COMP30026 Models of Computation

Recognisable and Decibale Languages

Bach Le / Anna Kalenkova

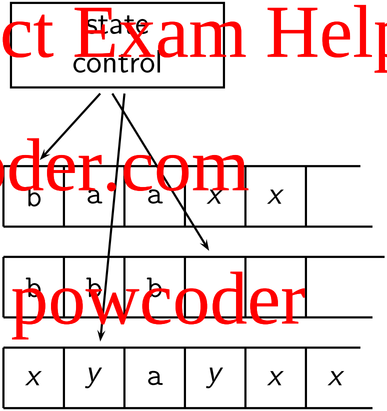
Lecture Week 11. Part 1

Semester 2, 2021

Turing Machines: Simulation

Multitape Machines

A multitape Turing machine has k tapes. It takes its input on tape 1, other tapes are blank.



The transition function now has type

$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

It specifies how the k tape heads behave when the machine is in state q_i, reading $a_1, \dots a_k$:

$$\delta(q_i, a_1, \dots, a_k) = (q_j, (b_1, \dots, b_k), (d_1, \dots, d_k))$$

Simulating a Multitape Machine

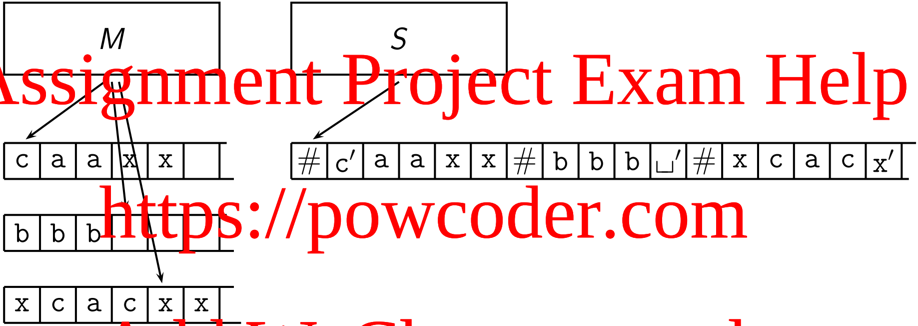
Theorem: A language is Turing recognisable iff some multitape Turing machine recognises it.

Proof sketch: We show how to simulate a multitape machine M by a standard Turing machine S.

Suppose the multitape machine's tape alphabet is Γ.

The standard machine has tape alphabet $\{\#\} \cup \Gamma \cup \Gamma'$ where $\#$ is a separator, not in $\Gamma \cup \Gamma'$, and there is some one-to-one correspondence between elements in Γ and elements in Γ'.

Simulating a Multitape Machine



S reorganises input $x_1 x_2 \cdots x_n$ into $\#x_1' x_2 \cdots x_n \underbrace{\#\sqcup'\# \cdots \#\sqcup'\#}_{k-1 \text{ times}}$

Note how elements of Γ' represent "marked" elements from Γ.

Simulating a Multitape Machine

Simulating an M move, S scans its tape to determine the marked symbols. On a second scan it updates the tape according to M's transition function.

If a "virtual head" of M moves to a $\#$, S shifts that symbol, and every symbol after it, one cell to the right. In the vacant cell it writes $\sqcup$.

Nondeterministic Turing Machines

A nondeterministic Turing machine has a transition function of type

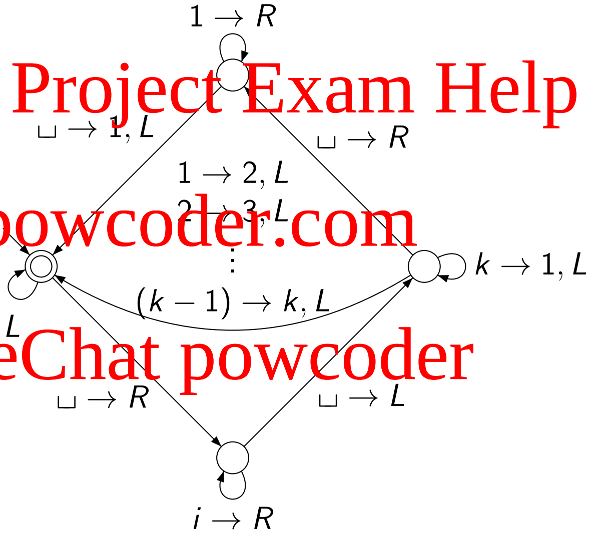
$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

If some computation branch leads to 'accept' then the machine accepts its input.

This is the same type of nondeterminism that an NFA possesses.

Simulating a Nondeterministic Turing Machine

First, a deterministic machine to generate $\{1, \ldots, k\}^*$, in order of increasing length.



Try running it for $k = 3$.

Simulating a Nondeterministic Turing Machine

Theorem: A language is Turing recognisable iff some nondeterministic Turing machine recognises it.

Proof sketch: We need to show that every nondeterministic Turing machine N can be simulated by a deterministic Turing machine D.

We show how it can be simulated by a 3-tape machine.

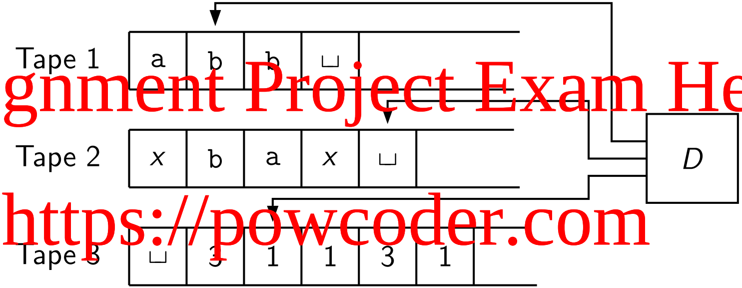
Let k be the largest number of choices, according to N's transition function, for any state/symbol combination.

Tape 1 contains the input.

Tape 3 holds longer and longer sequences from $\{1, \ldots, k\}^*$.

Tape 2 is used to simulate N's behaviour for each fixed sequence of choices given by tape 3.

Simulating a Nondeterministic Turing Machine



1. Initially tape 1 contains input w. The other two tapes are empty.
2. Overwrite tape 2 by w.
3. Use tape 2 to simulate N. Tape 3 dictates how N should make its choices. If N says accept, accept. If the "choice list" on tape 3 gets exhausted, go to step 4.
4. Generate the next choice list on tape 3. Go to step 2.

Enumerators

The Turing machine we built to generate all strings in $\{1, \ldots, k\}^*$ is an example of an enumerator.

We could imagine it being attached to a printer, and it would print all the strings in $\{1, \ldots, k\}^*$, one after the other, never terminating.

For an enumerator to enumerate a language L, for each $w \in L$, it must eventually print w.

The reason why we also call Turing recognisable languages recursively enumerable is the following theorem.

Enumerators

Thm: L is Turing recognisable iff some enumerator enumerates L.

Proof: Let E enumerate L. Then we can build a Turing machine recognising L as follows:

1. Let w be the input.
2. Simulate E. For each string s output by E, if $s = w$, accept.

Enumerators

Thm: L is Turing recognisable iff some enumerator enumerates L.

Proof: Let E enumerate L. Then we can build a Turing machine recognising L as follows:

1. Let w be the input.
2. Simulate E. For each string s output by E, if $s = w$, accept.

Conversely, let M recognise L. Then we can build an enumerator E by elaborating the enumerator from a few slides back: We can enumerate Σ^*, producing $s_1, s_2, \ldots$

Enumerators

Thm: L is Turing recognisable iff some enumerator enumerates L.

Proof: Let E enumerate L. Then we can build a Turing machine recognising L as follows:

1. Let w be the input.
2. Simulate E. For each string s output by E, if $s = w$, accept.

Conversely, let M recognise L. Then we can build an enumerator E by elaborating the enumerator from a few slides back: We can enumerate Σ^*, producing $s_1, s_2, \ldots$ Here is what E does:

1. Let $i = 1$.
2. Simulate M for i steps on each of $s_1, \ldots s_i$.
3. For each accepting computation, print that s.
4. Increment i and go to step 2.

Termination

Partial Functions

We have silently assumed that the domain of a function is known, so that $f : X \to Y$ means that $f(x)$ is defined for each $x \in X$.

In computer science, however, it is often more appropriate to deal with functions that are partial.

We write $f : X \hookrightarrow Y$ to say that f has a domain which is a subset of X, but $f(x)$ may be undefined for some $x \in X$.

Note that a total function $f : X \to Y$ is by definition also partial: $f : X \hookrightarrow Y$.

Partial Functions: Example 1

The function f defined by

$$f(n) = \begin{cases} 42 & \text{if } n = 0 \\ f(n-2) & \text{if } n \neq 0 \end{cases}$$

is a partial function $f : \mathbb{Z} \hookrightarrow \mathbb{Z}$.

In a natural interpretation, it is undefined if n is odd and/or negative. Its range is $\{42\}$.

In this case, it is not too hard to determine the set of values for which f is defined. So we could also choose to say that f is a total function $X \to \mathbb{Z}$, where $X = \{n \in \mathbb{Z} \mid n \geq 0 \wedge n \text{ is even}\}$.

However, it is not always easy, or even possible, to determine a function's domain.

Partial Functions: Example 2

The function c defined by

$$c(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ c(n/2) & \text{if } n \text{ is even and } n > 1 \\ c(3n+1) & \text{if } n \text{ is odd and } n > 1 \end{cases}$$

is a partial function $c : \mathbb{N} \hookrightarrow \mathbb{N}$ with range $\{1\}$.

Will the evaluation of $c(n)$ terminate for all n?

Partial Functions: Example 2

The function c defined by

$$c(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ c(n/2) & \text{if } n \text{ is even and } n > 1 \\ c(3n+1) & \text{if } n \text{ is odd and } n > 1 \end{cases}$$

is a partial function $c : \mathbb{N} \hookrightarrow \mathbb{N}$ with range $\{1\}$.

Will the evaluation of $c(n)$ terminate for all n?

It is not known whether c is total.

This is the so-called $3n+1$ problem, or Collatz's problem.

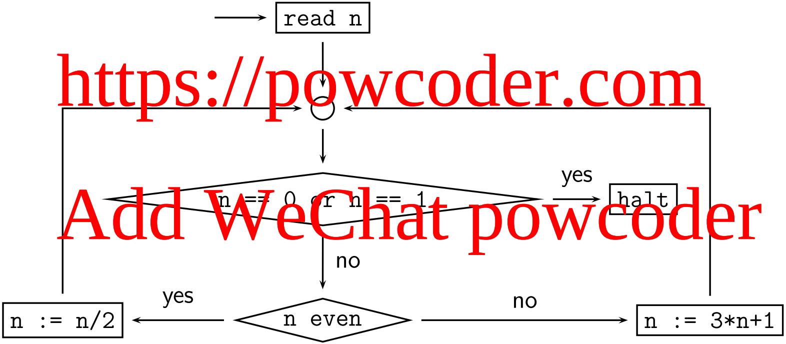
Program Termination

Here is a Haskell program that produces the list of n-values generated in successive recursive calls:


```
c :: Integer -> [Integer]
c 0 = [1]
c 1 = [1]
c n = n : c (if even n then n `div` 2 else 3*n+1)
```

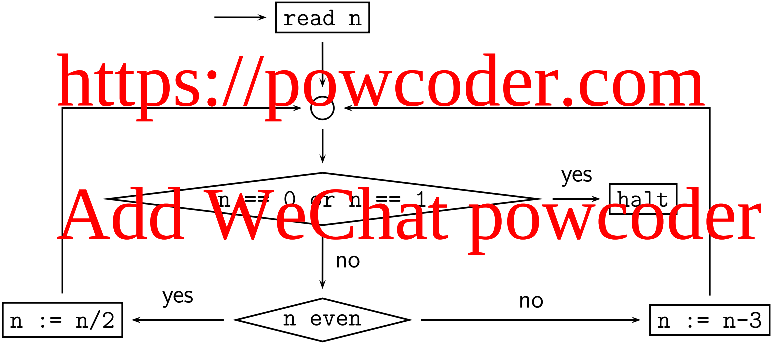

Colatz's sequence for 27: 27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1.

Here it is as a flowchart program:



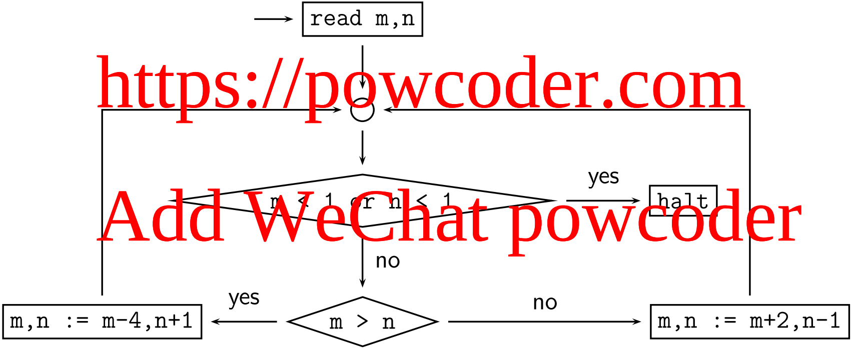
Quiz 1: Does It Terminate?

Here is a variant. Does this halt for all positive input?



Quiz 2: Does It Terminate?

And how about this? Does it halt for all input?



Proving Termination

Suppose that, for each loop in a program, we can find some "measure" (a function of the program variables) such that

1. the measure is a natural number, and
2. the measure gets smaller with each loop iteration.

Then the program must terminate for all input, because a natural number cannot be made smaller indefinitely.

The Termination Question

Termination of algorithms is a tricky problem (and the general problem is undecidable).

For example, we suggested earlier algorithms for translating propositional formulas to, say, DNF, including rules like

$$
\begin{array}{rcl}
\neg(\alpha \wedge \beta) & \rightsquigarrow & \neg\alpha \vee \neg\beta \\
\neg(\alpha \vee \beta) & \rightsquigarrow & \neg\alpha \wedge \neg\beta \\
\neg\neg\alpha & \rightsquigarrow & \alpha \\
\alpha \wedge (\beta \vee \gamma) & \rightsquigarrow & (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \\
(\beta \vee \gamma) \wedge \alpha & \rightsquigarrow & (\beta \wedge \alpha) \vee (\gamma \wedge \alpha)
\end{array}
$$

Note that some of the rules decrease the size of a term, while others increase it (and some duplicate certain subterms).

So why should this process terminate?

Quiz 3: A Marble Game

You have a bag of red and white marbles, plus a box of red marbles.

Repeat this process:

- If the bag contains exactly one marble, halt;
  otherwise take two random marbles from the bag.

- If they are of different colours, put them back.
- If they are of the same colour, discard both,
  but put one red marble (from the box) in the bag.

Does this terminate?

Well-Founded Orderings

The binary relation $\prec$ over some set X is well-founded iff there is no infinite sequence of X-elements $x_1, x_2, x_3, \ldots$ such that

$$x_1 \succ x_2 \succ x_3 \succ \cdots$$

We say that $(X, \prec)$ is a well-founded structure.

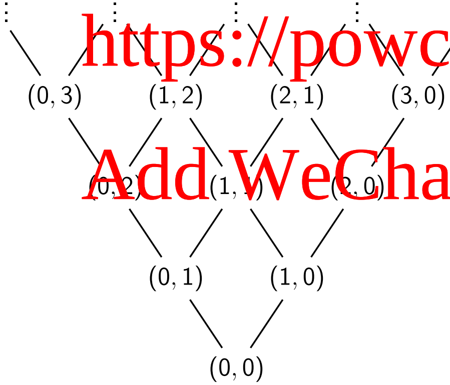
For example, $(\mathbb{N}, <)$ is a well-founded structure.

Given a finite number of well-founded structures $(X_1, \prec_1), \ldots, (X_n, \prec_n)$, we can obtain well-founded orderings of $X = X_1 \times \cdots \times X_n$ in a number of different ways.

Ordering Pairs: Component-Wise Ordering

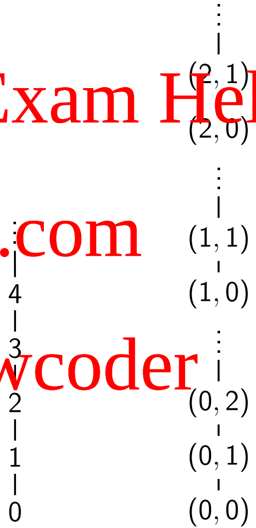
$$(x_1, x_2) \preceq (y_1, y_2) \text{ iff } x_1 \le y_1 \wedge x_2 \le y_2$$

$$p \prec q \text{ iff } p \preceq q \wedge p \neq q$$



A Hasse diagram for component-wise ordering of $\mathbb{N} \times \mathbb{N}$.

Values increase as you travel up along edges.



On the left: $\mathbb{N}$ with the usual ordering.

On the right: $(\mathbb{N} \times \mathbb{N}, \preceq)$, where $\preceq$ is lexicographic ordering: $(m, n) \preceq (m', n')$ iff $m \leq m' \wedge (m = m' \Rightarrow n \leq n')$.

Define again $p \prec q$ iff $p \preceq q \wedge p \neq q$.

Well-Founded Orderings on Tuples

Theorem: If $\prec$ is well-founded then so is its component-wise extension to tuples.

Theorem: If $\prec$ is well-founded then so is its lexicographic extension to tuples.

Component-Wise Ordering of Tuples

Ordering $X = X_1 \times \cdots \times X_n$ component-wise:

$$(x_1, \ldots, x_n) \preceq (y_1, \ldots, y_n) \text{ iff } \bigwedge_{i=1}^{n} x_i \preceq_i y_i$$

If each $(X_i, \prec_i)$ is well-founded, then so is $(X, \prec)$.

Example using component-wise ordering:

$$(2,2,2) \succ (2,2,1) \succ (2,1,1) \succ (2,0,1) \succ (2,0,0) \succ (0,0,0)$$

Lexicographic Ordering of Tuples

Ordering $X = X_1 \times \cdots \times X_n$ lexicographically:

$$(x_1, \ldots, x_n) \prec (y_1, \ldots, y_n) \text{ iff } \bigvee_{i=1}^{n} \left(x_i \prec_i y_i \wedge \bigwedge_{j=1}^{i-1} x_j = y_j \right)$$

If each $(X_i, \prec_i)$ is well-founded, then so is $(X, \prec)$.

Example using lexicographic ordering:

$$(2,2,2) \succ (2,1,42) \succ (1,3,1000) \succ (1,3,999) \succ (1,3,0) \succ (0,0,15)$$

Well-Founded Induction

There is an induction principle to go with well-founded relations.

Given a well-founded structure $(X, \prec)$ we can prove a statement "for all $x \in X$, $S(x)$" as follows:

We proceed in one step:

- Assume that $S(x')$ holds for all $x' \prec x$, and use that to establish $S(x)$.

Example 1: The Dutch Flag

We have 12 pebbles laid out in a row. Consider three rewrite rules

$\circ \quad \textcolor{red}{\bullet} \quad \rightsquigarrow \quad \textcolor{red}{\bullet} \quad \circ$ (for a white left of a red, swap)

$\textcolor{blue}{\bullet} \quad \textcolor{red}{\bullet} \quad \rightsquigarrow \quad \textcolor{red}{\bullet} \quad \textcolor{blue}{\bullet}$ (for a blue left of a red, swap)

$\textcolor{blue}{\bullet} \quad \circ \quad \rightsquigarrow \quad \circ \quad \textcolor{blue}{\bullet}$ (for a blue left of a white, swap)

$\textcolor{red}{\bullet} \quad \circ \quad \textcolor{blue}{\bullet} \quad \textcolor{blue}{\bullet} \quad \textcolor{red}{\bullet} \quad \textcolor{blue}{\bullet} \quad \circ \quad \circ \quad \textcolor{red}{\bullet} \quad \textcolor{blue}{\bullet} \quad \textcolor{red}{\bullet} \quad \circ$

To see that rewriting terminates, use $\textcolor{red}{\bullet} < \circ < \textcolor{blue}{\bullet}$ together with lexicographic ordering on 12-tuples.

Example 2: Ackermann's Function

The following is a definition of Ackermann's function:

$$
\begin{array}{lcl}
ack(0, y) & = & y + 1 \\
ack(x + 1, 0) & = & ack(x, 1) \\
ack(x + 1, y + 1) & = & ack(x, ack(x + 1, y))
\end{array}
$$

It grows incredibly fast.

For example, $ack(3, 3) = 61$ and $ack(4, 4) = 2^{2^{2^{2^{16}}}} - 3$

However, lexicographic well-founded induction allows us to conclude that the function is total—as a Haskell program it will terminate for all input (possibly after a very long time).

Example 2: Ackermann's Function

In each recursive call, the argument (x, y) decreases strictly

$$
\begin{aligned}
ack(x+1, 0) &= \overbrace{ack(x, 1)}^{x \text{ smaller}} \\
ack(x+1, y+1) &= \underbrace{ack(x, \underbrace{ack(x+1, y)}_{x \text{ same, } y \text{ smaller}})}_{x \text{ smaller}}
\end{aligned}
$$