# Formalizing Complexity Theory

Christian Reitwiessner

chris477@zulip
crei@github

leaning in 2026, Berlin

2026-03-12

# Complexity Theory

Complexity theory studies the relation between various *resource bounds* for computation:

| | |
|---|---|
| **Time** | number of computation steps |
| **Space** | number of memory cells used |
| **Nondeterminism** | verifying a proposed solution |
| **Randomness** | |
| **Quantum** | |

Not so much about finding efficient algorithms for given problems, but rather if e.g. randomness or nondeterminism acutally provide an advantage and what the trade-offs between resource bounds are.

# The P vs. NP Question

$\mathbb{P}$ Yes-no-problems solvable in polynomial time.

$\mathbb{NP}$ Problems whose solutions can be *verified* in polynomial time.
$L \in \mathbb{NP} \iff \exists W \in \mathbb{P}, (x \in L \iff \exists^p y, (x, y) \in W)$

$$\mathbb{P} \stackrel{?}{=} \mathbb{NP}$$

- One of the seven Millennium Prize Problems.
- If $\mathbb{P} = \mathbb{NP}$: cryptography, optimisation, formalizing mathematics would be revolutionised.
- Widely believed: $\mathbb{P} \neq \mathbb{NP}$, but no proof in sight.

# Decades of Stagnation

- $\mathbb{P} \stackrel{?}{=} \mathbb{NP}$ only one example
- Most existing techniques cannot solve the hard problems in complexity ("natural proofs", "relativization", "algebrization")
- Solution has to use mathematical insights far removed from actual machines.
- Community largely gave up on a near-term resolution.

# Space-efficient computations

- Obvious: $\mathrm{DTIME}(T) \subseteq \mathrm{DSPACE}(T)$

# Space-efficient computations

- Obvious: $\mathrm{DTIME}(T) \subseteq \mathrm{DSPACE}(T)$
- Hopcroft–Paul–Valiant (1975): $\subseteq \mathrm{DSPACE}(T/\log T)$

# Space-efficient computations

- Obvious: $\mathrm{DTIME}(T) \subseteq \mathrm{DSPACE}(T)$
- Hopcroft–Paul–Valiant (1975): $\subseteq \mathrm{DSPACE}(T/\log T)$
- Ryan Williams (2025): $\subseteq \mathrm{DSPACE}\left(\sqrt{T} \cdot \mathrm{polylog}(T)\right)$

# Personal Perspective

- I learnt about Ryan Williams' result and Lean roughly at the same time.
- Let's formalize the result before someone else does!

# Personal Perspective

- I learnt about Ryan Williams' result and Lean roughly at the same time.
- Let's formalize the result before someone else does!
- I had no idea.

# Personal Perspective

- I learnt about Ryan Williams' result and Lean roughly at the same time.
- Let's formalize the result before someone else does!
- I had no idea.
- First obstacle: cannot even *state* the theorem.
- Complexity theory is not formalized at all in Mathlib / Cslib

# Personal Perspective

- I learnt about Ryan Williams' result and Lean roughly at the same time.
- Let's formalize the result before someone else does!
- I had no idea.
- First obstacle: cannot even *state* the theorem.
- Complexity theory is not formalized at all in Mathlib / Cslib

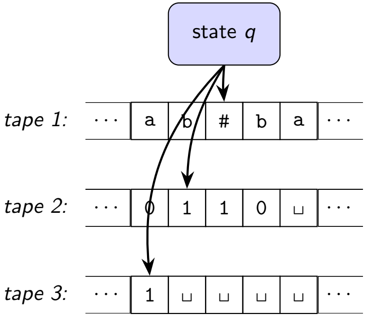
On to work!

# State of the Art

- Complexity theory generally seen as very difficult to formalize, several discussions in Zulip.
- Gäher and Kunze (2021) formalize computational complexity using Turing machines in Coq for the first time, down to the machine model.
- Their approach to use weak call-by-value $\lambda$-calculus does not work for logarithmic space.

# Turing Machine

- For computability, the computational model is largely irrelevant.
- For complexity, it is **not**—especially for space complexity.
- Williams' result is only known for Turing machines (local computation), and not for RAMs or other models.
- It gets tricky below linear space, but logarithmic space is one of the most important classes.



# Challenges: Why Is It So Hard?

- Many hand-wavy arguments in standard textbooks.
- Turing machines specified in prose.
- Frequent informal estimations that are hard to make rigorous.

CANYIELD = "On input $c_1$, $c_2$, and $t$:

1. If $t = 1$, then test directly whether $c_1 = c_2$ or whether $c_1$ yields $c_2$ in one step according to the rules of $N$. *Accept* if either test succeeds; *reject* if both fail.
2. If $t > 1$, then for each configuration $c_m$ of $N$ using space $f(n)$:
3. Run CANYIELD$(c_1, c_m, \frac{t}{2})$.
4. Run CANYIELD$(c_m, c_2, \frac{t}{2})$.
5. If steps 3 and 4 both accept, then *accept*.
6. If haven't yet accepted, *reject*."

# Current Progress

Formalized in `https://github.com/crei/cslib` on top of Cslib.

- Exploratory approach, focus on getting definitions right:
  - `sorry` proofs for the "obvious" but technically difficult parts
  - try to have good `simp` and `grind` proofs for composition results
- Elementary Turing machines and combinator tools.
- Currently trying to prove Savitch's Theorem, $\mathrm{NSPACE}(S) \subseteq \mathrm{DSPACE}(S^2)$, i.e. a space-efficient graph reachability algorithm

# TM DSL

The DSL evaluates to regular Turing machines but abstracts to stacks of words instead of tapes.
Similar to WHILE / LOOP programs, but with stacks instead of registers.

## Example: isZero machine

```
def isZero (i : Fin k) : MultiTapeTM k Symbol :=
  ite i (pop i ; push i []) (pop i ; push i [1])

theorem isZero_eval_list {i : Fin k}
  {tapes : Fin k -> List (List Symbol)} :
  (isZero i).eval_list tapes = .some (
    Function.update tapes i (
     (if (tapes i).headD [] = [] then
        [1]
      else
        []) :: (tapes i).tail)) := by
  simp [isZero]; grind
```

# Lessons Learnt so Far

- Simp lemmas work much better if you avoid assumptions about the tapes: Use `tm.eval tapes = ...` and use conditionals on the rhs.
- it is fine to have preconditions on the TM itself (i.e. always halts, etc).
- Requiring `[a, b, c].get.Injective` is much easier to deal with than $a \neq b \wedge b \neq c \wedge a \neq c$.
- run the simp linter!

# Upcoming: Computations on Structured Data

- For sub-linear space, we probably cannot stay with stacks but need to use arrays / inductive data structures.
- TMs on any lean type as long as it can be encoded into:
- ```
  inductive Data where
  | num :  Nat -> Data
  | list :  List Data -> Data
  ```
- Tapes are encodings of `Data`, tape head is `List Nat`, representing which branch was taken into the data structure
- Example: `((1)(2)((1)(2)))`
  ```

# SAT Verifier

```
def sat :=
  -- copy assignments to tape 1
  to_arg SatInput 2 0 ;
    copy Assignment 0 1 ;
    out_of_arg SatInput 2 0 ;
    to_arg SatInput 1 0
  -- for all clauses on tape 0 ...
  all_list Clause 0 2
    -- there is some literal ...
    (any_list Literal 0 2
      (case Literal 0
        -- that is positive and assigned "true", or
        (contains Var 0 1 2)
        -- negative and assigned "false"
        (contains Var 0 1 2 ; negate 2))) ;
  out_of_arg SatInput 1 0 ;
  erase Assignment 1
```

# Next Tasks

- Computations on structured data
- How to nicely map recursive algorithms to Turing machines?
- Find a good trade-off between precision and usability for space estimation.
- Create a working group inside Cslib? Many interested people (Bolton Bailey, Samuel Schlesinger, Shreyas Srinivas, ...)

# Thank you!

Slides:

https://crei.github.io/talks/2026_leaning_in_complexity/talk.pdf