The Church-Turing Thesis

Chapter 18
Are We Done?

FSM $\Rightarrow$ PDA $\Rightarrow$ Turing machine

Is this the end of the line?

There are still problems we cannot solve:

- There is a countably infinite number of Turing machines since we can lexicographically enumerate all the strings that correspond to syntactically legal Turing machines.

- There is an uncountably infinite number of languages over any nonempty alphabet.

- So there are more languages than there are Turing machines.
What Can Algorithms Do?

1. Can we make all true statements theorems?

2. Can we decide whether a statement is a theorem?
Gödel’s Incompleteness Theorem

Kurt Gödel showed, in the proof of his Incompleteness Theorem [Gödel 1931], that the answer to question 1 is no. In particular, he showed that there exists no decidable axiomatization of Peano arithmetic that is both consistent and complete.
The Entscheidungsproblem

Does there exist an algorithm to decide, given an arbitrary sentence \( w \) in first order logic, whether \( w \) is valid?

Given a set of axioms \( A \) and a sentence \( w \), does there exist an algorithm to decide whether \( w \) is entailed by \( A \)?

Given a set of axioms, \( A \), and a sentence, \( w \), does there exist an algorithm to decide whether \( w \) can be proved from \( A \)?
The Entscheidungsproblem

To answer the question, in any of these forms, requires formalizing the definition of an algorithm:

- Turing: Turing machines.
- Church: lambda calculus.

Turing proved that Turing machines and the lambda calculus are equivalent.
Church's Thesis
(Church-Turing Thesis)

All formalisms powerful enough to describe everything we think of as a computational algorithm are equivalent.

This isn’t a formal statement, so we can’t prove it. But many different computational models have been proposed and they all turn out to be equivalent.
The Church-Turing Thesis

Examples of equivalent formalisms:

- Modern computers (with unbounded memory)
- Lambda calculus
- Partial recursive functions
- Tag systems (FSM plus FIFO queue)
- Unrestricted grammars:
  \[ aS_a \rightarrow B \]
- Post production systems
- Markov algorithms
- Conway’s Game of Life
- One dimensional cellular automata
- DNA-based computing
- Lindenmayer systems
The Lambda Calculus

The successor function:

\[(\lambda \, x. \, x + 1)\]

\[(\lambda \, x. \, x + 1) \, 3 = 4\]

Addition:

\[(\lambda \, x. \, \lambda \, y. \, x + y) \, 3 \, 4\]

This expression is evaluated by binding 3 to \(x\) to create the new function \((\lambda \, y. \, 3 + y)\), which is applied to 4 to return 7.

In the pure lambda calculus, there is no built in data type number. All expressions are functions. But the natural numbers can be defined as lambda calculus functions. So the lambda calculus can effectively describe numeric functions just as we have done.
Tag Systems

A tag system (or a Post machine) is an FSM augmented with a FIFO queue.

A tag system for WW:

What about a tag system for PalEven?
The Power of Tag Systems

Tag systems are equivalent in power to Turing machines because the TM’s tape can be simulated with the FIFO queue.

Suppose that we push $abcde$ onto the queue:

```
abcde
```

To read the queue, we must remove the a first.

But suppose we want to remove e first:
The Power of Tag Systems

Tag systems are equivalent in power to Turing machines because the TM’s tape can be simulated with the FIFO queue.

Suppose that we push \textit{abcde} onto the queue:

\begin{center}
\begin{tabular}{cccccc}
  & a & b & c & d & e \\
\end{tabular}
\end{center}

To read the queue, we must remove the \textit{a} first.

But suppose we want to remove \textit{e} first:

\begin{center}
\textbf{Treat the queue as a loop.}
\end{center}
Post Production Systems

A **Post system** $P$ is a quintuple $(V, \Sigma, X, R, S)$:

- $V$ is the rule alphabet,
- $\Sigma$ is a subset of $V$,
- $X$ is a set of variables whose values are drawn from $V^*$,
- $R$ (the set of rules) is a finite subset of:
  $$(V \cup X)^* \times (V \cup X)^*$$
  Every variable on the RHS must also be on the LHS.
- $S$ can be any element of $V - \Sigma$.

- LHS may contain multiple symbols.
- Rules may contain variables.
- LHS must match entire string. So, $A \rightarrow B$ becomes:

  $$XAY \rightarrow XBY$$
A Post Production System

WW = \{ww : w \in \{a, b\}^*\}.

P = (\{S, a, b\}, \{a, b\}, \{X, Y\}, R, S), R =

(1) XS \rightarrow XaS  \quad /* Generate (a \cup b)^* S.
(2) XS \rightarrow XbS  \quad /* Create a second copy of X.
(3) XS \rightarrow XX

This Post system can generate, for example, the string abbabb:

S \Rightarrow (using (1) and letting X match \varepsilon)

aS \Rightarrow (using (2) and letting X match a)

abS \Rightarrow (using (2) and letting X match ab)

abbS \Rightarrow (using (3) and letting X match abb)

abbabb
A Markov algorithm $M$ is a triple $(V, \Sigma, R)$, where:

- $V$ is the rule alphabet, which contains both working symbols and input symbols. If $M$ is a decider or a semidecider, $V$ will contain two special working symbols, $\text{Accept}$ and $\text{Reject}$.
- $\Sigma$ is a subset of $V$, and
- $R$ is an ordered list of rules, each of which is an element of $V^* \times V^*$. There are two kinds of rules:
  - Continuing, written as $X \rightarrow Y$, and
  - Terminating, written as $X \rightarrow \bullet Y$

As with Turing machines, there is no special start symbol, but there will be an input string.
A Markov Algorithm Interpreter

\textit{Markov algorithm}(M, w) =

1. Until no rules apply or the process has been terminated by executing a terminal rule do:
   1.1 Find the first rule in the list \( R \) that matches against \( w \).
       If that rule matches \( w \) in more than one place, choose the leftmost match.
   1.2 If no rule matches then exit.
   1.3 Apply the matched rule to \( w \) by replacing the substring that matched the rule’s LHS with the rule’s RHS.
   1.4 If the matched rule is a terminating rule, exit.
2. If \( w \) contains the symbol \textit{Accept} then accept.
3. If \( w \) contains the symbol \textit{Reject} then reject.
4. Otherwise, return \( w \).

Notice: Markov algorithms are deterministic.
A Markov Algorithm for $A^nB^nC^n$

(1) $#a \rightarrow \%$
(2) $#b \rightarrow \bullet \ Reject$
(3) $#c \rightarrow \bullet \ Reject$
(4) $\%a \rightarrow a\%$
(5) $\%b \rightarrow ?$
(6) $\% \rightarrow \bullet \ Reject$
(7) $?b \rightarrow b?$
(8) $?c \rightarrow \epsilon$
(9) $? \rightarrow \bullet \ Reject$
(10) $# \rightarrow \bullet \ Accept$
(11) $\epsilon \rightarrow #$

Example: aabbbcccc
The Power of Markov Algorithms

*Theorem:* The Markov algorithm formalism is equivalent in power to the Turing machine.

*Proof:* By two constructions. Show that each formalism can simulate the other.
The Game of Life

Playing the game

At each step of the computation, the value for each cell is determined by computing the number of neighbors (up to a max of 8) it currently has, according to the following rules:

- A dead cell with exactly three live neighbors becomes a live cell (birth).
- A live cell with two or three live neighbors stays alive (survival).
- In all other cases, a cell dies or remains dead (overcrowding or loneliness).

We’ll say that a game halts iff it reaches some stable configuration.
Elementary Cellular Automata

Wolfram’s Rule 110 is a universal computer if you can figure out how to encode the program and the input in the initial configuration:
DNA Computing

Let $G$ be a directed graph, with one vertex $s$ designated as the start vertex and another vertex $d$ designated as the end vertex. A **Hamiltonian path** through $G$ is a path that begins at $s$, ends at $d$, and visits each other node in $G$ exactly once.

The **Hamiltonian path problem**: given a directed graph $G$, with designated $s$ and $d$, does there exist a Hamiltonian path through it?
Adelman’s Procedure

1. He chose a particular graph $G$ (with 7 vertices and 14 edges).

2. He encoded each vertex of $G$ as a sequence of 8 nucleotides. For example, a vertex might be represented as $ACCTGCAG$.

3. He encoded each directed edge of $G$ as a sequence of 8 nucleotides, namely the last four from the encoding of the start vertex and the first four from the encoding of the end vertex. So an edge from $ACTTGCAG$ to $TCGGACTG$ would be encoded as:

   $GCAGTCGG$. 
Adelman’s Procedure

4. He synthesized many copies of each of the edge sequences, as well as many copies of the DNA complements of all the vertex encodings.

Example: The complement of **ACTTGCAG** is **TGAACGTC**.

5. He combined the vertex-complement molecules and the edge molecules in a test tube, along with water, salt, some important enzymes, and a few other chemicals required to support the natural biological processes.
6. He allowed to happen the natural process by which complementary strands of DNA in solution will meet and stick together (anneal).

Adding another edge:
Adelman’s Procedure

7. He allowed a second biological reaction to occur. The enzyme ligase that had been added to the mixture joins adjacent sequences of DNA.

8. He used the polymerase chain reaction (PCR) technique to make massive numbers of copies of exactly those sequences that started at the start vertex and ended at the end one. Other sequences were still present in the mix after this step, but in much lower numbers.

9. He used gel electrophoresis to select only those molecules whose length corresponded to a Hamiltonian path through the graph.
Adelman’s Procedure

10. He checked that each of the remaining vertices occurred in the selected molecules. For each intermediate vertex do:
   10.1. Use a DNA “probe” to attract molecules that contain the DNA sequence for the vertex that is being checked.
   10.2. Use a magnet to attract the probes.
   10.3. Throw away the rest of the solution.

11. Only molecules that corresponded to paths that started at the start vertex, ended at the end vertex, had the correct length for a path that visited each vertex exactly once, and contained each of the vertices could still be present. So if any DNA was left, a Hamiltonian path existed.