Turing Machines and Computability

COMP2600 — Formal Methods for Software Engineering

Presented by Jinbo Huang

The following people have worked on these slides in previous years: Ranald Clouston, Jeremy Dawson, Malcolm Newey, Clem Baker-Finch
Alan Turing (1912–1954)

Let’s start by looking at the life and works of the extraordinary Alan Turing...
**Scientific contributions**

1936: Introduced *Turing machines* and the study of *computability* (our final topics);

1950: The *Turing test* set up artificial intelligence as a concrete research problem;

1952: Pioneering work on computation in *nature*;

Also: Key figure in the invention of the earliest modern computers.

Remarkable as these advances were, there is another facet to his life, kept secret until well after his death...
Turing at War

In World War Two, Nazi Germany used the most sophisticated cryptography system ever devised to communicate secret orders - the Enigma machine.

A British codebreaking effort was assembled at Bletchley Park, with Turing as their chief scientific genius.

Their efforts to crack Enigma has been credited with shortening World War Two by as much as two years!

- Many of the ideas and technologies of Bletchley Park fed directly into the invention of modern computers.

Death and Legacy

In 1952, less than a decade after his heroic efforts for his country in WWII, Turing was prosecuted for the ‘crime’ of homosexuality.

He was sentenced to chemical castration, and subsequently committed suicide in 1954.

He is now widely recognised as the father of computing:

- The Turing award is computer science’s equivalent to the Nobel Prize;
- UK government apologised for his persecution in 2009;
- 2012 is Turing Year.
Turing Machines – Introduction

Alan Turing’s 1936 paper

On computable numbers, with an application to the Entscheidungsproblem provides a remarkably simple mathematical machine that is claimed to be able to simulate any action of a computer.

But wait a minute - wasn’t 1936 before computers were even invented..?

(ENIAC in 1946 is generally considered the first modern computer.)

So what was Turing talking about?
A model for ‘computers’

In 1936 a computer was not a machine, it was a profession.

Turing set himself the task of giving a precise mathematical definition of everything that such ‘computers’ were capable of.

His 1936 paper justifies his definition by references to ‘states of mind’ etc.

(Section 9.1 of his paper, justifying his definition, is quite readable; see http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf)

But no computer (as we now understand the word) has ever been built that is more powerful than a Turing Machine; Turing seems to have discovered the limits of what is possible to compute by any means, although this can never be proved.
A PDA with its auxiliary store is *almost* a whole computer, except we can only directly access the symbol on the top of the stack.
We generalise the PDA into a Turing Machine by using a tape memory for the store instead of a stack memory.

We can access an arbitrary memory symbol by moving the tape head.
**Single tape Turing Machines**

Two tapes are unnecessary; we simplify our picture by using a single tape for both input data and auxiliary storage.

Note that we can go back and forth along this tape, and *write* on it, rather than just reading it left-to-right as with FSAs, PDAs.

The tape is assumed to be *infinite* in both directions.
Turing Machines as language recognisers

Turing Machines are deterministic - at each point there is at most one action that is legal.

- If there is no action that is legal, the TM halts.
- If a TM halts in a final state, it has accepted the original input. The set of strings accepted by a TM is its language.
- If it halts in a non-final state, this is an ‘error’, i.e. the input is rejected.
- A TM may also loop forever...

The languages recognisable by some TM are called the recursively enumerable languages.
Output

A TM is supposed to be able to perform any computable function.

So we are interested in output more sophisticated than yes/no.

If a TM halts in a final state then whatever is left on the tape will be its output.

(The tape is infinite in both directions, but a halted computation can only have written on a finite part of it, so it is that non-blank bit ‘in the middle’ that we regard as the output.)
Turing Machine – formal definition

A Turing Machine has the form \((Q, q_0, F, \Gamma, \Sigma, \Lambda, \delta)\), where

- \(Q\) is the set of states, \(q_0 \in Q\) is the initial state and \(F \subseteq Q\) are the final states;
- \(\Gamma\) is the set of tape symbols, \(\Sigma \subseteq \Gamma\) is the set of input symbols, and \(\Lambda \in \Gamma \setminus \Sigma\) is the blank symbol.
- \(\delta\) is a (partial) transition function

\[
\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}
\]

\[
\delta : (\text{state, tape symbol}) \rightarrow (\text{new state, new tape symbol, direction})
\]

direction tells the read/write head which way to go: Left, Right, or Stay.

- **Exercise:** Show that stationary moves are unnecessary (every TM can be converted to one that accepts same language without stationary moves)
Running a TM

Initialisation:

- some input (a finite string over $\Sigma$) is written on the tape;
- every other tape cell is a blank – $\Lambda$;
- the read/write head sits over the left-most cell of the input (or over any $\Lambda$ if the input is $\varepsilon$);
- the state is the start state $q_0$.

The TM then runs according to the deterministic transition function $\delta$, halting if it reaches a final state, or a state-symbol pair for which $\delta$ is undefined.
Language of a TM

The language of a TM is the set of strings accepted by the TM

- Only way to accept a string: Enter a final state

Two ways to reject a string

- Halt in a non-final state (no move defined)
- Run forever
A Graphical Representation of the Transition Function

Similar to FSA, annotate transition edges with commands for accessing tape.

- **Numerator**: symbol read from tape.
  - $\Lambda$ means the tape is blank at that position.

- **Denominator**: symbol written, direction of head movement.
  - direction one of L, R, S for Left, Right, Stay.
What does it do?

- Adds two to a unary number!
- First phase travels to the right of the input (why bother?).
- Second phase writes two extra 1’s.
- Halts in the final state $H$. 
Tracing a TM

We write instantaneous description, or ID,

\[ x_0 x_1 x_2 \cdots x_{i-1} [Q] x_i x_{i+1} \cdots x_n \]

to indicate that the head is currently reading the \( i \)'th token of a string, and is in state \( Q \):

\[
[S_0]11 \Rightarrow 1[S_0]1 \\
\Rightarrow 11[S_0]\Lambda \\
\Rightarrow 111[S_1]\Lambda \\
\Rightarrow 111[H]1
\]

ID encodes TM state, whole contents of the tape, plus tape head position
Suppose the input to this TM contains a token other than 1.

The TM would halt in state $S_0$, as there is no arc telling us what to do if we meet such a token (this is the job of the rightwards scan).

This is an error - the input is rejected.

(The language of this TM is $\{1^n \mid n \geq 0\}$.)
Harder Problems

- Incrementing a binary number

\[
\begin{array}{c}
\text{1 0 0 0 1 0 1 1}
\end{array}
\]

- Adding numbers

\[
\begin{array}{c}
\text{1 1 0 0 0 1 \# 1 1 1 0 1 1}
\end{array}
\]

The \# symbol is a delimiter marking the gaps between numbers.

- Multiplication - and so on!
Incrementing a binary number

| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

Solution:

- From state $S_0$:
  - Transition on 0: $0 \rightarrow 0, R$
  - Transition on 1: $1 \rightarrow 1, R$

- From state $S_1$:
  - Transition on 0: $0 \rightarrow 1, S$
  - Transition on 1: $1 \rightarrow 0, L$

- Final state: $H$
Decrementing a Binary Natural Number

Note that if this TM reads $\Lambda$ in state $S_1$ it will fail.

This is because we cannot decrement a number that is all 0’s.

**Exercise:** can you design a TM for decrementing that rejects input with leading 0’s, and that eliminates any leading 0 created?
How to add two numbers?

Input e.g. $\overline{10100101}$ $\# \overline{100101010}$

Go back and forth between $m$ and $n$, decrementing one (until this fails) and incrementing the other.

We decrement $m$, and increment $n$, because $n$ will expand leftwards.

$m$ gets changed to $11\ldots 1$, $n$ is replaced by the sum.

We’ll finally delete the $\#11\ldots 1$ on the right.
How to add two numbers? ctd.

COMP 2600 — Turing Machines
How to multiply two numbers?

Tape looks like e.g. 
\[
\begin{array}{c}
\#10100101\#100101010 \\
p & n & m
\end{array}
\]

Repeatedly decrement \(m\) (until this fails) and add \(n\) to \(p\) (\(p\) is initially blank)

We must modify the addition routine to not erase the number \(n\) being added.

**Solution:** Two new tape symbols, 0’ and 1’.

Before each addition stage, change each 1 in \(n\) to 1’.

When decrementing \(n\), swap 0’s and 1’s as usual, but keep the primes.

When finished adding \(n\) to \(p\), go back and use the primes to restore \(n\) (sending each 1’ to 1, and each 1 to 0).

This sort of trick is typical for TM computations.
Using States to Remember a Tape Symbol

We sometimes want to ‘remember’ a tape symbol.

E.g. design a TM that takes a binary string, scans for the first 0, and inserts a 1 before it.

This involves shifting everything after the first 0 to the right by one cell (or everything before it left one) - a very common TM operation.

We’ll use our states to remember what token should be inserted next.
Inserting a Token

The state $S_0$ remembers that the next token to insert is a 0, and the state $S_1$ remembers 1.
Programming Issues – Data

- Number representation: Usually use unary or binary for integers.

- Vocabulary
  Could be anything, but 0, 1, # often all that is used.
  Other characters can be represented as strings of bits.

- Variables, Arrays, Files
  Use markers on the tape to separate values.
Programming Issues – Control

• Common techniques include:
  Scan to blank, or to find a symbol.
  Insert a symbol in the tape, as in previous slide.
  Delete a symbol and close up the gap (Exercise!)

• Composition
  If you have a TM to multiply by 3 and one to multiply by 5, put them together to multiply by 15.

• Decisions (conditional computation)
  We can branch on tape symbol 0 or 1 (or T or F).

• Loops — several of our previous examples had these.
Church-Turing Thesis

The Church-Turing thesis claims that any computable function can be calculated by a Turing machine.

Equivalently, any problem solvable by algorithm can be solved by a Turing Machine.

This thesis could in theory be formally disproved – simply one find one computable function that a TM cannot express.

It can never be formally proved.

But we can accumulate enough evidence to make it look very very likely to be true.
Church-Turing Thesis: Argument 1

Turing started with an analysis of how computers (= people who compute) perform their task.

He argued that everything computers did was essentially a TM operation (write something down, rub something out, move your attention to a new ‘cell’ on your paper, change ‘state of mind’ to start a new part of your task).

He further argued that it was impossible for a computer to do anything more powerful (e.g. distinguish between infinitely many different tape symbols).

It is this intuitive argument that really persuaded people in the 1930s that Turing had discovered a true model of all computation.
The next reason to believe the Thesis is that most sensible extensions to TMs turn out not to add power.

- **Multi-tape machines:** Adding an extra tape for scratch space may make the TM easier to program, but provides no extra power.

- **Multi-head machines:** Adding multiple heads to access multiple symbols at once may make the TM easier to program, but provides no extra power.

- **Non-determinism:** simulate with a breadth-first search.

If we had *infinitely* many tapes, heads, or branching non-determinism we’d get more power, but this is to enter the realm of fantasy...
Church-Turing Thesis: Argument 3

The third argument – perhaps the one that persuades the most computer scientists – is that we now have *many* models of computation.

They have been created for many different reasons over the years, often quite independently of any considerations about TMs or computability.

It turns out that *none* of these models of computation can compute anything that a TM cannot.

In fact, many of them compute the exact same set of functions as TMs can (they are said to be Turing-complete).
**Argument 3A – $\lambda$-calculus**

The **Church** in Church-Turing comes from Alonzo Church, inventor of the $\lambda$-calculus.

The $\lambda$-calculus was actually the **first** computational model claimed to be able to express all computable functions.

But why should anybody believe **that** claim?

It turns out that TMs can express $\lambda$-term evaluation, and that we can define a $\lambda$-term to simulate any TM – they are equally powerful!

This ‘coincidence’, and the intuitive appeal of Turing’s model, convinced many people of Church’s original claim.

Hence – Church-**Turing** thesis.
Argument 3B – grammars

Any language generated by a grammar (right-linear, context-free, context-sensitive, unrestricted) can be recognised by a TM.

**Strategy:** write the *start symbol* $S$ on our tape (away from our input).

Do a breadth-first search of all possible derivations from $S$.

Every time we reach a *sentence* (no non-terminals), check it against the input.

If they match, halt in a final state.

If $S$ can generate the input we’ll eventually find such a match (if not, we may well loop forever).
Argument 3B – grammars ctd

Remember **unrestricted grammars**?

Every production has form $\alpha \rightarrow \beta$ where $\alpha$ contains at least one non-terminal and $\beta$ could be anything.

It turns out that any TM can be turned into an unrestricted grammar that generates the same language the TM accepts!

**Strategy:** the non-terminals will be the TM states.

We then run the TM *backwards* (we are interested in inputs, not outputs):

```
T 0
```

```
\rightarrow
```

```
I,R
```

```
U
```

yields the production $1U \rightarrow T0$.

(There is a bit more to it, e.g. how to manage the blanks $\Lambda$.)
Argument 3C – Computers & Programming Languages

No computer ever invented, or machine language or programming language built on top of one, has ever been found to be capable of anything a TM is not.

In fact every useful programming language that I know of is Turing-complete.

(Even the tiny language we defined for the Hoare logic lectures.)
Wait – are real computers really Turing-complete?

- A Turing Machine has an *infinite tape*.

- My computer has 4,294,967,296 bits of RAM (4 Gigabytes); A lot – but not infinite!

- Physical computing devices must have finite memory, so they can only exist in a finite number of states. They are Finite State Automata.

- .. but the number of states can be very, very large. How big is $2^{4,294,967,296}$? – *pretty big!*

- So it isn’t very feasible to treat a real computer as a FSA for most practical purposes *(although FSA are used for some hardware formal methods)*.

- Physical computers *approximate* Turing Machines.
**Metaprograms**

A *metaprogram* is a computer program that treats other programs as data (input or output).

This may seem theoretical (and they are very theoretically important) but practical examples also abound:

- **Lexical analysers** and **parsers**;
- **SPARK Examiner**, which automatically proves correctness properties of programs written in SPARK Ada;
- **Code generation** which automatically produces codes from e.g. a Z specification.

We will now look at Turing Machines that accept other TMs as input.
Coding a TM onto tape

For this to work we need a method to convert TMs into tape format.

In fact, we will convert TMs into **binary strings**: 

Order the states $S_1, S_2, \ldots, S_k$, where $S_1$ is the start state and $S_2$ the unique final state (a unique final state $\neq$ the start state doesn’t reduce expressive power).

Order the tape symbols $X_1, X_2, X_3, \ldots, X_l$ where $X_1$ is 0, $X_2$ is 1, and $X_3$ is $\Lambda$.

Order the directions $\{L, R, S\}$ as $D_1, D_2, D_3$ respectively.

Then each transition $\delta(S_i, X_j) = (S_k, X_l, D_m)$ is sent to the string

$$0^i \ 1 \ 0^j \ 1 \ 0^k \ 1 \ 0^l \ 1 \ 0^m$$

so the number of 0’s carry the information, and solitary 1’s act as separators.
Coding a TM onto tape ctd.

A TM probably has more than one transition, so if $C_1, C_2 \ldots, C_n$ is the list of coded transitions (in any order), the TM is

$$C_1 \ 11 \ C_2 \ 11 \ \cdots \ 11 \ C_n$$

So $11$ is used as a separator between transitions.

We will later be interested in TMs that accept the coding of a TM and another argument, also a binary string.

We will separate the TM encoding, and the other binary string, with the string $111$. 
Coding a TM onto tape – example

The transitions are

- 0 1 0 0 1 0 1 0 0 1 0 0
- 0 1 0 0 0 1 0 0 0 1 0 0
- 0 0 0 1 0 0 0 1 0 0 1 0 0
- 0 0 0 1 0 0 0 1 0 0 1 0 0 0

Recall: States, Symbols, Directions.

So the TM as a whole is encoded by the binary string

010010100100 11 010001000100100 11 00010001001001000
String and Machines

Each TM may have a few different binary encodings (re-order the states, the tape symbols, or the transitions).

Most binary strings don’t code for any TM, e.g. 110.

Given a binary string $w$, we will ask two questions:

- Is $w$ an encoding of a TM? (easy)
- If $w$ is an encoding of a TM $M$, does $M$ accept the string $w$, or reject it?

This may seem a bizarre question, but this slipperiness between machines and data is actually very important, as is the idea of ‘feeding a TM to itself’.
A language that is not recursively enumerable

Let $L_d$ be the set of binary strings $w$ such that

\[ w \text{ codes for a Turing Machine } M \text{ that rejects } w \]

recalling that rejection may mean halting in a non-final state, or failure to halt.

We will show that $L_d$ is not recursively enumerable; i.e. no TM can be built that accepts exactly this language:

Suppose for contradiction that a TM $M$ existed with $L(M) = L_d$.

$M$ has a binary encoding $C$ (it may have many; just pick one).

Simple question: is $C \in L_d$?
A language that is not recursively enumerable ctd.

- If $C \in L_d$ then $M$ accepts $C$, because $M$ accepts all strings in $L_d$
  - But $L_d$ contains only encodings $w$ for TMs that reject $w$ – so $C \notin L_d$.
  - **Contradiction.**

- So $C$ must *not* be in $L_d$.
  - Then $M$ rejects $C$ because $M$ rejects all strings not in $L_d$.
  - But $L_d$ contains all the encodings $w$ for TMs that reject $w$ – so $C \in L_d$.
  - **Contradiction.**

We get a contradiction either way, so we must go to the start of our proof and reject our sole assumption.

*There cannot exist a TM whose language is $L_d$.*
Reflecting on this example

The language $L_d$ is fairly artificial, designed to give an easy contradiction proof.

But we should still reflect on something extraordinary here.

$L_d$ was not (that) hard to define, and there is nothing paradoxical about its definition; it is a subset of $\{0, 1\}^*$ defined via mathematical predicate like so many others we’ve seen in this course.

Yet there’s no TM to accept $L_d$ (even though we don’t require the TM to always halt).

Some languages (problems) are accepted by TMs, but none that always halt
The Universal Language

A simple and important question:

- Given a Turing machine $M$ and input $w$, does $M$ accept $w$?

For some $(M, w)$ pairs it’s pretty easy to answer – but is there an algorithm that could take any $M$ and $w$ and give you the answer?

Set of pairs $(M, w)$ where $M$ accepts $w$ is called the universal language, $L_u$.

Is there a TM that accepts $L_u$?
Universal TM

Working out what a TM $M$ does on an input $w$ does not require cleverness, merely recall of the rules and attention to detail – it is an algorithm.

And an any algorithm can be carried out by a TM!

We can therefore build a *Universal Turing Machine* that accepts pairs $(M, w)$ where $M$ is an (encoded) Turing machine, $w$ is any string, and then runs exactly as $M$ would on $w$.

*How?* just keep track of the current state and head position in your input, scan the TM instructions, and follow them.

This machine will accept exactly the universal language $L_u$. 
**$L_u$ is recursively enumerable**

If the answer to our question is ‘yes’ then we’ll eventually get that answer

(This is better than the language $L_d$, about which we could never guarantee any sort of answer at all)

But if the answer is ‘no’ then the universal TM may not tell us that – it may loop forever while we sit there wondering if a ‘yes’ might arrive at any moment...

Can there be some other TM that accepts $L_u$ and always halts with a ‘yes’ or ‘no’?
Recursively Enumerable vs. Decidable Languages

Recall that a recursively enumerable language $L$ is one that is accepted by some TM.

Such a TM may get stuck on an infinite loop on something outside of $L$ though.

Another view – you can enumerate all members of an r.e. language, but never be sure whether a string not in your list-so-far will eventually appear or not.

A decidable language $L$ is one that is accepted by some TM that halts on all input.

Synonymous terms: recursive = decidable.

We will see that while the universal language $L_u$ is recursively enumerable, it is not decidable (i.e., is undecidable).
The universal language $L_u$ is undecidable

Suppose we had a TM $U$ that accepts $L_u$ and always halts.

Then we can build another TM $Paradox$ that takes an encoded TM $M$ and

- If $U$ accepts $(M,M)$ (i.e., if $M$ accepts its own encoding), loop forever
- If $U$ rejects $(M,M)$, accept

**Observation:** - $Paradox$ accepts precisely $L_d$

**Contradiction** – We already know $L_d$ is not accepted by any TM
Reflections on the proof

Positive decidability proofs involve giving an algorithm, and proving it correct.

Negative decidability proofs are often structured like the last slide’s proof:

- **Assume** a TM existed solving the problem $P$ you are interested in;
- **Use** that TM to build another TM that would solve a problem $Q$ which is known to be unsolvable

This is called reducing problem $Q$ to problem $P$ (i.e., if you could solve $P$, then you could solve $Q$).

Often our reduction strategy is to say ‘if this TM existed, then we could decide the universal language $L_u$’.

In fact we can develop a library of undecidable problems that we can use to prove new undecidable problems.
Turing Machines with Nonempty Languages

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

$L_{ne}$ is undecidable:

- Let’s reduce $L_u$ to $L_{ne}$
- For any TM $M$ and string $w$ we can define a TM $M_w$ that ignores its input and runs like $M$ would on $w$
- $M_w$ accepts all strings if $M$ accepts $w$, and accepts nothing otherwise
- If there were a TM to decide $L_{ne}$, giving it $M_w$ would tell us whether $M$ accepts $w$
- So we would have a procedure (TM) to decide the universal language $L_u$
- But $L_u$ is undecidable, so $L_{ne}$ is undecidable
Rice’s Theorem

All nontrivial properties about the language of a TM are undecidable

Examples of properties: being nonempty, being finite, being regular, being context-free, having only even-length strings, etc.

Property is trivial if all RE languages have it, or no RE languages have it; otherwise it’s nontrivial (RE = recursively enumerable)

Hence following are all undecidable

- Set of (codes for) TMs that accept a finite language
- Set of (codes for) TMs that accept a regular language
- Set of (codes for) TMs that accept only even-length strings
Rice’s Theorem

Applies only to the language of a TM

Problems about other aspects of a TM can be decidable

• Set of (codes for) TMs that have five states
• Set of (codes for) TMs that sometimes make at least five moves

Informal equivalent formulations of the same problems

• Does the given TM have five states?
• Does the given TM ever make at least five moves?
Two Ways to Prove Undecidability

Options 1: Use Rice’s theorem where it’s applicable

- Must ensure that the problem is, or can be formulated as, a set of TMs whose language has a certain property
- Must argue that the property is nontrivial, i.e., some have it, some don’t

Option 2: Reduce a known undecidable problem (such as $L_u$) to the problem that you want to show is undecidable

- See example on Slide 54
The Halting Problem

In some texts, the Halting Problem is discussed, in place of the universal language, as the quintessential undecidable problem.

This is because in earlier definitions, TMs accepted by halting, not by final state.

Hence the halting problem in that context is equivalent to the universal language $L_u$ we’ve studied here.
Back to the Entscheidungsproblem

Recall that Turing’s original paper suggested an application to the *Entscheidungsproblem*

The Entscheidungsproblem asked for an algorithm that could compute the answer to any mathematical question specified in first-order logic.

If there was such an algorithm, mathematicians might be made as redundant as computers (in the old sense) were!

(Although efficiency might also be an issue...)

However Turing showed that assuming the existence of a TM that computes the Entscheidungsproblem leads to a contradiction.

(Again, Alonzo Church has precedence here, using the $\lambda$-calculus).
But... even decidable problems may not be feasible

- Each of the inner sets is a tiny proportion of the set that contains it.
Time Complexity

- Negating a boolean value takes constant time.

- Multiplication of integers takes an amount of time proportional to the square of the total number of digits.
  
  If we double number of digits, then the algorithm will take four times as long to complete.
  
  If we set $n$ to be the total number of digits, we can say that multiplication has a complexity of order $n^2$.

- Smart matrix multiplication is order $n^{2.376}$

- Some theorem-proving algorithms are order $2^{2^n}$

Problems are generally considered feasible if they can be solved in polynomial time, i.e. within $n^k$ for some $k$. 
P vs. NP

- The complexity class $P$ is the class of problems that can be *solved* in polynomial time.

- The complexity class $NP$ is the class of problems whose solutions can be *verified* in polynomial time.

Coming up with a solution within a time bound seems intuitively harder than checking someone else’s solution in that time.

- We won’t take 3 hours to mark each of your final exams!

But as yet there is no *proof* that $P = NP$ or $P \neq NP$.

This is the biggest open problem in all of computer science.

One of the Clay Mathematics Institute’s million dollar ‘Millennium Prize’ Problems (7 set in 2000; 1 solved so far).  *Good luck!*