ITCS Notes

Christopher Dalziel

November-December 2023

Contents

0 Introduction

Hello! These are my notes for ITCS, which I hope will be useful to you if you're reading this.

These notes are adapted from the excellent slides provided by Dr. Liam O'Connor. Not all proofs are contained here, so if there's something that seems to be missing do check the slides!

I've added a definitions section, which mostly contains definitions which weren't in the course that are assumed knowledge from courses like DMP. Those definitions which were in the course but I've moved from the main body of the notes are the ones which I either felt stood well on their own (the note on Turing Machines for example) or were not critical to understanding a general section of notes (e.g. parse trees)

As with my CSEC notes there are highlighted sections of text which act as links to different sections of the document - something I find particularly useful for this course!

I do hope these notes are useful to you, I've had good reviews on the CSEC ones - regardless, good luck with the exam!

1 Finite Automata

A finite automaton takes a string as input and replies "yes" or "no". If an automaton A replies "yes" on a string S we say A "accepts" S.

A string S is a possibly empty sequence of symbols from an alphabet Σ.

The language of a finite automaton A, written $\mathcal{L}(A)$ is the set of strings for which A accepts.

Deterministic Finite Automata

A deterministic finite automaton (DFA) is a quintuple $(Q, \Sigma, q_0, \delta, F)$ where:

- Q is a finite set of states
- Σ is an alphabet
- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $F \subseteq Q$ is the set of final states

A DFA accepts a string $w \in \Sigma^*$ iff $\delta^*(q_0, w) \in F$, where δ^* is δ applied successively for each symbol in w. The language of a DFA A is the set of all strings accepted by A, $\mathcal{L}(A) \subseteq \Sigma^*$ is the set of all strings accepted by A.

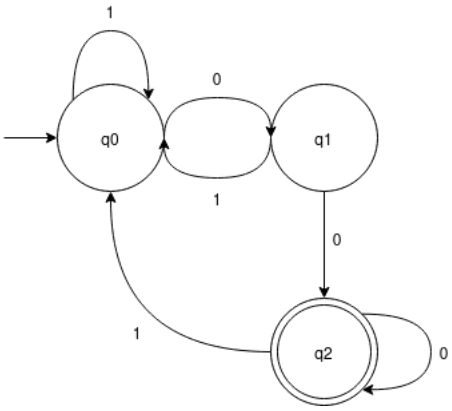

Figure 1: Example of a DFA

The following pseudocode may help explain δ^* if the above is confusing:

```
0 define d*(q_0: State, w: String) -> Boolean {
```

```
    state = q_0;

    for char in w {
        state = d(state, char);
    }

    return state in F;
}
```

In a DFA, the transition function is a total function which gives exactly one next state for each input symbol. This means it is deterministic.

If δ were partial we could express a total equivalent using less information - any partial DFA could be a total one.

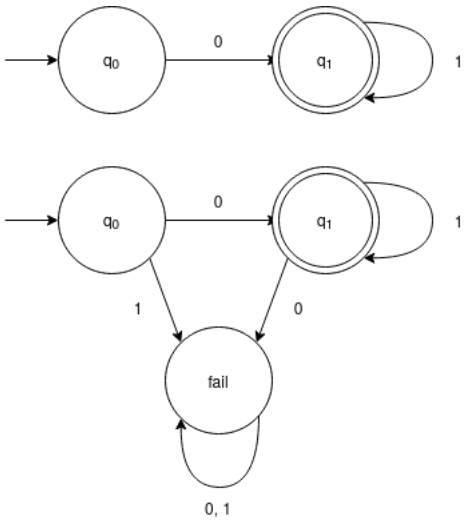

Figure 2: Partial DFA (top) and an equivalent total DFA (bottom)

Non-Deterministic Finite Automata

Non-determinism would mean that δ can return more than one successor state, it instead returns a set of possible states - no states is an empty set.

Non-deterministic finite automata (NFA) are angelically non-deterministic. An NFA is a quintuple $(Q, \Sigma, q_0, \delta, F)$ where:

- Q is a finite set of states
- Σ is an alphabet

- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function
- $F \subseteq Q$ is the set of final states

The only difference between the definition of a DFA and that of an NFA is that in an NFA δ returns an element from the power set of Q, $\mathcal{P}(Q)$.

Adding non-determinism doesn't change "expressivity". Given an NFA A there is an equivalent DFA D such that $\mathcal{L}(D) = \mathcal{L}(A)$ and vice versa.

Proof: NFA $\to$ DFA: A DFA is already an NFA, one where the transition function always returns a singleton set.

DFA $\to$ NFA: This is harder, and we use the subset construction.

Subset Construction

For an NFA A, the corresponding DFA D tracks the set of states that A could possibly be in, giving the string read so far. So each state of D is a set of states from A.

Given an NFA $(Q_A, \Sigma, q_0, \delta_A, F_A)$, we can construct a DFA $(\mathcal{P}(Q_A), \Sigma, \{q_0\}, \delta_D, F_D)$ where:

$$\delta_D(S, a) = \bigcup_{q \in S} \delta_A(q, a) \text{ for each } S \subseteq Q$$

and:

$$F_D = \{S \subseteq Q \mid S \cap F_A \neq \emptyset\}$$

A subset construction involves making a DFA where the states are all possible combinations of states that the original NFA can be in.

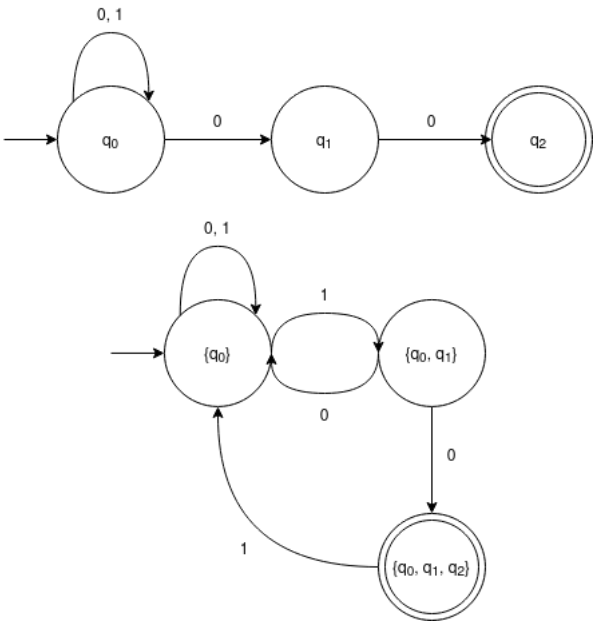

Figure 3: An example of an equivalent DFA and NFA

ϵ-NFA

If we allow non-deterministic state changes that don't consume any input symbols. We label silent moves using ϵ - meaning the empty string.

We define the ϵ-closure $E(q)$ of a state q as the set of all states reachable from q by silent moves. That is, $E(q)$ is the least set satisfying:

- $q \in E(q)$
- For any $s \in E(q)$ we also have $\delta(s, \epsilon) \subseteq E(q)$

We also extend this to sets where $E(s) = \bigcup_{q \in s} E(q)$

These too are equal in expressive power to NFA and DFA.

2 Regular Languages

Any language which can be accepted by a finite automaton is called a regular language.

Regular languages are also those recognised by Regular Expressions.

Regular Expressions

Regular expressions exactly characterise the regular languages, just as finite automata do.

The following table shows the syntax and semantics (the meaning of the syntax) of a regular expression.

begintable[H]

Syntax	Semantics	
a	$[\![a]\!] = \{a\}$	$(a \in \Sigma)$
$\emptyset$	$[\![\emptyset]\!] = \emptyset$	
ϵ	$[\![\epsilon]\!] = \{\epsilon\}$	
$R_1 \cup R_2$	$[\![R_1 \cup R_2]\!] = [\![R_1]\!] \cup [\![R_2]\!]$	
$R_1 \circ R_2$	$[\![R_1 \circ R_2]\!] = [\![R_1]\!][\![R_2]\!]$	
R^*	$[\![R^*]\!] = [\![R]\!]^*$	

Examples: At least one 0:

$$(0 \cup 1)^* 0 (0 \cup 1)^*$$

At least one 1 and at least one 0:

$$((0 \cup 1)^* 01 (0 \cup 1)^*) \cup ((0 \cup 1)^* 10 (0 \cup 1)^*)$$

Generalised NFA

A generalised NFA is an NFA where transitions have regular expressions instead of symbols, there is only one unique final state, and the transition relation is full, except that the initial state has no ingoing transitions and the final state no outgoing ones.

The process to do this is as follows:

1. Add a new start state, connected via ϵ transitions to the old one
2. Add a new final state, connected via ϵ transitions to all old final states
3. If two states q_0 and q_1 have two transitions between them $q_0 \xrightarrow{a} q_1$ and $q_0 \xrightarrow{b} q_1$, replace them with $q_0 \xrightarrow{a \cup b} q_1$

4. Introduce $\emptyset$-labelled transitions where needed to make the transition relation full

By eliminating each of the inner states of a GNFA one by one, you can get a GNFA with a start and final state connected by a single transition between them. The label on that transition is the regular expression that characterises the original GNFA.

3 Beyond Regular Languages

Most languages are not regular, programming languages for example are mostly non-regular.

Determining matching parentheses is non-regular.

$$L_{()} = \{(^n)^n \mid n \in \mathbb{N}\}$$

Doing this would require counting the number of opening brackets as a regular string - an unbounded natural number, requiring unbounded memory and resulting in non-finite states.

This breaks the definition of a Finite Automaton - meaning it cannot be represented by one, and so mustn't be a regular language.

To determine if a language is regular or not we have two methods: the Pumping Lemma and the Myhill-Nerode Theorem.

The Pumping Lemma

If a DFA with k states accepts a word of length greater than k, we know that the DFA must have visited one of its states more than once - the DFA must have a loop.

Therefore if we go through the loop any number of times, the DFA will accept those words too. We call this "pumping".

If $L \subseteq \Sigma^*$ is regular then there's a pumping length $p \in \mathbb{N}$ such that for any $w \in L$ where $|w| \geq p$, we may split w into three pieces $w = xyz$ satisfying three conditions:

- $xy^iz \ \forall i \in \mathbb{N}$
- $|y| > 0$, and
- $|xy| \leq p$

Because we know that this is true for all regular languages, the contrapositive statement - that if we can't pump a language it isn't regular - must also be true.

Importantly, the opposite is not true. Meaning that not all pumpable languages are regular.

The Myhill-Nerode Theorem

Let $L \subseteq \Sigma^*$ and $x, y \in \Sigma^*$, if there exists a suffix string z such that $xz \in L$ but $yz \notin L$ or vice versa, then x and y are distinguishable by L.

If x and y are not distinguishable by L, then we say $x \equiv_L y$ - this is an equivalence relation.

A language L is regular iff the number of equivalence classes using this relation $\equiv_L$ is finite. If there were not a finite number of classes there would need to be an infinite number of states in the DFA.

To show a language is non-regular then, we can show that it has infinite equivalence classes - that is, we find an infinite sequence $u_0 u_1 u_2 \ldots$ of strings such that for any i, j (where $i \neq j$) there is a string w_{ij} such that $u_i w_{ij} \in L$ but $u_j w_{ij} \notin L$ or vice-versa.

For the matching brackets example we can choose $u_i = (^i$ and $w_{ij} =)^i$.

4 Context-free Languages

By adding recursion to regular expressions we can begin to recognise some non-regular languages.

All regular languages are context free.

A language is context-free iff it is recognised by a context-free grammar.

Context-free Grammars

A Context-free Grammar (CFG) is a 4-tuple (N, Σ, P, S) where:

- N is a finite set of variables or non-terminals
- Σ is a finite set of terminals
- $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of rules or productions.
  - Typically productions are written $A \to aBc$
  - Productions with common heads can be combined, $A \to a$ and $A \to Aa$ can be combined into $A \to a \mid Aa$
- $S \in N$ is the starting variable

We use α, β, γ to refer to sequences of terminals.

We make a derivation step $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ whenever $(A \to \gamma) \in P$; The language of a CFG G is:

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w^*\}$$

Where $\Rightarrow_G^*$ is the reflexive, transitive, closure of $\Rightarrow_G$.

Context-free grammars are ambiguous. They are closed under union, concatenation, and kleene star.

Eliminating Ambiguity

We want to eliminate ambiguity in context-free grammars while still accepting all the same strings.

This can be done for our language of regular expressions:

- First defining atomic expressions: $A \to (S) \mid \emptyset \mid \epsilon \mid a \mid b$
- Then ones which use Klene Star: $K \to A \mid A^*$

- Then ones which may use left-associative composition: $C \to K \mid C \circ K$
- Finally expressions which use unions: $S \to C \mid S \cup C$

The order of operations here is therefore bottom to top; unions come before compositions come before kleene star etc.

Push-down Automata

Push-down automata are to context-free grammars what finite automata are to regular expressions.
They are implementationally identical to ϵ-NFA with one addition, a stack.
The recursive element of CFGs is implemented using a standard last-in-first-out stack.

Transitions in a push-down automata take the form $x, y \to z$, which is read as "consume the input x, popping y off the stack, and push z onto the stack". We can allow actions that don't consume, pop, or push by setting x, y, or z to ϵ.

PDA operate on an input alphabet Σ, but their stack has its own "stack alphabet": $\Gamma = \Sigma \cup \{\bullet\}$. This bullet ($\bullet$) character is used for bookkeeping, usually as a way of determining when we've reached the bottom of the stack.

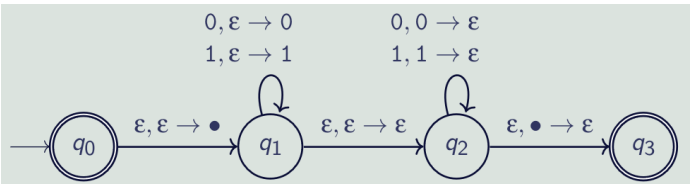

Figure 4: A PDA for recognising even length palindromes on $\Sigma = \{0, 1\}$

Formally, a push-down automata is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q, Σ, and Γ are all finite sets.

Γ is the stack alphabet and δ now may take a stack symbol as input or return one as output.

$$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$$

A string w is accepted by a PDA if it ends in a final state, i.e. $\delta^*(q_0, w, \epsilon)$ gives a state q and a stack γ such that $q \in F$.

Any language that is recognised by a push-down automata is context free.

CFG to PDA

The upper loop on q_1 is added for every terminal a in the CFG. The lower loop on q_1 is shorthand for a looping sequence of states added for each production $A \to w$ that builds up w on the stack one symbol at a time.

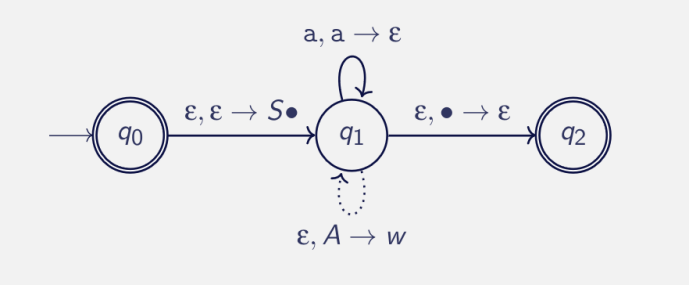

Figure 5: The general form of converting a CFG to a PDA

PDA to CFG

Firstly we ensure that the PDA has only one accept state, empties its stack before terminating, and has only transitions that either push or pop a symbol (but not transitions that do both or neither).

Given such a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ we provide a CFG (V, Σ, R, S) with V containing a non-terminal A_{pq} for every pair of states $(p, q) \in Q \times Q$.

The non-terminal A_{pq} generates all strings that go from p with an empty stack to q with an empty stack. Then S is just $A_{q_0 q_{accept}}$.

R consists of:

- $A_{pq} \to aA_{rs}b$ if $p \xrightarrow{a,\epsilon \to t} r$ and $s \xrightarrow{b,t,\to \epsilon} q$ (for intermediate states r, s and stack symbol t
- $A_{pq} \to A_{pr}A_{rq}$ for all intermediate states r
- $A_{pp} \to \epsilon$

Pumping for CFLs

Supposing a CFG has n non-terminals, and the parse tree has a height $k > n$, we know that some non-terminal state v must appear as its own descendant in the tree.

Unlike when pumping for regular languages, we can pump up or down the parse tree when pumping for context-free languages.

We can pump down by cutting the tree at a higher occurrence of V and replacing it with the sub-tree at the lower occurrence of V.

We can pump up by cutting away the lower occurence of V instead, and replacing it with a fresh copy of the higher occurence.

If a language L is context-free, then there exists a pumping length $p \in \mathbb{N}$ such that if $w \in L$ with $|w| \geq p$ then w may be split into five pieces $w = uvxyz$ such that:

1. $uv^ixy^iz \in L \; \forall i \in \mathbb{N}$
2. $|vy| > 0$ and
3. $|vxy| \leq p$

The general rule of thumb should be to pick a string w that allows as few cases for partitions of $w = uvxyz$ as possible

Chomsky Grammars

Context-free grammars are a special case of Chomsky Grammars. Chomsky grammars are similar to CFG, except that the left-hand side of a production may be any string that includes at least one non-terminal.

$$S \to abc \mid aAbc$$

$$Ab \to bA$$

$$\vdots$$

Such a grammar is called "context-sensitive".

The Chomsky Hierarchy

A grammar $G = (N, \Sigma, P, S)$ is of type:

0. (or computably enumarable/turing recognisable) in the general case.
1. (or context-sensitive) if $|\alpha| \leq |\beta|$ for all productions $\alpha \to \beta$, except we also allow $S \to \epsilon$ if S does not occur on the right hand side of any rule.
2. (or context-free) if all productions are of the form $A \to \alpha$.
3. (or right-linear/regular languages) if all productions are of the form $A \to w$ or $A \to wB$ where $w \in \Sigma$ and $B \in N$.

5 Algorithms for Languages

Emptiness for Regular Languages

Can we write a program to determine if a given regular language is empty?

Given a finite-automaton this is an instance of graph reach-ability, so we can use a depth-first search.

Emptiness for Context-free Languages

Can we write a program to determine if a given context-free language is empty?

Given a CFG for our language, we can perform the following process:

1. Mark the terminals and ϵ as generating
2. Mark all non-terminals which have a production with only generating symbols in their right hand side as generating
3. Repeat until nothing new is marked
4. Check if S is marked as generating or not

Equivalence of DFA

Is it possible to write a program to determine if two discrete finite automata are equivalent?

Given two DFA for L_1 and L_2, we can use our standard constructions to produce a DFA of the symmetric set difference:

$$(L_1 \cap \overline{L}_2) \cup (L_2 \cap \overline{L}_1)$$

6 Register Machines

A register machine, or RM, consists of:

- a fixed number m of registers $R_0 \dots R_{m-1}$, which each hold a natural number
- a fixed program P which is a sequence of n instructions $I_0 \dots I_{n-1}$

Each instruction is one of the following:

- `INC(i)`: which increments the register R_i by one
- `DECJZ(i, j)`: which decrements register R_i unless $R_i = 0$ in which case it jumps to instruction I_j

With RMs we can compute anything that another computer can compute.

Their major flaw is that at present they're tedious and cumbersome to work with. We can fix this with "macros", which make it more similar to a real world assembly language.

This includes new instructions, which can make simple tasks like addition simple again, and including symbolic labels for jumps.

Our macros are allowed to access special negative-indexed registers which we can guarantee won't be used by normal programs - this is entirely for convenience and does not give us any more exclusivity.

Example Macros for RMs

Goto I_j using R_{-1} as temp (`GOTO(j)`):

```
0          DECJZ -1, j
```

Clear R_i (`CLEAR(i)`):

```
0          DECJZ i, 2
1          GOTO 0
```

Copy R_i to R_j using R_{-2} as temp (`COPY(i, j)`:

```
0          CLEAR R_j
1  loop_1: DECJZ i, loop_2
2          INC R_j
3          INC R_{-2}
4          GOTO loop_1
5  loop_2: DECJZ R_{-2}, end
```

```
6           INC R_i
7           GOTO loop_2
8   end:
```

How many registers?

So far we've made the assumption that we have as many RM registers as we need, but how many do we actually need?

We can use a pairing function and a corresponding unpairing function to pack every value we need into a single register - meaning we only need as many registers as the pairing function requires and a single user register. This turns out to give us that two registers is enough.

Universality

A model of computation is universal if it can simulate any other model of computation. To simulate a machine we need a machine which given some encoding of a machine M (written $\ulcorner M \urcorner$) as input will produce the correct result of running M as output.

To start proving a model of computation is universal it is generally useful to prove that it can simulate itself.

For a given register machines M, making an encoding involves using a pairing function to pack our instructions and registers into a single value $\ulcorner M \urcorner$.

$$\begin{aligned} \ulcorner INC(i) \urcorner &= \langle 0, i \rangle \\ \ulcorner DECJZ(i,j) \urcorner &= \langle 1, i, j \rangle \\ \ulcorner P \urcorner &= \langle \ulcorner I_0 \urcorner, \ldots, \ulcorner I_{n-1} \urcorner \rangle \\ \ulcorner R \urcorner &= \langle R_0, \ldots, R_{m-1} \rangle \\ \ulcorner M \urcorner &= \langle \ulcorner P \urcorner, \ulcorner R \urcorner \rangle \end{aligned} \tag{1}$$

The Church-Turing Thesis

The Church-Turing thesis states that any problem is computable by any model of computation iff it is computable by a Turing Machine.

For our purposes this matters for RMs, TMs, and λ-calculus.

Other examples are combinator calculus, general recursive functions, pointer machines, counter machines, cellular automata, queue automata, enzyme-based DNA computers, Minecraft, Magic the Gathering (highly recomend this paper), and others.

7 Decidability

Problems with no Algorithm

While the above problems are all computable, this is not always the case. The questions asked, i.e. "can we determine x?", are not always something we can answer using a program - if this is the case then that problem is considered undecidable.

Many such problems exist for Context-free Languages:

- Are two CFG equivalent?
- Is a given CFG ambiguous?
- Is there a way to make a given CFG unambiguous?
- Is the intersection of two CFLs empty?
- Does a CFG generate all strings Σ^*?
- ...

The Halting Problem

Given a register machine encoding can we write a program to determine if the simulated machine will halt or not?

If we suppose H is such a register machine, which takes a machine encoding $\ulcorner M \urcorner$ in R_0 and halts with 1 if M halts and halts with 0 if M doesn't halt.

We can construct a new machine $L = (P_L, R_0, \dots)$ which, given a program $\ulcorner P \urcorner$ runs H on the program with itself as input, the machine $(P, \ulcorner P \urcorner)$ and loops if it halts.

If we run L on P_L itself we get a problem. If L halts on $\ulcorner P_L \urcorner$ that means H says $(P_L, \ulcorner P_L \urcorner)$. If L loops on $\ulcorner P_L \urcorner$ that means that H says $(P_L, \ulcorner P_L \urcorner)$ halts.

This is a contradiction!

The halting problem proves that there are some programs which cannot be decided by register machines. What about other machines?

8 Undecidability

The existence of the halting problem means we know there can be problems which are not decidable. We can use a counting argument to point out that there are uncountably many languages and RMs are enumerable, so there are languages which are not decided (or even recognised) by any RM.

There are many undecidable problems - but how do we show that a problem is undecidable?

Reductions

A reduction is a transformation from one problem to another. To prove that a problem P_2 is hard, show that there is an easy reduction from a known hard problem P_1 to P_2.

To prove that a problem P_2 is undecidable, show that there is a computable reduction from a known undecidable P_1 to P_2. The direction here matters - it tells us nothing to know that there's an easy way to make an easy problem difficult!

If it is a well known fact that Hyunwoo cannot lift a car, we can prove that Hyunwoo cannot lift a loaded truck by making the following reduction: If we suppose he could lift the loaded truck, then we could have him lift the car by putting it in the loaded truck - but we know he cannot lift a car.

Mapping Reductions

A mapping reduction (or a many-to-one reduction) from P_1 to P_2 is a Turing transducer f such that $d \in Q_1$ iff $f(d) \in Q_2$.

If A is mapping reducible to B, and A is undecidable, then B is undecidable.

Turing Reductions

A Turing reduction from P_1 to P_2 is an RM/TM equipped with an oracle for P_2 which solves P_1.

Decidability results carry across during Turing reductions as with mapping reductions, but mapping reductions make finer distinctions of computing power.

Rice's Theorem

All nontrivial semantic properties are undecidable.

Rice's theorem is useful for deciding properties like whether a language is empty, non-empty, regular, context-free etc.

It cannot be applied to questions like whether a TM has fewer than 7 states, a final state, a start state etc. These properties are of machines and not languages.

It also doesn't apply to questions like is a language a subset of Σ^* or whether a language of a RM is a language of a TM - these are trivial properties!

The consequences of this is that we cannot write programs which answer non-trivial questions about the black-box behaviours of programs.

9 Computability

Computably Enumerable

A set S is called computably enumerable if the enumeration function $f : \mathbb{N} \to S$ is computable.

In terms of RM and TM we can think of enumeration as outputting an infinite list as it executes forever.

The halting problem may be undecidable, but is it computably enumerable? Yes! Interleaving shows that H is computably enumerable.

All semi-decidable problems are computably enumerable, and any computably enumerable problem is semi-decidable - being semi-decidable is the same as being computably enumerable.

To prove that a problem P_2 is not c.e. we show that there is a mapping reduction from a known not-c.e. problem P_1 to P_2. We must use mapping reductions, as H is c.e. but L is not - but L is Turing reducible to H by flipping the answer.

Complexity Measures

When performing addition in a TM we have a time complexity of $\mathcal{O}(\log n)$, where it is $\mathcal{O}(n)$ for RMs - there's an exponential penalty for using a register machine!

Addition can be $\mathcal{O}(1)$ if we add dedicated `ADD(i, j)` and `SUB(i, j)` instructions - but this doesn't remove the inaccuracy it just makes it smaller.

Complexity is useful, but the measures are slightly bogus:

- $\mathcal{O}(n)$ is not always easy - if n is massive for example
- $\Omega(2^n)$ isn't always hard - there are problems much worse than this that are still solvable for real examples
- $\mathcal{O}(n^{10})$ and $\Omega(n^{10})$ seem extremely difficult, but a new model or algorithm could reduce that significantly
- Since we also ignore coefficients, we could have something like $f(n) \geq 10^{100} \log n$ which is slow but still only logarithmic - this isn't common enough to worry generally however.

Complexity Classes

Let $t : \mathbb{N} \to \mathbb{R}_{\geq 0}$. A time complexity class $\mathbf{TIME}(t(n))$ is the collection of all problems that are decidable by a deterministic machine (RM, TM etc.) in $\mathcal{O}(t(n))$ time.

Given $A = \{0^i 1^i \mid i \in \mathbb{N}\}$, a TM can decide this in $\mathcal{O}(n^2)$, which means that $A \in \mathbf{TIME}(n^2)$.

We also define $\mathbf{NTIME}(t(n))$ to be the collection of all problems decidable by a nondeterministic machine (NRM, NTM, etc.) in $\mathcal{O}(t(n))$.

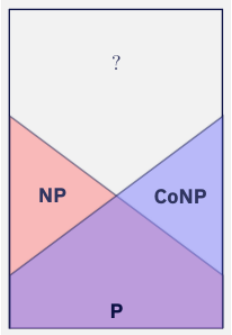

Figure 6: The three basic complexity classes

P: Polynomial Time

The polynomial complexity class $\mathbf{P}$ is the class of problems decidable with some deterministic polynomial time complexity.

$$\mathbf{P} = \bigcup_{k \in \mathbb{N}} \mathbf{TIME}(n^k)$$

Problems in $\mathbf{P}$ are called tractable. Any problem not in $\mathbf{P}$ is $\Omega(n^k)$ for every k.

The class itself is robust, reasonable changes to model don't change it and reasonable translations between problems preserves membership in $\mathbf{P}$.

A polynomially-bounded RM together with a polynomial (n^k for some k, without loss of generality), such that given an input w it will always halt after executing $|w|^k$ instructions. A problem Q is in $\mathbf{P}$ iff it is computed by such a machine.

NP: Nondeterministic-polynomial Time

The polynomial complexity class **NP** is the class of problems decidable with some nondeterministic polynomial time complexity.

$$\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(n^k)$$

We don't know if every exponentially bounded problem is in **NP**, we think that it's probably not the case.

CoNP

We don't know if the class **NP** is closed under complement. We cannot flip the result because the result of flipping in a nondeterministic machine involves turning it from angelic nondeterminism to demonic nondeterminism (otherwise known as co-nondeterminism) or vice versa.

AP: Alternating Polynomial Time

The class **AP** is the class of all problems decidable by an alternating machine in polynomial time with no restriction on swapping quantifiers.

AP is known to be equal to **PSPACE**.

PSPACE: Polynomial Space

An RM/TM is $f(n)$-space-bounded if it may use only $f(inputsize)$ space. For TMs this is the number of cells on the tape, where register machines uses bits in registers.

Polynomial Reductions

A polynomial reduction from $P_1 = (D_1, Q_1)$ to $P_2 = (D_2, Q_2)$ is a **P**-computable function $f : D_1 \to D_2$ such that $d \in Q_1$ iff $d \in Q_2$.

If P_2 is in **P**, then P_1 is in **P** straightforwardly. Therefore to prove a problem is not in **P** we can show that there is a polynomial reduction from a problem P_2 which isn't in **P** to our problem P_1

A problem P_1 is polynomially reducible to P_2, written $P_1 \leq_P P_2$ if there is a polynomially-bounded reduction from P_1 to P_2.

NP-Hard

A problem P is **NP**-Hard if for every $A \in \mathbf{NP}$, $A \leq_P P$.

That is, if a problem P_1 is **NP**-Hard and P_1 is polynomially reducible to P_2, then P_2 is also **NP**-Hard.

We can use this fact to prove other problems are **NP**-Hard by showing a reduction from a known **NP**-hard problem.

NP-Complete

A problem is **NP**-Complete if it is both **NP**-Hard and in **NP**.

The Cook-Levin Theorem

The Cook-Levin theorem states that the **NP** problem SAT is **NP**-Complete.

The proof of this involves showing it is **NP** by nondeterministically guessing assignments and checking them in polynomial time, and then showing it's **NP**-Hard by reducing any **NP** problem to SAT.

10 The Polynomial Heirarchy

Σs

The set Σ_1^P describes all problems which can be phrased as $\{y \mid \exists^P x \in \mathbb{N}.\, R(x, y)\}$, where R is a **P**-decidable predicate and $\exists^P x \ldots$ indicates that x is of size polynomial in the size of y.

We can say that x is a certificate showing which guesses can be made by our NRM giving an accepting run.

If a problem $Q \in \Sigma_1^P$ then Q is **NP**, because it is a problem for which we can verify the answer in polynomial time. If a problem Q is in **NP** then $Q \in \Sigma_1^P$.

So, $\mathbf{NP} = \Sigma_1^P$

Πs

The set Π_1^P describes all problems that can be phrased as $\{y \mid \forall^P x \in \mathbb{N}.\, R(x, y)\}$, where R is a **P**-decidable predicate and $\forall^P x \ldots$ indicates that x is of size polynomial in the size of y.

$\Pi_1^P = \overline{\Sigma_1^P}$, and $\Pi_1^P = \mathbf{CoNP}$

Δs

There are two subtly conflicting definitions of Δ_1^P:

1. The set Δ_1^P describes the intersection of Σ_1^P and Π_1^P
2. The set Δ_1^P describes the set **P**

Moving Higher

The next layer of the heirarchy goes as follows:

- Σ_2^P is all problems of the form $\{x \mid \exists^P y.\, \forall^P z.\, R(x, y, z)\}$
- Π_2^P is all problems of the form $\{x \mid \forall^P y.\, \exists^P z.\, R(x, y, z)\}$
- $\Delta_2^P = \Sigma_2^P \cap \Pi_2^P$

We can also use oracles to get an alternate definition:

- Δ_2^P is all problems that are decidable in polynomial time by some deterministic RM/TM with an $\mathcal{O}(1)$ oracle for some problem in Σ_1^P (it is **P** with an $\mathcal{O}(1)$ oracle for **NP**)

- Σ_2^P allows the TM/RM to be nondeterministic (it is **NP** with an $\mathcal{O}(1)$ oracle for **NP**)
- Π_2^P is **CoNP** with an oracle for **NP**

In general for any $n > 1$:

- Δ_n^P is all problems decidable by a deterministic polynomially bounded TM/RM with an $\mathcal{O}(1)$ oracle for some problem in Σ_{n-1}^P.
- Σ_n^P is all problems decidable by some nondeterministic polynomially bounded TM/RM with an $\mathcal{O}(1)$ oracle for some problem in Σ_{n-1}^P.
- Π_n^P is all problems decidable by some co-nondeterministic polynomially bounded TM/RM with an $\mathcal{O}(1)$ oracle for some problem in Σ_{n-1}^P.

Alternation

Equivalently to the general definition given above, Σ_n^P can be defined as all problems that can be phrased as some alternation of polynomially bounded quantifiers starting with $\exists^P$.

Π_n^P has a similar definition, starting instead with $\forall^P$

Alternating machines combine the acceptance modes of both angelic and demonic nondeterministic machines.

Alternating register machines would replace the NRM's `MAYBE` instruction with the $\texttt{MAYBE}^{\exists}$ and $\texttt{MAYBE}^{\forall}$ instructions, which are nondeterministic branching choices where acceptance depends on if one branch accepts ($\exists$) or both branches accept ($\forall$).

Alternating Turing Machines are defined by labelling states with either $\forall$ or $\exists$.

The class Σ_n^P can therefore be described as the class of problems decided in polynomial time by an alternating machine that initially uses $\exists$-nondeterminism and swaps quantifiers at most $n-1$ times. This extends to Π_n^P swapping starting with $\exists$ to starting with $\forall$.

11 λ-Calculus

The λ-Calculus is a method of computation which resembles a programming language, unlike register machines and turing machines which are much closer to machines.

One of the advantages of this is that λ-Calculus is "higher-order", meaning that computations can take other computations as input - it doesn't need to rely on machine encodings.

Syntax

λ-calculus computations are expressed as λ-terms:

$$\begin{array}{rcl} t & ::= & x \text{ (variables)} \\ & | & t_1\ t_2 \text{ (application)} \\ & | & \lambda x.\ t \text{ (λ-abstraction)} \end{array}$$

A λ-term $(\lambda x.\ y)$ can be thought of as a function that given an input bound to the variable x, returns the term y.

Function application is left associative:

$$f\ a\ b\ c = ((f\ a)\ b)\ c$$

λ-abstraction extends as far as possible:

$$\lambda a.\ f\ a\ b = \lambda a.\ (f\ a\ b)$$

All functions are unary, multiple argument functions are modeled via nested λ-abstractions:

$$\lambda x.\ \lambda y.\ f\ y\ x$$

λ-calculus is higher-order, in that functions may be arguments to functions themselves:

$$\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$$

α-equivalence

α-equivalence is a way of saying that two statements are semantically identical but differ in the choice of bound variable names.

$$e_1 = (\lambda x.\ \lambda x.\ x + x)$$

$$e_2 = (\lambda a.\, \lambda y.\, y + y)$$

These two statements are α-equivalent, we say that $e_1 \equiv_\alpha e_2$, the relation $\equiv_\alpha$ is an equivalence relation. The process of consistently renaming variables that preserves α-equivalence is called α-renaming or α-conversion.

Substitution

A variable x is free in a term e if x occurs in e but is not bound by a λ-abstraction in $e\cdot$ The variable x is free in $\lambda y.\, x + y$, but not in $\lambda x.\, \lambda y.\, x + y$.

A substitution, written $e[^t/_x]$ is the replacement of all free occurrences of x in e with t.

Variable Capture

Capture can occur for a substitution $e[^t/_x]$ whenever there is a bound variable in the term e with the same name as a free variable occurring in t.

Fortunately it is always possible to avoid capture by α-rename the offending bound variable to an unused name.

β-reduction

The rule to evaluate function applications is called β-reduction:

$$(\lambda x.\, t)\ u \ \mapsto_\beta \ t[^u/_x]$$

β-reduction is a congruence:

$$\frac{}{(\lambda x\, t)\ u \mapsto_\beta t[^u/_x]}$$

$$\frac{t \mapsto_\beta t'}{s\ t \mapsto_\beta s\ t'} \quad \frac{s \mapsto_\beta s'}{s\ t \mapsto_\beta s'\ t} \quad \frac{t \mapsto_\beta t'}{\lambda x.\, t \mapsto_\beta \lambda x.\, t'}$$

This means we can pick any reducible subexpression (known as a redex) and perform β-reduction.

Confluence

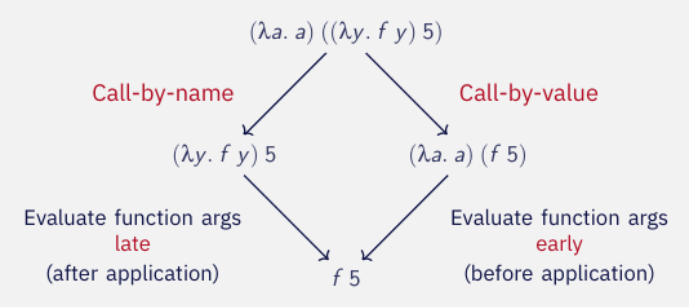

Figure 7: There are often many different ways to reduce the same expression

The Church-Rosser Theorem states that if a term t β-reduces to two terms a and b, then there is a common term t' to which both a and b are β-reducible.

Equivalence

The notion of confluence means we can define another notion of equivalence which equates more than α-equivalence.

Two terms are $\alpha\beta$-equivalent - written $\equiv_{\alpha\beta}$ - if they β-reduce to α-equivalent terms.

Furthermore there is another kind of equation that cannot be proved from β-equivalence alone called η-reduction:

$$(\lambda x.\ f\ x) \mapsto_\eta f$$

Adding this reduction to the system preserves confluence (and therefore the uniqueness of normal forms), so we have a notion of $\alpha\beta\eta$-equivalence also.

Normal Forms

A λ-term which cannot be reduced further is called a normal form.

Not every term has a normal form. Thankfully, all terms that do have a normal form are unique due to the Church-Rosser theorem.

This example reduces to itself: $(\lambda x.\ x\ x)(\lambda x.\ x\ x)$

Church Encodings

In order to demonstrate that λ-calculus is a usable programming language we need to show how to encode certain useful structures and primitives like

booleans and natural numbers with their operations as λ-terms.

The general idea is to turn a data type into the type of its eliminator, in other words we make a function which serves the same purpose as the data type when used.

Booleans

We use booleans to choose between results, so the encoding of a boolean is a function that given two arguments returns the first if true and the second if false.

$$\text{True} \equiv \lambda a.\ \lambda b.\ a$$

$$\text{False} \equiv \lambda a.\ \lambda b.\ b$$

An If statement becomes:

$$\text{If} \equiv \lambda c.\ \lambda t.\ \lambda e.\ c\ t\ e$$

Church Numerals

We use natural numbers to repeat processes n times, so the encoding of a natural number is a function that takes a function f and a value x and will apply f to x that number of times.

$$\begin{aligned}\text{Zero} &\equiv \lambda f.\ \lambda x.\ x\\ \text{One} &\equiv \lambda f.\ \lambda x.\ f\ x\\ \text{Two} &\equiv \lambda f.\ \lambda x.\ f\ (f\ x)\\ &\vdots\end{aligned}$$

Then operations like the successor and addition are relatively simple:

$$\begin{aligned}\text{Suc} &\equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)\\ \text{Add} &\equiv \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)\end{aligned}$$

The $\mathcal{Y}$ Combinator

The $\mathcal{Y}$ combinator is a "fixed point combinator", and it's the way we achieve recursion in λ-calculus. One $\mathcal{Y}$ combinator is the following:

$$\mathcal{Y} \equiv (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$$

Simply Typed λ-Calculus

Higher-Order Logic

Originally the λ-calculus was intended for use as a term language for a logic, called higher-order logic.

The existence of the $\mathcal{Y}$ combinator and recursion in general causes a problem for this because we can cause statements like:

$$\mathcal{Y}\,\neg \equiv_\beta \neg\,(\mathcal{Y}\,\neg)$$

A logic which can have statements equivalent to their own negation is not ideal at all. To solve this church introduced types.

Types

Typed λ-calculus has a set of base types like `nat` and `bool`.

Given two types σ and τ, $\sigma \to \tau$ is the type signature of a function from σ to τ, type signatures are right associative:

$$\sigma \to \tau \to \rho = \sigma \to (\tau \to \rho)$$

λ-abstraction also specifies the type of the parameter: $\lambda x : \tau.\, t$

Things like the $\mathcal{Y}$ combinator or other recursive structures (things like the term which β-reduces to itself) cannot be typed.

Natural Deduction

We can specify a logical system as a deductive system by providing a set of rules and axioms that describe how to prove various connectives. The same can be done for typing.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash t\ u : \tau} A \qquad \frac{x : \sigma, \Gamma \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma.\, t) : \sigma \to \tau} I$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t\ u : \tau} E$$

This is the full rule-set for simply typed λ-calculus. The structure is that from the bottom you can derive the top, and that the right of the $\vdash$ symbol can be assumed true as a result of the left of it.

A is assignment, I is introduction, and E is elimination.

The Results of Simply Typed λ-calculus

This simply typed λ-calculus has the following properties:

- Uniqueness of types: In a given context (types for free variables), any simply typed λ-terms has at most one type. Deciding this is in **P**.
- Subject reduction (or type safety): Typing respects $\equiv_{\alpha\beta\eta}$, i.e. reduction does not affect a term's type.
- Strong normalisation: Any well-typed term evaluates in finitely many reductions to a unique irreducible term. If the type is a base type, this term is a constant.

The $\mathcal{Y}$ combinator cannot be typed, and strong normalisation means that such a term cannot exist.

If we want to do general computation in λ-calculus we need recursion back. The way this is done is by extending it to include a new built in feature called **fix**.

We extend β-reduction to unroll recursion in a single step (which will keep strong normalisation).

Some type-theoretic languages avoid adding general recursion to the underlying λ-calculus - for reasons which are not examinable.

12 Definitions

Ambiguity

A grammar is **ambiguous** if there is more than one valid parse tree for a given string. This can cause problems with parsing and interpretation.

Bijectivity

A binary relation $a \sim b$ is **bijective** if it is both injective and surjective.

Binary Relations

A **binary relation** associates elements of one set with elements of another, $a \sim b$ says that a is related to b.

Relations are "one-to-many", meaning that a can relate to more than one value.

Examples of a binary relations between numbers are $=$, $<$, and $\geq$.

Closure

A set S is **closed** under an operation if that operation performed on elements of S always produces another valid element of S.

The natural numbers are closed under addition, as adding two natural numbers together always produces a natural number.

They are not closed under subtraction however, as the result of $a - b$ when $b > a$ is not a natural number.

Angelic and Demonic Non-determinism

A machine with **angelic** non-determinism accepts if at least one non-deterministic path reaches an accepting state. Angelic non-determinism is sometimes represented using $\exists$.

A machine with **demonic** non-determinism accepts only if every non-deterministic path reaches an accepting state. Demonic non-determinism is sometimes represented using $\forall$.

Conjunctive Normal Form

A clause ϕ is in conjunctive normal form if it is of the form:

$$\bigwedge_i \bigvee_j P_{ij}$$

Where P_{ij} is a literal (either a variable P or a negation of one $\neg P$.

ϕ is in k-CNF if each clause $\bigvee P_{ij}$ has at most k literals.

Computable Functions

A total function $\mathbb{N} \to \mathbb{N}$ is **computable** if there is a RM/TM which computes f, i.e. given an x in R_0 leaves $f(x)$ in R_0.

Decidability

A decision problem (D, Q) is decidable or computable if $d \in Q$ is characterised by a computable function $f : D \to \{0, 1\}$.

A problem which is both semi-decidable and co-semi-decidable is decidable. There are problems which are neither semi-decidable or co-semi-decidable.

Semi-decidability

The halting problem is not symmetric. If $M \in H$ we can determine this: run M - when it halts it will say "yes". But, if $M \notin H$ we can't determine this by running M as it will run forever. The halting problem is semi-decidable.

A decision problem (D, Q) is semi-decidable if there is a RM/TM that returns "yes" for every $d \in Q$ but may return "no" or loop forever when $d \notin Q$.

If a problem P is semi-decidable, its complement $\overline{P}$ is co-semi-decidable.

Semi-decidable problems are sometimes called recognisable.

Co-semi-decidability

The looping problem is not symmetric. If $M \notin L$ we can determine this: run M - when it halts it will say "no". But, if $M \in L$ we can't determine this by running M as it will run forever. The looping problem is co-semi-decidable.

A decision problem (D, Q) is co-semi-decidable if there is a RM/TM that returns "no" for every $d \notin Q$ but may return "yes" or loop forever when $d \in Q$.

If a problem P is co-semi-decidable, its complement $\overline{P}$ is semi-decidable.

Decision Problems

A decision problem is a set D and a query subset $Q \subseteq D$. Decision problems can be decidable, semi-decidable, or co-semi-decidable.

In language problems (DFAs and CFGs etc.) are decision problems where $D = \Sigma^*$ and Q is the language in question.

Other such problems are $D = \mathbb{N}$ and $Q =$ Primes, or $D =$ RMs and $Q =$ halting RMs

Decidable languages are closed under union, intersection, and complement

Determinism and Non-determinism

A function $f : A \to B$ is **deterministic** if when given a particular input in A, will always produce the same output from B.

A function $f : A \to B$ is **non-deterministic** if when given a particular input in A doesn't always produce the same output from B.

Enumerable

A set S is **enumerable** if there is a bijection between S and $\mathbb{N}$.

Equivalence Classes

An **equivalence class** of a set S is some subset of S such that every element of S has an equivalence relation to every other element of S.

Every element of S is contained in exactly one equivalence class, though it may be the only element in that class.

Equivalence Relations

A binary relation $a \sim b$ is an **equivalence relation** if it is reflexive, symmetric, and transitive.

Equality ($=$) is an equivalence relation because it can be shown to be reflexive, symmetric, and transitive. Having the same birthday is an equivalence relation on the set of birthdays.

Injectivity

A binary relation $a \sim b$ is **injective** if every element in the domain maps to one unique element of the range, injective functions are "one-to-one" mappings.

Interleaving

The set of valid RM descriptions is decidable, given $n \in \mathbb{N}$ we can check whether $n = \ulcorner M \urcorner$ for some machine M.

This means we can sequentially enumerate all machines $\ulcorner M_0 \urcorner, \ulcorner M_1 \urcorner, \ldots$

Interleaving is the process of running all register machines in sequence one step at a time, and checking some decision problem as we go.

```
0  ms = ⟨⟩; i = 0;
1
2  while true {
3      ms = ms ++ ⟨⌜M_i⌝⟩;
4      for ⌜M⌝ in ms {
5          run M for one step, and update ms;
6          if [[condition]] is met {
7              output ⌜M⌝;
8              delete M from ms;
9          }
10     }
11 }
```

Kleene Star

The **Kleene closure** of a language L - written L^* - is the language of strings that consist wholly of zero or more strings in L

$$L^* = \bigcup_{i \in \mathbb{N}} L^i$$

Regular languages are closed under Kleene Closure

Machine Properties

A property is a set of RM/TM descriptions

Nontrivial Properties

A property is nontrivial if it contains some but not all descriptions.

Trivial properties hold for all languages or for none.

Semantic Properties

A property P is semantic if:

$$\mathcal{L}(M_1) = \mathcal{L}(M_2) \Rightarrow (\ulcorner M_1 \urcorner \in P \Leftrightarrow \ulcorner M_2 \urcorner \in P)$$

In other words, it concerns the language and not the particular implementation of the machine.

Nondeterminism in RM and TM

We can have nondeterministic RM and TM just as we can have nondeterministic finite automata.

For register machines this involves adding a special instruction `MAYBE(j)` which will either nondeterministically jump to to I_j or do nothing.

An NRM accepts if there is some run (a sequence of instructions through the choices) that halts and accepts. The presence of infinite runs doesn't matter here because we only care about if there are accepting finite runs.

An NRM has the same deciding power as a regular RM, as we can use interleaving to simulate all runs of an NRM at once on a regular RM - however, in time n an RM can only explore $\mathcal{O}(n)$ possibilities where an NRM can explore $2^{\mathcal{O}(n)}$ possibilities.

All of the above is true of NTMs, with some minor differences to get the same results.

Oracles

Given a decision problem (D, Q), an **oracle** for Q is a "magic" RM instruction $\texttt{ORACLE}_Q(i)$ which, given an encoding of $d \in D$ in R_i sets R_i to contain 1 iff $d \in Q$.

An oracle is only useful when it predicts for an undecidable problem, as an oracle for a decidable problem could be replaced by a a program which decidably computes that result.

Pairing Functions

A **pairing function** is an injective function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

An example is $f(x, y) = 2^x 3^y$

We write $\langle x, y \rangle_2$ to denote the result of pairing two values x and y. If $z = \langle x, y \rangle_2$ then we say $z_0 = x$ and $z_1 = y$.

A function which pairs two numbers is enough to generalise to cram an arbitrarily long sequence of numbers into one.

Parse Trees

A parse tree is a tree that shows how to devise a string of characters from a non-terminal state of a CFG.

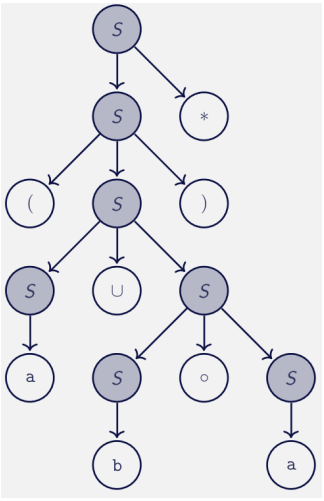

Figure 8: The yield of this tree is $(a \cup b^*) \circ a$

The yield of a parse tree is the concatenation of all the symbols at its leaves. If the route of the tree is S then the yield $x \in \mathcal{L}(G)$.

Parse trees are typically left-associative, meaning that we parse assuming that left bracketing is correct.

Pigeonhole Principle

The **pigeonhole principle** states that if you have n items are put into m containers, where $n > m$, at least one of the containers must have more than one item in it.

Power Set

The **power set** of a given set S, denoted $\mathcal{L}(S)$ is the set containing all subsets of S, including $\emptyset$ and S itself.

$$\mathcal{P}(S) = \{s \mid s \subseteq S\}$$

Given $n = |S|$, $|\mathcal{P}(S)| = 2^n$.

Reflexivity

A binary relation is **reflexive** if every value relates to itself, i.e. $a \sim a$.

Equality ($=$) is a reflexive binary relation between two numbers, where strictly less than ($<$) is not - because $a < a$ isn't true.

Satisfiability

SAT

SAT is a decision problem which asks the following question: given a boolean formula ϕ over a set of boolean variables X_i, is there an assignment of values to X_i which satisfies ϕ?

$(A \vee B) \wedge (\neg B \vee C) \wedge (A \vee C)$ is satisfiable by making $A, C = \top$

The size of a SAT problem is the number of symbols in ϕ.

3SAT

3SAT is the problem of determining satisfiability for formula given in 3-CNF.

Sequential Composition

The **sequential composition** of two languages L_1 and L_2 - written L_1L_2 - is the language of strings that consist of a string in L_1 followed by a string in L_2.

$$L_1L_2 = \{vw | v \in L_1, w \in L_2\}$$

Regular languages are closed under sequential composition.

Surjectivity

A binary relation $a \sim b$ is **surjective** if every element b in the co-domain B is mapped to by at least one element a in the domain A.

Symmetry

A binary relation is **symmetric** if $a \sim b \Leftrightarrow b \sim a$.

Equality ($=$) is symmetric, as $a = b$ means $b = a$. Greater than or equal ($\geq$) is not symmetric, as $a \geq b$ doesn't necessarily mean $b \geq a$.

Time Complexity

The time complexity of a deterministic machine M that halts on all inputs is a function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of size n.

Big $\mathcal{O}$

Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Say that $f(n) \in \mathcal{O}(g(n))$ if there exists $c, n_0 > 0$ such that for all $n > n_0$:

$$f(n) \leq c \cdot g(n)$$

Big Ω

Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Say that $f(n) \in \Omega(g(n))$ if there exists $c, n_0 > 0$ such that for all $n > n_0$:

$$f(n) \geq c \cdot g(n)$$

Total and Partial Functions

A **total** function $f : A \to B$ is a function which is defined for every input value in A.

A **partial** function $f : A \rightharpoonup B$ is a function which may not be defined on every input value in A, it returns either an element of B or $\perp$.

Transitivity

A binary relation is **transitive** if when $a \sim b$ and $b \sim c$, $a \sim c$.

Equality $(=)$ is transitive, because if $a = b$ and $b = c$ we know $a = c$. Strictly less than $(<)$ is also transitive, as if $a < b$ and $b < c$, we know $a < c$.

Turing Machines

A turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:

- Q : states
- Σ : input symbols
- $\Gamma \supseteq \Sigma$: tape symbols, including a blank symbol $\llcorner$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$: a transition function
- $q_0, q_{accept}, q_{reject} \in Q$: start, accept, and reject states

Turing machines are equivalent in power to register machines, the proof of which involves creating an RM to simulate a TM, and vice versa.

The upshot of this equivalence is that the halting problem applies to TMs just as with RMs.

None of the following adds any extra expressively: the ability to stay in place, making the tape unbounded in both directions, restricting the number of symbols, having multiple tapes, and adding non-determinism.

Turing Transducers

A **Turing Transducer** is a RM/TM which takes an instance d of a problem $P_1 = (D_1, Q_1)$ in R_0 and halts with an instance $d' = f(d)$ of $P_2 = (D_2, Q_2)$ in R_0. Thus, f is a computable function $D_1 \to D_2$.

Undecidable Problems

$$A$$

$$A = \{\langle \ulcorner M \urcorner, w\rangle \mid M \text{ accepts } w\}$$

This is analogous to the proof for the halting problem.

$$NotEmpty$$

$$NotEmpty = \{\ulcorner M \urcorner \mid \mathcal{L}(M) \neq \emptyset\}$$

We can sketch a mapping reduction from A. Given an instance $\langle M, w\rangle$ of A, our reduction constructs a machine M' whose language is either $\{w\}$ or $\emptyset$.

Given input x, it will reject if $x \neq w$, else run M on w. We know that $\langle M, w\rangle \in A$ iff $M' \in NotEmpty$ too is undecidable.

Uniform Halting: UH

$$UH = \{\ulcorner M \urcorner \mid M \text{ halts on all inputs}\}$$

This is a demonic version of the halting problem. We can reduce from H to UH. Given a machine M and an input w, build a machine M' which ignores its input, writes w to the tape, and then behaves as M. Then M' halts on any inut iff M halts on w.

The Looping Problem: L

Let L be the subset of RMs/TMs that go into an infinite loop.

Since L is the complement of H this seems easy, but we can't fit it neatly into our definition of a mapping reduction.

However, since L is Turing reducible to H we can show it is undecidable.