Some Basic Turing-Complete Systems

Parth Deshmukh

Outline

Section 1

Turing-completeness

What are some Turing-complete systems?

- Turing machines
- Lambda calculus, formal grammars and languages
- Most programming languages (all of the useful ones)
- Electrical circuits with NAND gates (or NOT and AND/OR gates)
- TeX, SQL (more on that later!)
- (kind of) Minecraft, Cities: Skylines, Dwarf Fortress, Magic: The Gathering, you could probably add to this list

What does it mean to be Turing-complete?

It runs DOOM (eventually)

- A system of computation is Turing-complete if it can simulate any Turing machine
- Turing machines are *abstract computers*
  - Memory is an infinite tape
  - Computation is performed by following a table of states
  - Importantly, Turing machines have *infinite time and space*
- To show something is Turing-complete, we can show that it simulates an arbitrary Turing machine

Why Turing machines?

Some philosophy

- We want to define what it means to compute some function $f : \mathbb{N} \to \mathbb{N}$ for any input x
- Turns out, two important ways to define this are equivalent:
    1. Turing-computable functions can be run on a Turing machine
    2. λ-computable functions can be written (and solved) in the language of lambda calculus: composing building block functions
- The Church-Turing thesis [Cop24] posits that these definitions are enough to define all computable functions - there is no challenge to this
- The efficient Church-Turing thesis, in comparison, posits these define all *realistically computable* functions - this is up to debate

Section 2

Tag systems

Motivations

- It can be useful to find simple Turing-complete systems to study
- Most frequently, it's easiest to show a system of interest is Turing-complete through one or more intermediary steps
- We will demonstrate this through *cyclic tag systems*

Tag systems

Definition

A m-tag system is specified via a triplet (m, A, P) :

1. m : the deletion number, $m \geq 1$
2. A : the alphabet, a set of symbols
3. P : a set of production rules, essentially a function $P : A \to A^*$

A tag system takes as input a word W in A^*. Let w be the first letter in W. At each step, it removes the first m letters and concatenates $P(w)$ to the end of W, re-reading w as it goes. As the tag system iterates, we denote W to be the register.

Halt states

Unlike Turing machines, tag systems can halt in two ways:

1. Either the machine halts if the register goes below a set size...
2. ...or we include a halting symbol $h \in A$ that halts the machine when it reaches the head (when $P(h)$ is called).

A worked example

This example is from [De 08] and calculates the Collatz/hailstone sequence of the given number.

$$m = 2$$

$$A = \{a, c, y\}$$

$$P = \begin{cases} a & \to cy \\ c & \to a \\ y & \to aaa \end{cases}$$

A number n is represented with repeating a n times. We'll start with $n = 5$, so the starting word is $aaaaa$. We halt when our register has only one symbol in it.

Doing the worked example

- First letter is a: we append cy, and remove 2 letters from the front, so $aaaaa \rightarrow aaacy$
- Similarly, $aaacy \rightarrow acycy \rightarrow ycycy$
- Now, the first letter is y: append aaa, so $ycycy \rightarrow ycyaaa \rightarrow yaaaaaa \rightarrow aaaaaaaa = 8$
- We've computed two steps in one: $5 \rightarrow 3(5) + 1 \rightarrow \frac{3(5)+1}{2} = 8$
- If you continue from here, you'll see the production rule $c \rightarrow a$ is used to divide our number by 2
- We halt when our register has one symbol, which ends up being $a \implies$ we halt iff the Collatz sequence terminates

Tag systems are Turing-complete
(trust me bro)

Theorem

2-tag systems are Turing-complete [CM64].

Proof

Cocke and Minsky show a construction to make a tag system simulating a given Turing machine. The construction is too complicated to cover here; it uses 17 symbols per state in the Turing machine it wants to simulate, and works by representing Turing machine states with special words.

Section 3

Cyclic tag systems

Can we go simpler?

Tag systems are tough to work with!

- The answer is yes, actually
- Cyclic tag systems can simulate any tag system and have much simpler specifications, making them way easier to work with
- The downside is that $P(x)$ has a different function signature
- Halting is also more confusing

Cyclic tag systems

Definition

A cyclic tag system is a modified tag system where $m = 1$ and $A = \{0, 1\}$. The set of production rules is a function $P : \mathbb{Z}_n \to A^*$ where n is the number of production rules.

A cyclic tag system takes as input a word W in A^* and additionally tracks its step number $k \geq 1$. At each step, it removes the first letter w, concatenating $P(k)$ to the end of W only if $w = 1$.

Some notes on the definition

- This is a non-standard but equivalent definition; usually P is taken to be an ordered list of words instead of a function
- Why this definition? Since choice of P *exactly specifies a cyclic tag system!*
  - Therefore, there are as many cyclic tag systems with n rules as there are functions $\mathbb{Z}_n \to A^*$
  - Also, this definition is easier to code: removes one pointer
- We can rewrite the updating rule at each step compactly:

$$(W, k) \to (Wq, k+1)$$

$$q = \begin{cases} \epsilon & \text{if } w = 0 \\ P(k) & \text{if } w = 1 \end{cases}$$

Halt states

Cyclic tag systems also have two halts:

1. Either the register clears out entirely, or...
2. ...the system enters an infinitely repeating loop of states

What an infinite loop usually would be is actually represented by the register growing to unbounded length!

Side note: These two are equivalent; see `https://cs.stackexchange.com/a/44931` for a sketch of why.

Are cyclic tag systems Turing-complete?

- Remember how we saw that tag systems are Turing-complete?
- The heavy lifting's already done! Just show that we can simulate tag systems with cyclic tag systems
- Thankfully, it's a simple construction: just one-hot encode the alphabet of a tag system
  - One-hot encoding is a common trick: if you have n categories, you map each category to one of the standard basis vectors in $\mathbb{R}^n$ to keep them both equally weighted and orthogonal to each other

Proving cyclic tag systems are Turing-complete

- Suppose we're given a tag system $(m, A = \{a_1, a_2, \dots a_n\}, P)$. We assume A is ordered; order it if not
- First, one-hot encode the letters: $a_1 = 100\dots, a_2 = 010\dots$, so on until $a_n = \dots 001$
- Now create the new production rules $\hat{P} : \mathbb{Z}_{(mn)} \to \{0,1\}^*$ as follows:
  - For $1 \leq k \leq n$, $\hat{P}(k) = P(a_k)$ in the one-hot encoded form. This removes the first letter w and concatenates $P(w)$ onto the register.
  - For $n < k \leq mn$, $\hat{P}(k) = \epsilon$. This deletes the remaining $m - 1$ symbols from the front of the register.
- Apply this to Cocke and Minsky's construction to simulate any Turing machine!

Section 4

Conclusion

So what was the point?

- I promised a proof that SQL is Turing-complete
- We've shown that all we need to do is implement a cyclic tag system within SQL: that's a proof that SQL is Turing-complete!
- `https://wiki.postgresql.org/wiki/Cyclic_Tag_System` implements a cyclic tag system in *only 28 lines*

So what was the point?

- Cyclic tag systems are far easier to reason with and implement than directly implementing Turing machines
- I've written a cyclic tag system simulator in Haskell: `https://github.com/papermango/cyts`
  - This is a proof that Haskell is Turing-complete!
  - It also lets you run any cyclic tag system of your choice on the command line - check it out if that sounds interesting
- For fun, go implement a cyclic tag system in any language of your choice - it won't take too long, but you'll have written something that can run DOOM!

Questions?

Google "hypercomputation."

— SCOTT AARONSON (2016)

Bibliography I

John Cocke and Marvin Minsky.

Universality of tag systems with p = 2.

J. ACM, 11(1):15–20, jan 1964.

B. Jack Copeland.

The Church-Turing Thesis.

In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2024 edition, 2024.

Liesbeth De Mol.

Tag systems and collatz-like functions.

Theoretical Computer Science, 390(1):92–101, 2008.