ECM3404: Logic and Computation

Antony Galton

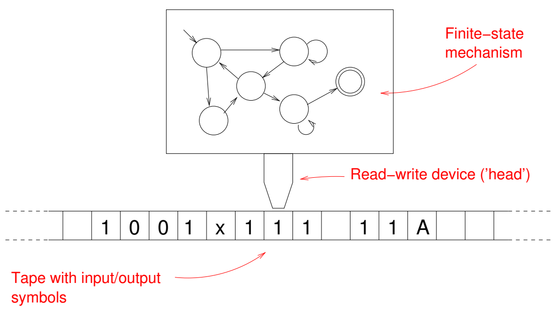
Turing Machines

What is a Turing Machine?

A Turing Machine is a finite-state automaton augmented with an unlimited amount of memory in the form of an extensible linear *tape*, divided into discrete *cells*:

- It can *read* from the tape, and *write* to it, one cell at a time.
- The initial contents of the tape is the *input*.
- The machine's action at each step is determined by its state and what it reads from the current cell. The action consists of
  - moving to a new state (or not),
  - replacing the tape symbol it has read (or not), and
  - moving to a new tape cell (or not).
- The computation continues until the machine reaches the designated *halt state*.
- The *output* is then read off from the tape

Picture of a Turing Machine

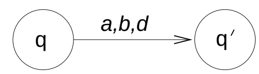


Inside the Turing Machine

The state-transition diagram for a Turing Machine is very similar to that for a finite-state automaton.

The differences are that:

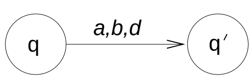
1. Whereas for a FA each transition is labelled just with the input symbol, for a TM it is labelled with the input symbol together with the output symbol and the tape shift:



2. Whereas for a FA any state can be an accepting state, for a TM there is just one accepting state (the *halt state*), from which there can be no transitions.

Reading a transition

The transition

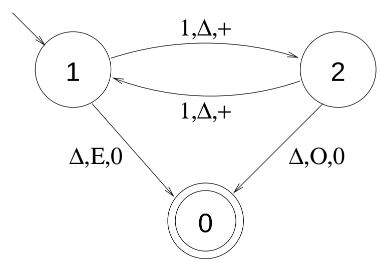


is to be understood as follows:

- The transition is triggered when the machine is in state q and reading symbol a on the tape.
- The transition consists of the following actions:
  1. Symbol a is replaced by symbol b. NOTE: We use Δ to represent a null symbol (i.e., the cell is empty).
  2. The machine moves its read/write device along the tape by the displacement d. This may be -1 (move one cell left), 0 (stay put), or 1 (move one cell right).
  3. The machine goes into state q'.

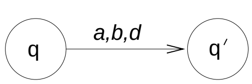
Example: A Parity Checker

This machine, when given as input a string of n 1s, will output O or E depending on whether n is odd or even.



Representing a TM by means of a quintuple listing

The transition



can be represented by the **quintuple**

$$q, a, q', b, d$$

The Turing Machine is completely specified by a listing of its quintuples.

Exercise: Write down the listing for The Parity Checker on the previous slide.

1,	1,	2,	Δ,	1
1,	Δ,	0,	E,	0
2,	1,	1,	Δ,	1
2,	Δ,	0,	O,	0

Specification of a Turing Machine

To define a TM uniquely, you must specify the **states**, **alphabet**, and **transitions**, as follows:

- **States:** There must be a **start state** (q_0) and a **halt state** (h), where $h \neq q_0$. The set of non-halt states (including q_0) is designated Q.
- **Alphabet:** There must be a finite **input alphabet** Σ_I, and, disjoint from this, a finite (possibly empty) **auxiliary alphabet** Σ_A. Together with the symbol Δ (indicating a blank cell) these make up the **tape alphabet** $\Sigma_T = \Sigma_I \cup \Sigma_A \cup \{\Delta\}$.
- The **transition function** (δ) which maps a (non-halt) state/symbol pair onto a triple consisting of the next state, the new symbol, and the shift ($\delta : Q \times \Sigma_T \to (Q \cup \{h\}) \times \Sigma_T \times \{-1, 0, 1\}$).

In our examples we shall designate states by natural numbers, with $q_0 = 1$ and $h = 0$.

The configuration of a Turing machine

At any stage during a Turing machine computation, the machine is in a particular **configuration**. This is specified by means of four ingredients:

- The state (q)
- The symbol in the currently scanned cell (a)
- The string of symbols running left from the currently scanned cell, up to the leftmost non-blank cell ($b_1 b_2 \ldots b_m$)
- The string of symbols running right from the currently scanned cell, up to the rightmost non-blank cell ($c_1 c_2 \ldots c_n$)

This configuration will be represented as follows:

$$\cdots \Delta b_m b_{m-1} \cdots b_1 \ \underset{q}{\boxed{a}} \ c_1 c_2 \cdots c_n \Delta \cdots$$

Special types of configuration

The starting configuration with input $i_1 i_2 \cdots i_n$ is

$$\cdots \Delta \ \underset{1}{\boxed{i_1}} \ i_2 i_3 \cdots i_n \Delta \cdots$$

If the input is the null string Λ, the configuration is

$$\cdots \Delta \ \underset{1}{\boxed{\Delta}} \ \Delta \cdots$$

A halting configuration is

$$\cdots \Delta b_m b_{m-1} \cdots b_1 \ \underset{0}{\boxed{a}} \ c_1 c_2 \cdots c_n \Delta \cdots$$

The output here would usually be taken to be $a c_1 c_2 \cdots c_n$ (but may vary depending on the convention used for a particular TM).

Computation steps

Assume the machine is in configuration

$$\cdots \Delta b_m b_{m-1} \cdots b_1 \ \underset{q}{\boxed{a}} \ c_1 c_2 \cdots c_n \Delta \cdots$$

Then the configuration at the next computation step will be:

- If $\delta(q, a) = (q', a', 1)$:

$$\cdots \Delta b_m b_{m-1} \cdots b_1 a' \ \underset{q'}{\boxed{c_1}} \ c_2 c_3 \cdots c_n \Delta \cdots$$

- If $\delta(q, a) = (q', a', -1)$:

$$\cdots \Delta b_m b_{m-1} \cdots b_2 \ \underset{q'}{\boxed{b_1}} \ a' c_1 c_2 \cdots c_n \Delta \cdots$$

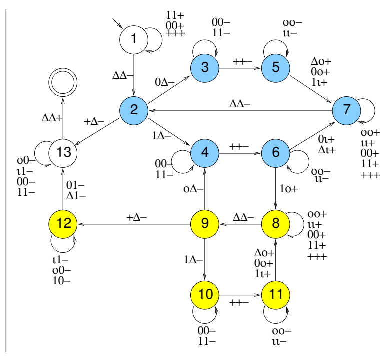
- If $\delta(q, a) = (q', a', 0)$:

$$\cdots \Delta b_m b_{m-1} \cdots b_1 \ \underset{q'}{\boxed{a'}} \ c_1 c_2 \cdots c_n \Delta \cdots$$

A Binary Adder

Specification
Input: $\langle m \rangle + \langle n \rangle$
Output: $\langle m + n \rangle$
where $\langle x \rangle$ is the binary numeral for integer x.

$Q = \{1, 2, \ldots, 13\}$
$\Sigma_I = \{0, 1, +\}$
$\Sigma_A = \{\iota, o\}$
$q_0 = 1$
$h = 0$



How the Binary Adder works

It fetches bits from $\langle n \rangle$ one by one and adds them to the corresponding bits in $\langle m \rangle$, using ι, o to represent 1, 0 respectively.

It does this in either a 'no carry' condition (states 2–7) or a 'carry' condition (states 9-11), according to the rules tabulated below.

When there are no more bits in $\langle n \rangle$ (states 12, 13), any remaining carries are handled, o, ι are changed to 0, 1, and the machine halts.

	0, no carry	0, carry 1, no carry	1, carry
0	0, no carry	1, no carry	0, carry
1	1, no carry	0, carry	1, carry

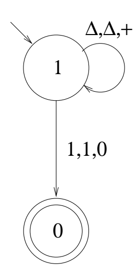
Possible behaviours of a Turing machine

When a Turing machine is run, starting with a given input tape, there are three possibilities for what ultimately happens:

1. The machine eventually reaches the halt state, delivering an output: the computation **succeeds**.
2. The machine gets stuck in a non-halting state: the computation **fails**. (This will happen if there is no quintuple beginning $q, a, \ldots$)
3. The machine fails to halt, endlessly repeating some sequence of states: the computation **loops**.

These three possibilities correspond to familiar behaviours of computer programs: succeed, crash, and hang.

Illustration of the three behaviours



- Loops in state 1, moving right:

$$\cdots \Delta \ \underset{1}{\boxed{\Delta}} \ \Delta \cdots$$

- Succeeds after two steps:

$$\cdots \Delta \ \underset{1}{\boxed{\Delta}} \ 1\Delta \cdots$$

- Fails in state 1 after one step:

$$\cdots \Delta \ \underset{1}{\boxed{\Delta}} \ 2\Delta \cdots$$

Turing machine simulators

When we simulate the action of a Turing machine, either by hand, or using a computer program, this is itself a computational process.

Turing asked the question: Can this computational process be performed by a Turing machine?

In other words: *Can a Turing machine simulate other Turing machines?*

Turing constructed a Turing machine that can simulate *any* Turing machine (including itself).

Such a Turing machine is called a **Universal Turing Machine** (UTM).

What a Universal Turing Machine does

A Universal Turing Machine U takes as input a string consisting of

- A quintuple listing $quins(M)$ of the Turing machine M to be simulated.
- A copy of the input i on which we want to simulate the action of M.

U then simulates the action of M running with input i:

- If M halts with output o when run with input i, then U halts with output o when run with input $quins(M), i$.
- If M fails to halt when run with input i, then U fails to halt when run with input $quins(M), i$.

Such machines can be constructed!

Design for a UTM

Input has the form $X\langle workspace\rangle Y\langle listing\rangle Z\langle tape\rangle$, where

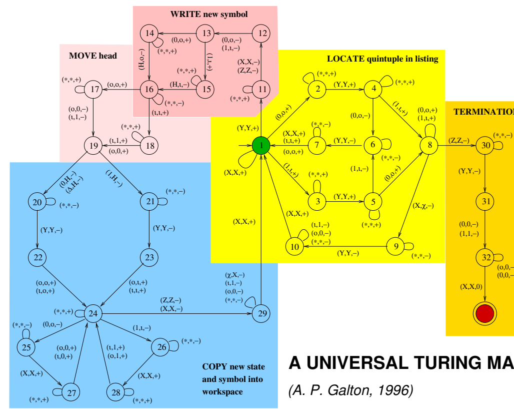
- $\langle workspace\rangle$ contains the start state and the initially scanned symbol.
- $\langle listing\rangle$ contains the quintuples of the simulated machine, separated by 'X'.
- $\langle tape\rangle$ contains the (one-way) tape, with the head-position marked by 'h'.

Simulated machine states are numbered in binary, with halt state 0, start state 1.
All states are represented by strings of the same length.
The tape alphabet of the simulated machine is $\{0,1\}$, where 0 stands for Δ.
The shift is 0 (left) or 1 (right).

Execution Cycle of UTM

1. LOCATE a quintuple (q, a, q', a', d) in $\langle listing\rangle$ such that q and a match the state and symbol in $\langle workspace\rangle$.
2. WRITE a' at the simulated head position in $\langle tape\rangle$ (overwriting 'h').
3. MOVE simulated head to new position on $\langle tape\rangle$ (using d to determine where to go).
4. COPY q' (from the quintuple) and new scanned symbol (from the tape) into $\langle workspace\rangle$.

This cycle is repeated until the LOCATE procedure fails to find a suitable quintuple. If the current state (shown in $\langle workspace\rangle$) is 0 (the halt state) then SUCCEED, else FAIL.

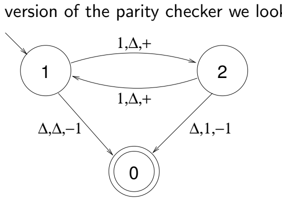


A UNIVERSAL TURING MACHINE

(A. P. Galton, 1996)

Example

We'll illustrate by using the UTM to simulate a slightly modified version of the parity checker we looked at earlier.

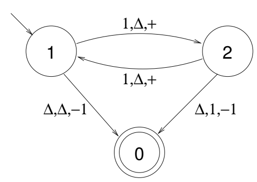


1,	1,	2,	Δ,	1
1,	Δ,	0,	0,	-1
2,	1,	1,	Δ,	1
2,	Δ,	0,	1,	-1

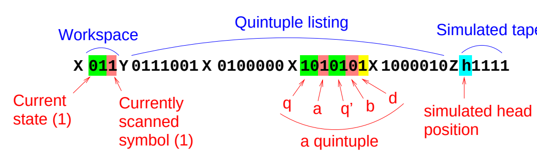
The initial tape for the UTM, corresponding to the parity-checker with input 11111, is

```
X011Y0111001X0100000X1010101X1000010Zh1111
```


Explanation of the input to the UTM



1,	1,	2,	Δ,	1
1,	Δ,	0,	0,	-1
2,	1,	1,	Δ,	1
2,	Δ,	0,	1,	-1



The Halting Problem

Can we know in advance whether or not a Turing machine, when run with a given input, will ever reach the halt state?

Sometimes it is clear that we can (as in our examples above).

In other cases it may be far from obvious.

The **Halting Problem** (HP) is specified as follows: To determine, for an arbitrary Turing machine M and input i, whether or not M will reach the halt state when run with input i.

In particular: Is there a Turing machine H which can solve the Halting Problem for M, i when given input $quins(M), i$?

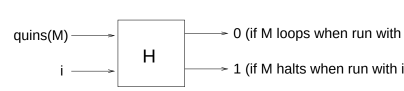
A clever argument by Turing shows that the answer to this is **NO**.

Insolubility of the Halting Problem I

Suppose we have a Turing machine H which someone claims solves HP for arbitary TM/input pairs.

Such a machine, when given input $quins(M), i$,

- halts with output 1 if M would halt when given input i
- halts with output 0 if M would loop when given input i

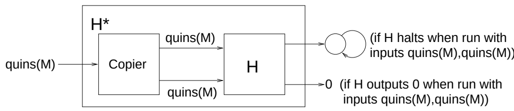


We shall show that it is impossible for a Turing Machine to behave in this way, i.e., the claim is incorrect.

Insolubility of the Halting Problem II

Given H, we can modify it to a machine H^* as follows:

- Instead of taking two inputs $quins(M)$ and i, it takes a single input $quins(M)$, which it copies, so that the two inputs to H are effectively $quins(M)$ and $quins(M)$.
- After that it acts exactly like H until it reaches a transition leading to the halt state.
- If the output at this point is 1, instead of going to the halt state, H^* goes into a loop.



Insolubility of the Halting Problem III: Taking Stock

The story so far:

We started with a machine H which, it is claimed, solves the Halting Problem, i.e., given inputs $quins(M), i$ it delivers output

- 0 if M would loop when given input i
- 1 if M would halt when given input i

Given this machine H, we created a new machine H^* which, when given input $quins(M)$,

- halts with output 0 if H would output 0 when run with inputs $quins(M), quins(M)$
- loops if H would output 1 when run with inputs $quins(M), quins(M)$

Insolubility of the Halting Problem IV

What happens if we run H^* with input $quins(H^*)$?
(i.e., we're giving H^* its own quintuples as input)

- Suppose it halts with output 0. This means that H would output 0 when run with inputs $quins(H^*)$, $quins(H^*)$.
  If H solves HP, this would mean that H^* would loop when run with input $quins(H^*)$; since it doesn't, H does not solve HP.
- Suppose instead it loops. This means that H would output 1 when run with inputs $quins(H^*)$, $quins(H^*)$.
  If H solves HP, this would mean that H^* would halt when run with input $quins(H^*)$; it doesn't, so H does not solve HP.

We have shown that whether H^* halts or loops when run with input $quins(H^*)$, H cannot solve HP.

Since this argument applies to any machine H put forward as a solution to HP, it follows that *the Halting Problem is insoluble*.

The Halting Problem for Computer Programs

A program H to solve HP for computer programs would, when run with inputs P and i (where P is a computer program),

- halt with output 0 if P would loop when run with input i
- halt with output 1 if P would halt when run with input i.

If H is		then H^* is	
	input P; input i; $<$ *main code* $>$ output a		input P; $i \leftarrow P$; $<$ *main code* $>$ if $a = 1$ then $a \leftarrow 1$; output a

Then we run H^* with input H^*. If H solves HP, then

- If H^* halts then it loops
- If H^* loops then it halts.

Since this behaviour is impossible, H cannot solve HP.

The Limits of Computability

There are many computational problems which can be shown to be insoluble by showing that they are *equivalent* to the Halting Problem.

An important example is: Find an algorithm to determine, for an arbitrary set of first-order logical statements, whether or not they are consistent.

We have reached *the limits of computability*.

There are some problems which just can't be solved, at least not by Turing machines or anything equivalent to them.

And there is good reason to believe that *any* form of computation is equivalent to Turing machine computation.

This is called the **Church-Turing Thesis**.

The Church-Turing Thesis

The Church-Turing Thesis (CTT) states that if something that can be computed at all, then there is a Turing machine that can compute it.

There are two kinds of evidence for CTT:

- **Negative evidence.** Despite many attempts, no-one has yet come up with a method of computation which can be shown to be more powerful than Turing machines.
- **Positive evidence.** All the many different attempts to characterise in formal terms exactly what is meant by computation have resulted in definitions that can be shown to be equivalent to Turing machine computation.

Despite this, some researchers believe that non-Turing-equivalent computation should be possible, and have invented the term **hypercomputation** to refer to it.