_{TM} \not\in R$ but $A_{TM} \in RE$

What do these three statements really mean? I.e., 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!

2 $A_{TM} \not\in R$

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 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 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 you have the answer, 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 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,

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

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

How do we know that it's undecidable?

---

**Claim**: A decider for $HALT$ is a self-defeating object. Therefore it doesn't exist.

---

**A decider for $HALT$**

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

We could represent this in software as a procedure `will_halt(program, input)`

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

```python
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
```

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
```

It halts on the input!

Theorem: $\text{HALT} \notin \mathbb{R}$.

Proof: By contradiction; assume that $\text{HALT} \in \mathbb{R}$. Then there is some decider $D$ for $\text{HALT}$, 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, $\text{HALT} \notin \mathbb{R}$.
Claim: \( \text{HALT} \in \text{RE} \).

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.

Acknowledgments

This lecture incorporates material from:

W. Daniel Hillis
Nancy Ide
Keith Schwarz
Michael Sipser