Highlights from attending the Simons Workshop on Transformers as a Computational Model

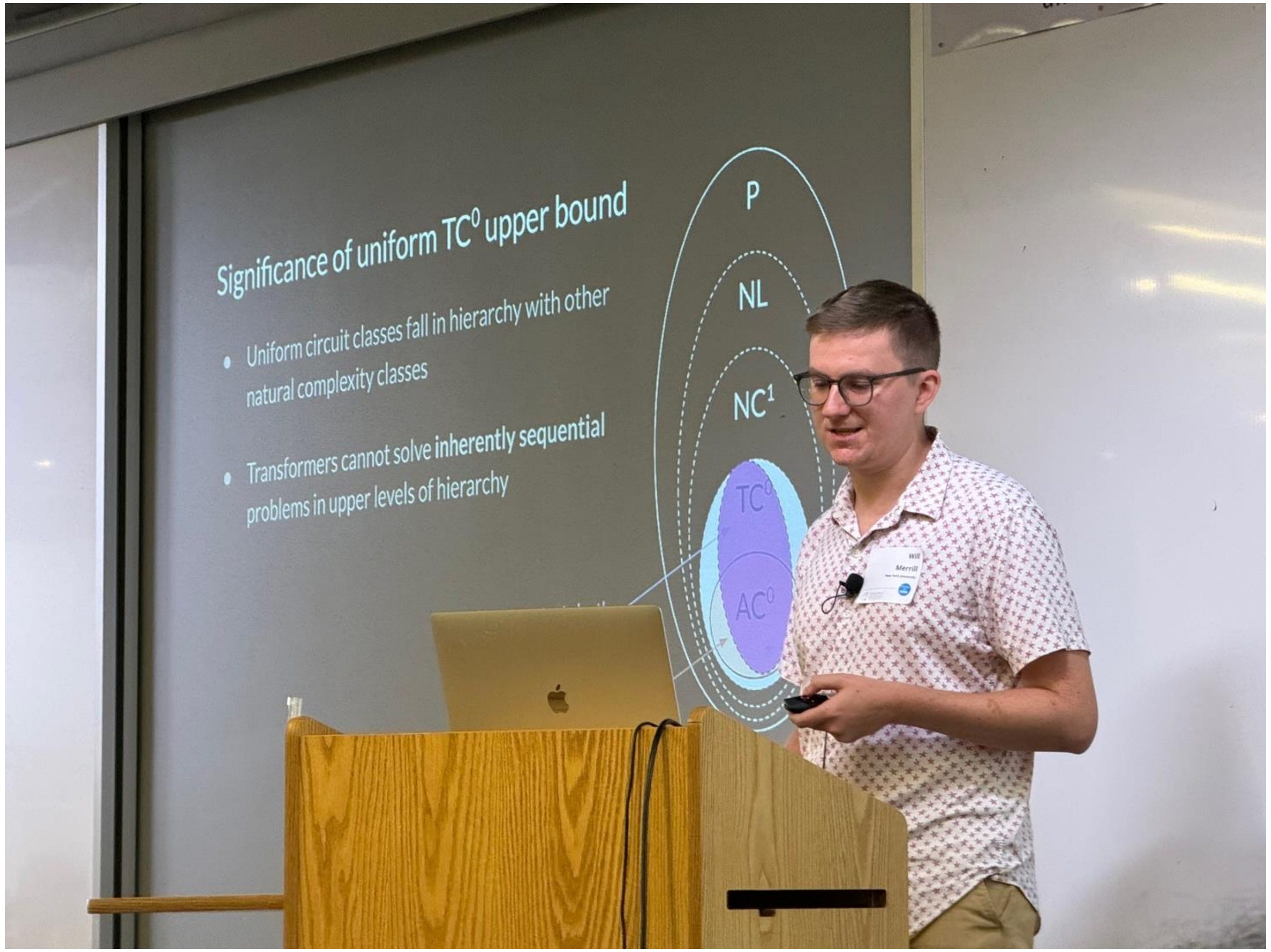
Breandan Considine

Three overall themes

- Formal language theory
  - Logical expressivity
  - Circuit complexity
  - Algebraic decomposition
- Mechanistic interpretability
  - RASP programs
  - Training dynamics
- Scaling up inference time compute
  - Recurrence and SSMs
  - Search and planning
  - Generator verifier gap

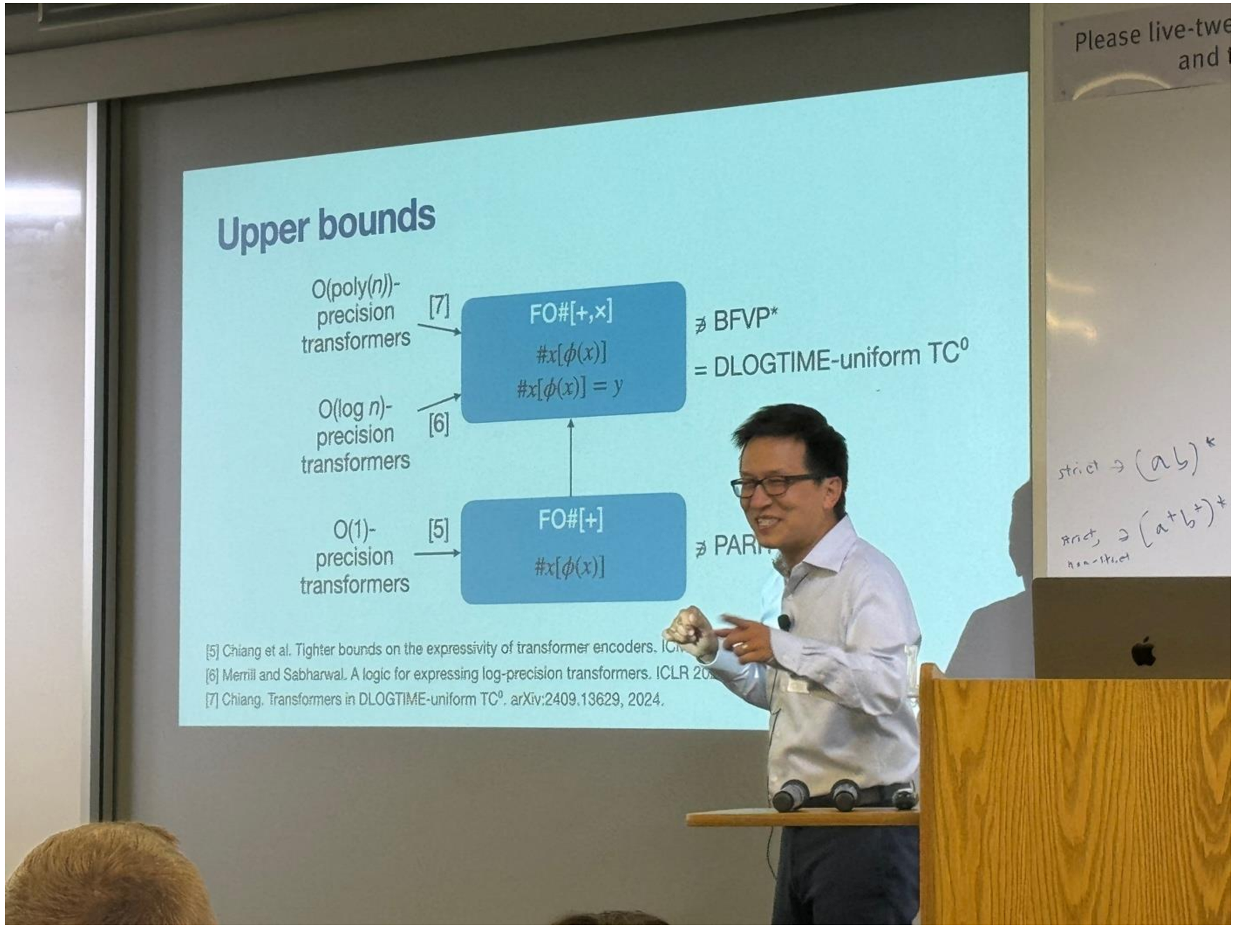
Circuit complexity (Merrill)

- Fixed precision circuits
- Can be studied as logical gates
- Transformer layers = circuit depth
- Computational parallelism vs. sequentiality



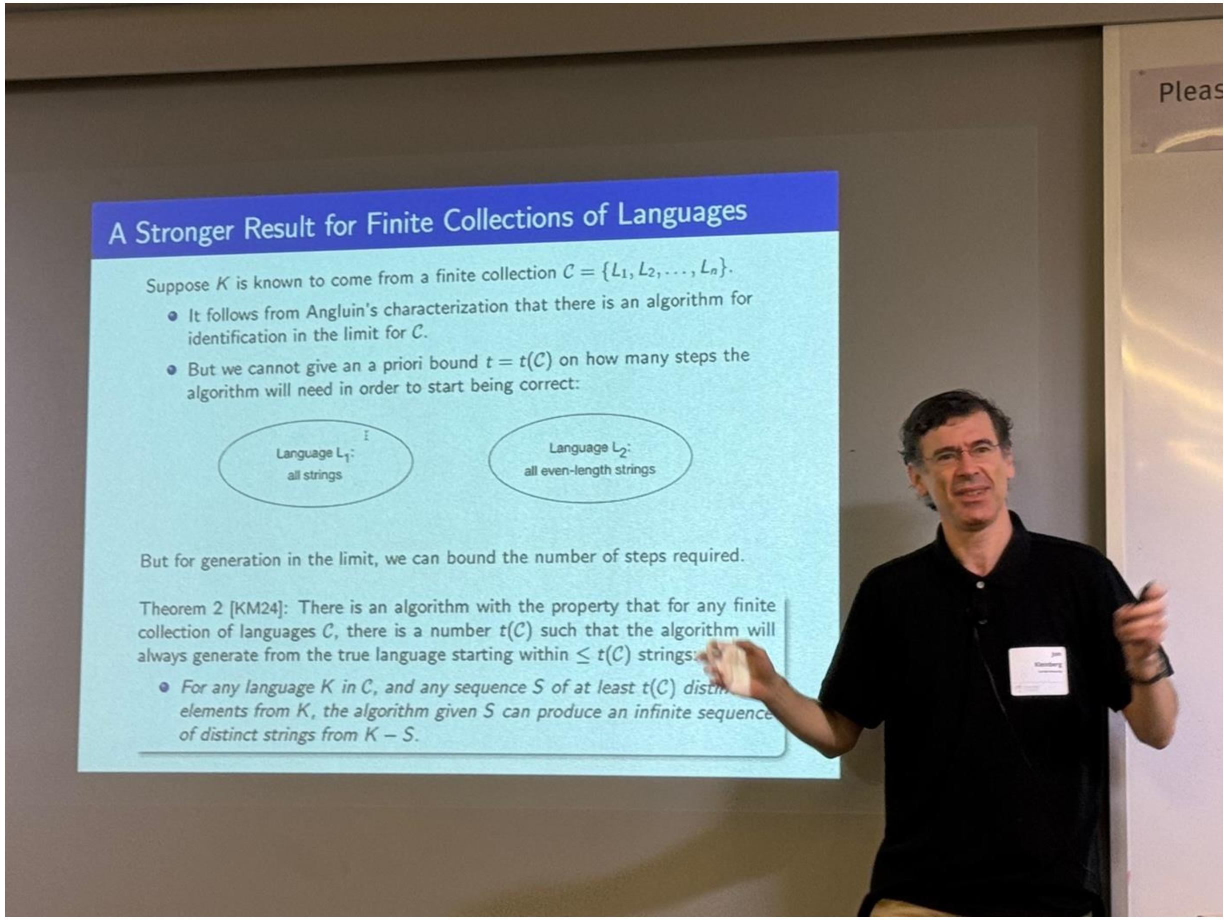
Logical expressivity (Chiang)

- Starts with linear logic
- Permits fine-grained control over resources
- Can be used to study transformer model expressivity
- Many fruitful connections with PL theory



Language generation in the limit (Kleinberg)

- Identifying the grammar for a language is not possible (Gold)
- But generating strings from the language is possible!
- Very elegant proof technique



Learning to reason with LLMs (Brown)

- Starts with his work designing poker bots
- (Re?)discovered the importance of planning
- Generator-verifier gap
- Cue O1, OpenAI’s newest LLM with SoTA reasoning capabilities

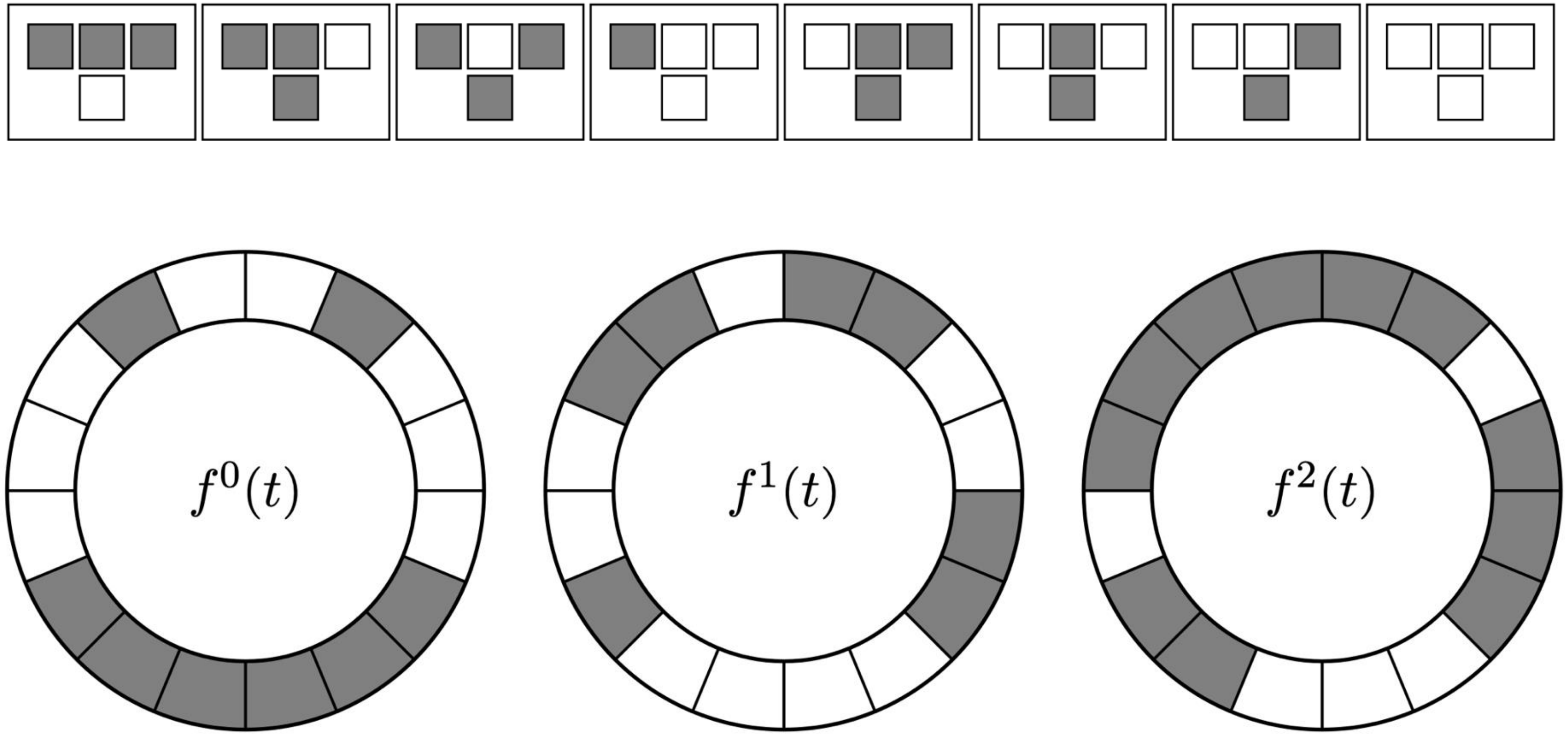
State space models (Gu)

- Combines attention with SSM layers
- S4, Mamba, H3 models
- Gives model variable time to “think”

$$x'(t) = \boldsymbol{A}x(t) + \boldsymbol{B}u(t)$$
$$y(t) = \boldsymbol{C}x(t) + \boldsymbol{D}u(t)$$

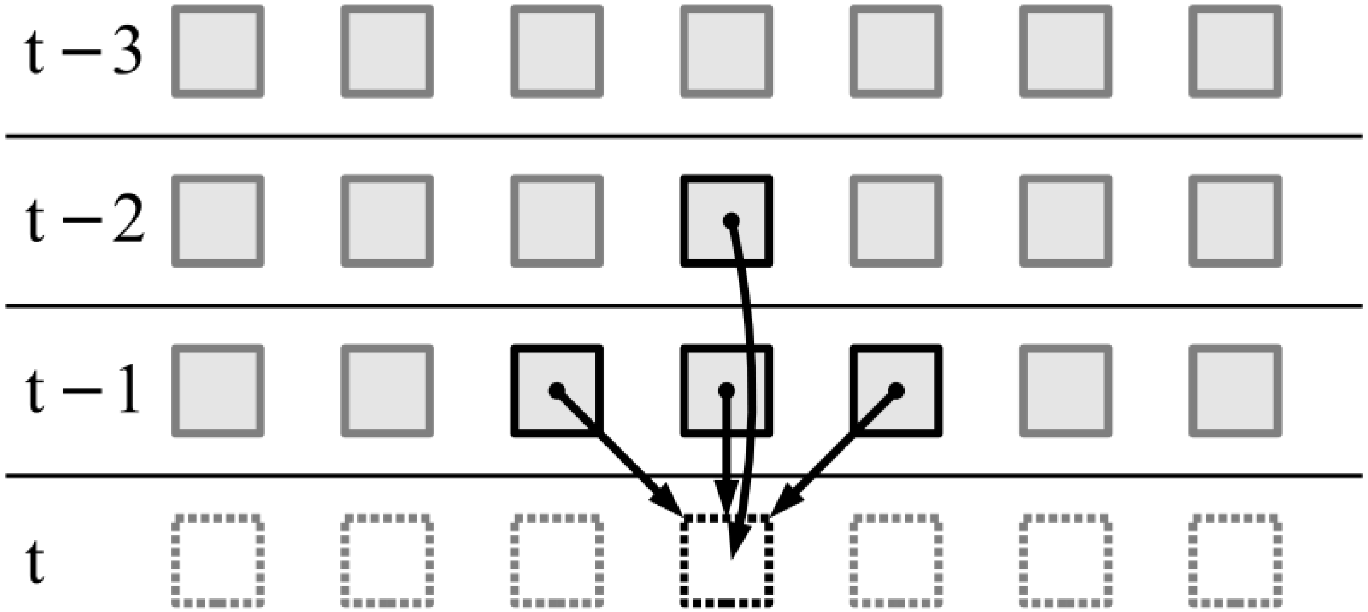
State space models (Breandan's primer)

- Consider the first-order discrete dynamical system



State space models (Breandan's primer)

- Now consider a second-order dynamical system



State space models (Breandan's primer)

- A very natural question to ask is the following:

  How do we represent a non-Markovian dynamical system in a Markovian way?

State space models (Breandan's primer)

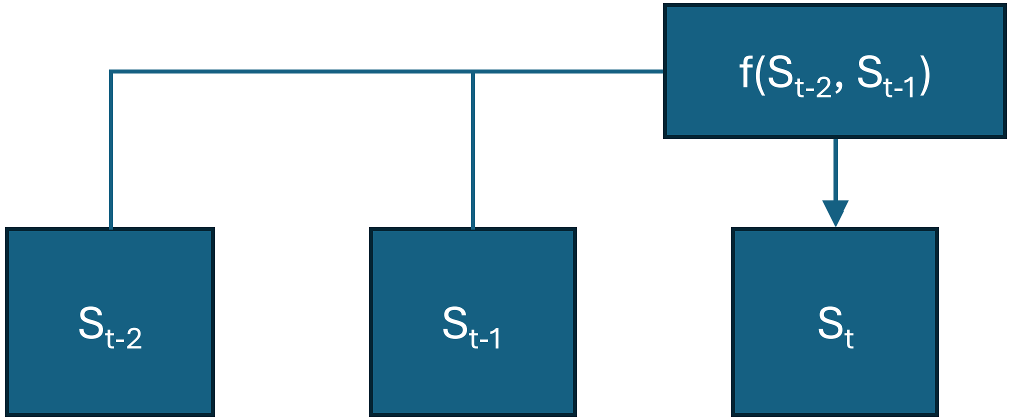
- A very natural question to ask is the following:

  How do we represent a non-Markovian dynamical system in a Markovian way?

- Using the state-space representation!
- Transforms higher-order dependence into a set of 1^{st}-order (i.e., Markovian) differential equations

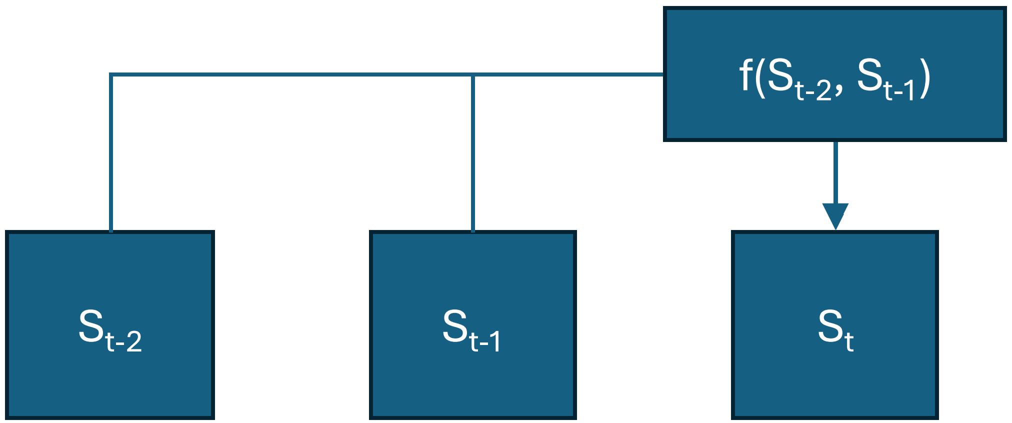
State space models (Breandan's primer)

We can take a second-order differential equation...

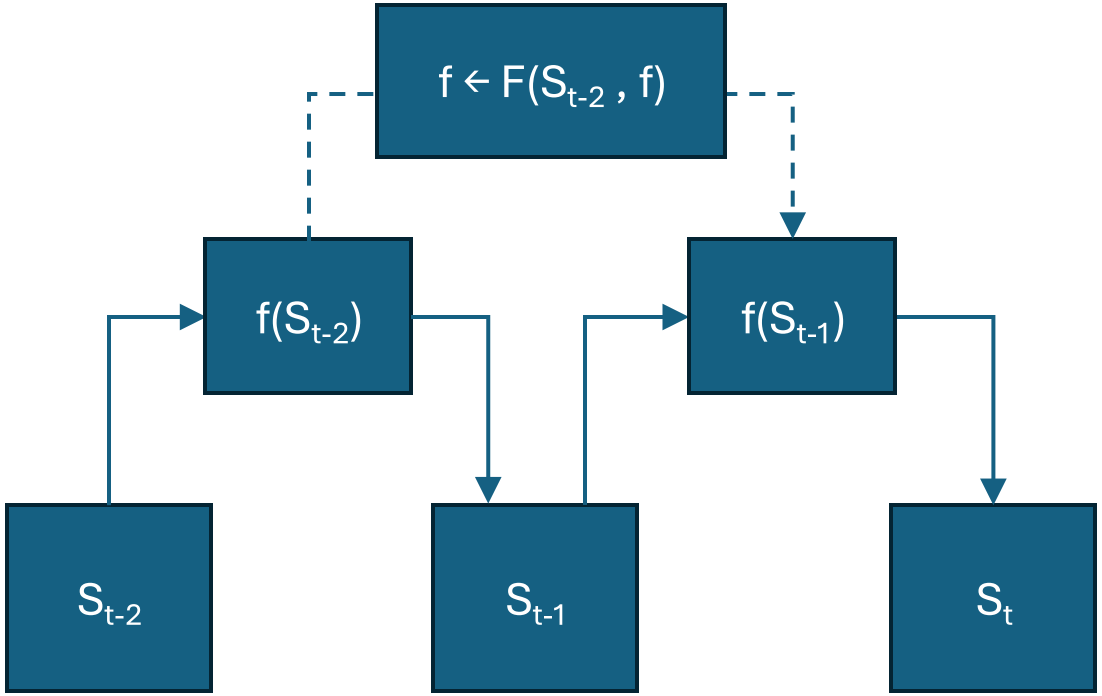


State space models (Breandan's primer)

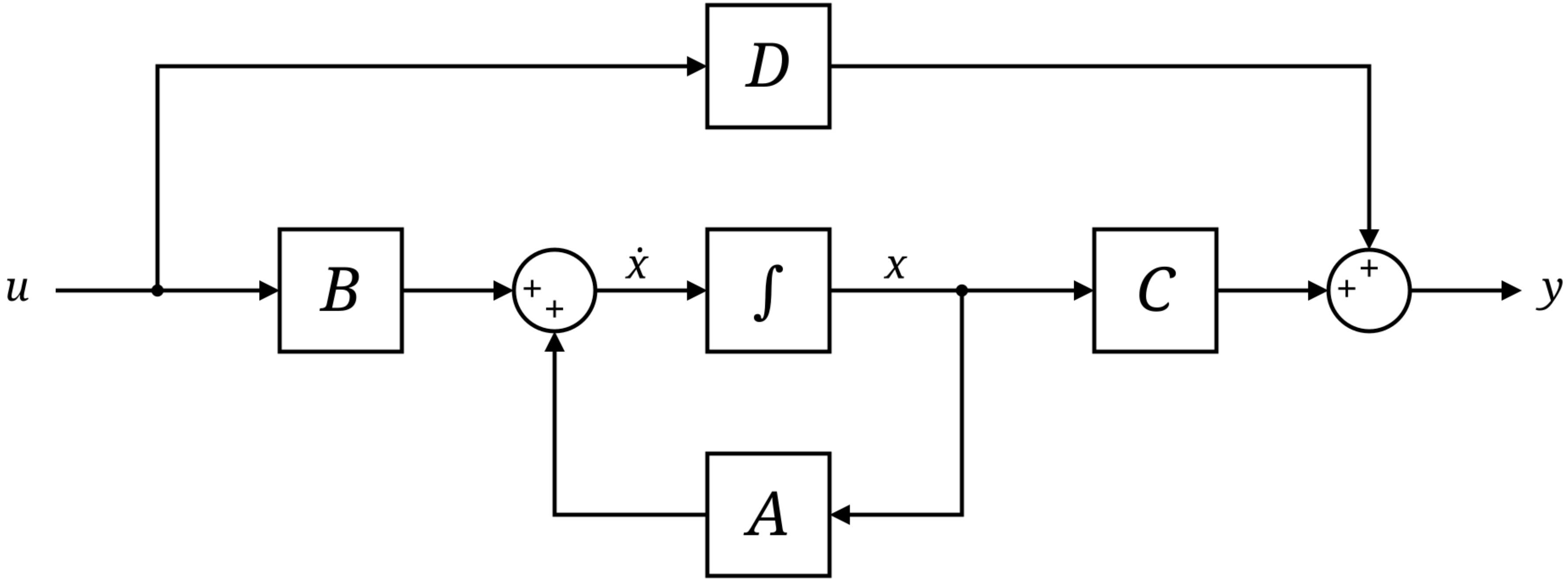
We can take a second-order differential equation…



…and make it stateful with an evolving transition function.



State space models (Breandan's primer)

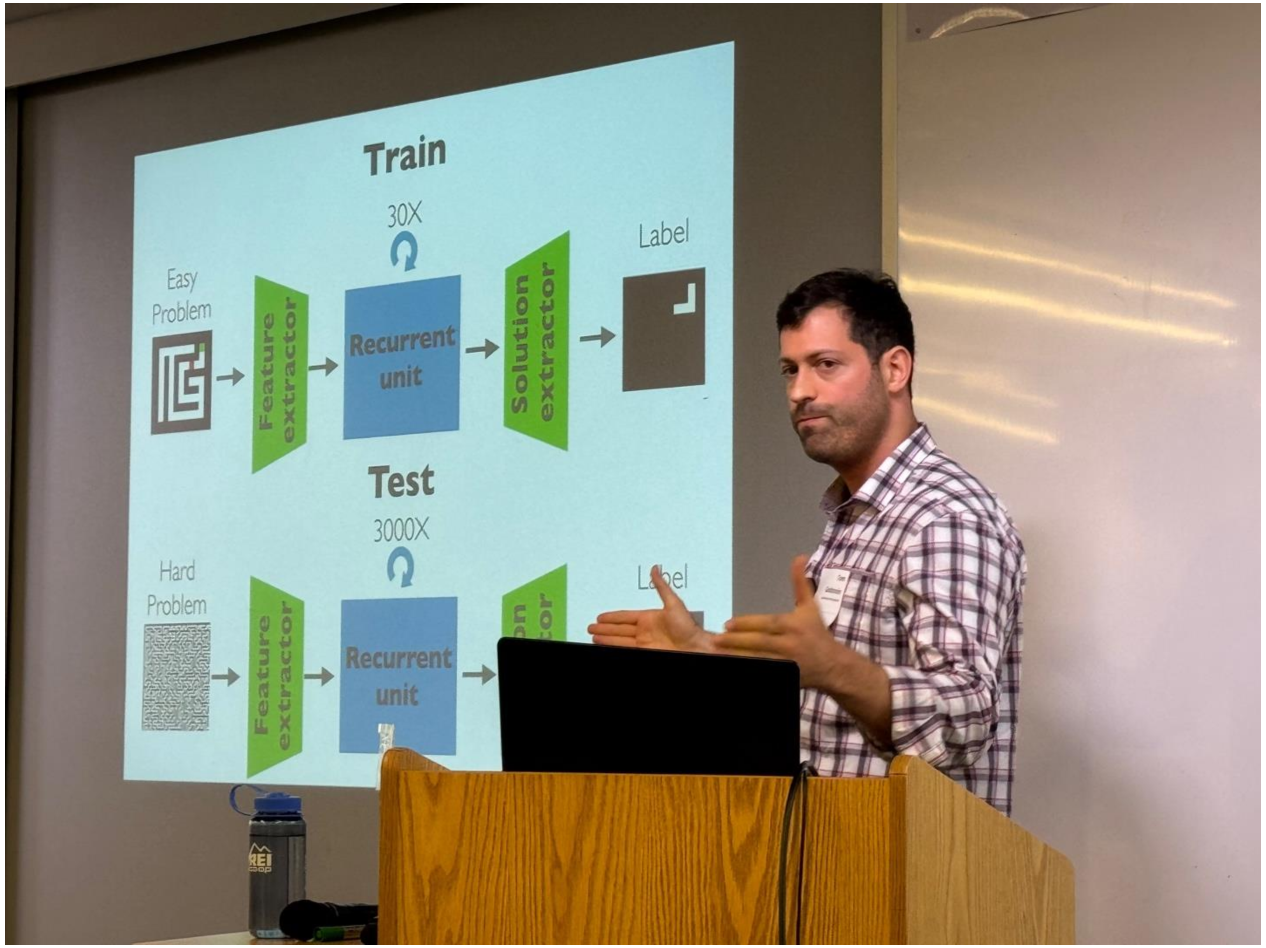


$$x'(t) = \boldsymbol{A}x(t) + \boldsymbol{B}u(t)$$

$$y(t) = \boldsymbol{C}x(t) + \boldsymbol{D}u(t)$$

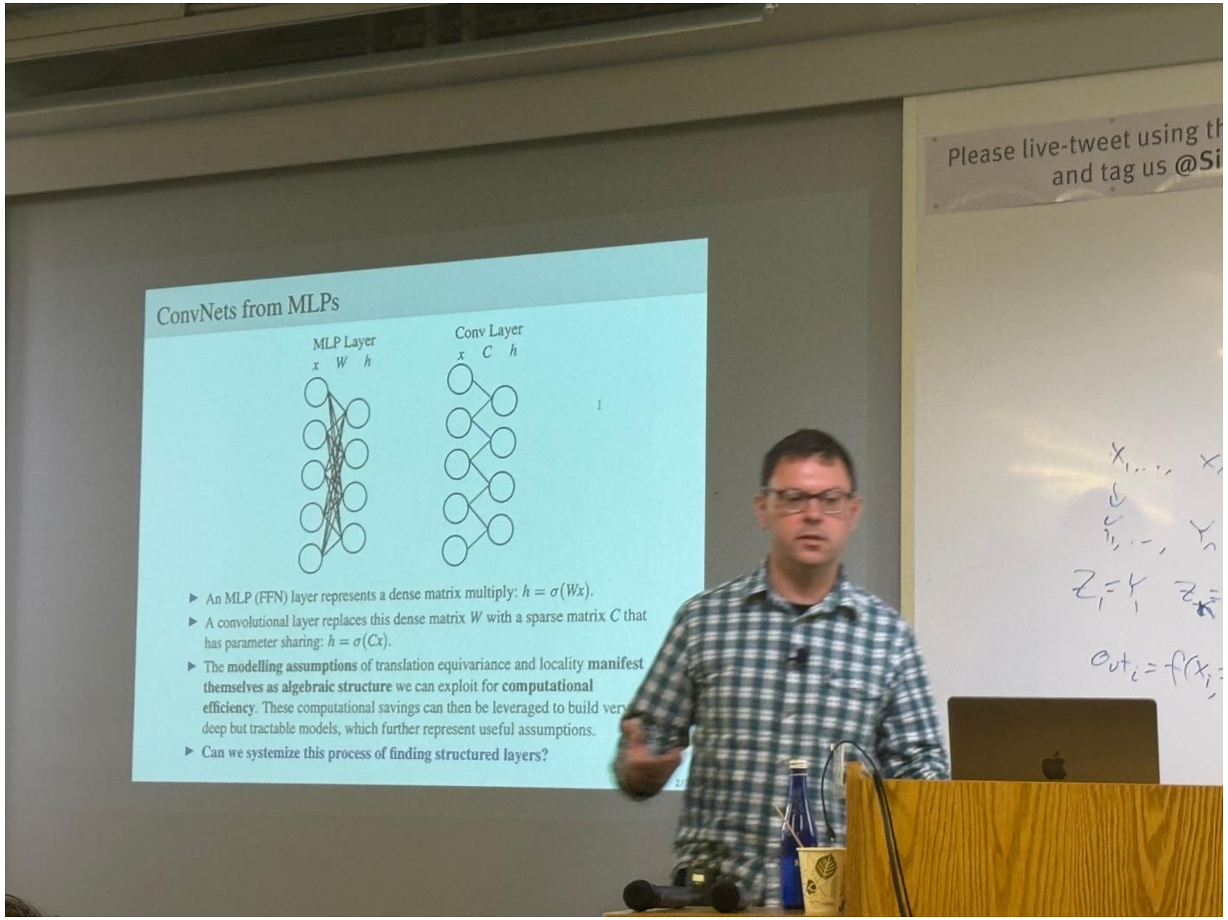
Weak to strong generalization (Goldstein)

- Train on easy reasoning tasks, then evaluate on harder instances of the same task
- Uses recurrent units
- Shows plenty of convincing results on mazes and sudoku

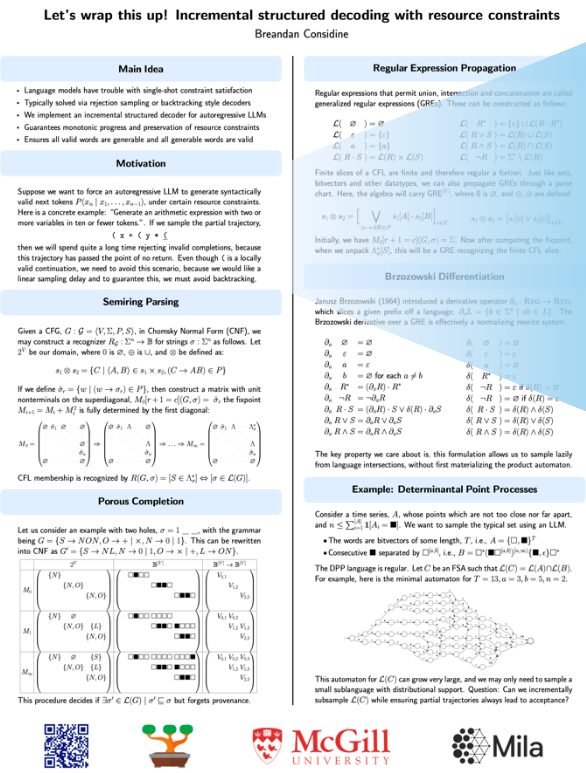


Machine learning is linear algebra (Wilson)

- Inductive biases for pretraining transformers
- Can leverage sparse structure from domain
- Continuous design space of structured matrices and tensors



My poster: Incremental structured decoding



Main Idea

- Language models have trouble with single-shot constraint satisfaction
- Typically solved via rejection sampling or backtracking style decoders
- We implement an incremental structured decoder for autoregressive LLMs
- Guarantees monotonic progress and preservation of resource constraints
- Ensures all valid words are generable and all generable words are valid

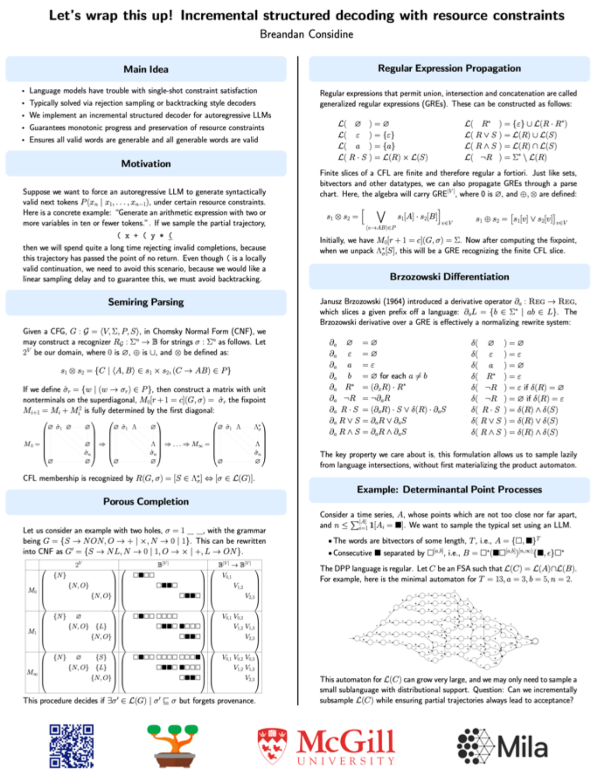
Motivation

Suppose we want to force an autoregressive LLM to generate syntactically valid next tokens $P(x_n \mid x_1, \ldots, x_{n-1})$, under certain resource constraints. Here is a concrete example: "Generate an arithmetic expression with two or more variables in ten or fewer tokens.". If we sample the partial trajectory,

$$(\ x \ + \ (\ y \ * \ \underline{(}$$

then we will spend quite a long time rejecting invalid completions, because this trajectory has passed the point of no return. Even though (is a locally valid continuation, we need to avoid this scenario, because we would like a linear sampling delay and to guarantee this, we must avoid backtracking.

My poster: Incremental structured decoding



Regular Expression Propagation

Regular expressions that permit union, intersection and concatenation are called generalized regular expressions (GREs). These can be constructed as follows:

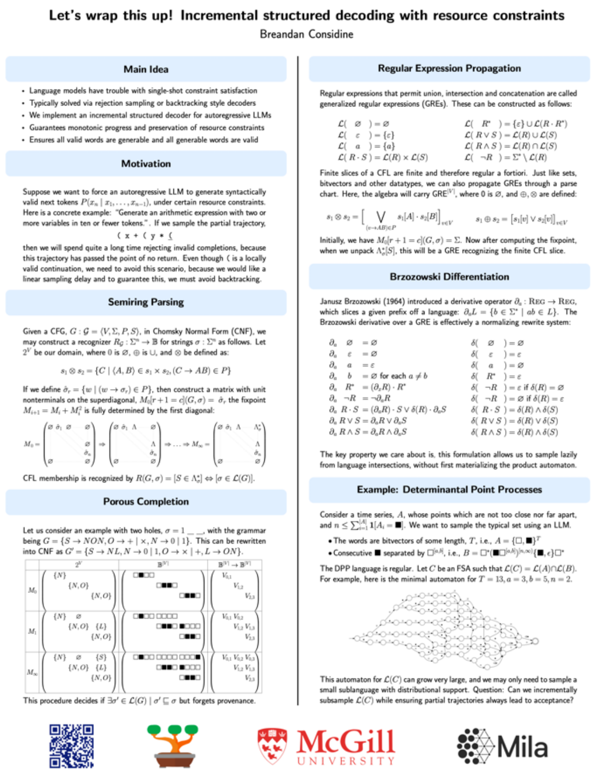
$$
\begin{aligned}
\mathcal{L}(\ \varnothing\ ) &= \varnothing & \mathcal{L}(\ R^*\ ) &= \{\varepsilon\} \cup \mathcal{L}(R \cdot R^*) \\
\mathcal{L}(\ \varepsilon\ ) &= \{\varepsilon\} & \mathcal{L}(\ R \vee S\ ) &= \mathcal{L}(R) \cup \mathcal{L}(S) \\
\mathcal{L}(\ a\ ) &= \{a\} & \mathcal{L}(\ R \wedge S\ ) &= \mathcal{L}(R) \cap \mathcal{L}(S) \\
\mathcal{L}(\ R \cdot S\ ) &= \mathcal{L}(R) \times \mathcal{L}(S) & \mathcal{L}(\ \neg R\ ) &= \Sigma^* \setminus \mathcal{L}(R)
\end{aligned}
$$

Finite slices of a CFL are finite and therefore regular a fortiori. Just like sets, bitvectors and other datatypes, we can also propagate GREs through a parse chart. Here, the algebra will carry $\text{GRE}^{|V|}$, where 0 is $\varnothing$, and $\oplus, \otimes$ are defined:

$$
s_1 \otimes s_2 = \Big[\bigvee_{(v \to AB) \in P} s_1[A] \cdot s_2[B] \Big]_{v \in V} \qquad s_1 \oplus s_2 = \big[s_1[v] \vee s_2[v] \big]_{v \in V}
$$

Initially, we have $M_0[r+1=c](G,\sigma) = \Sigma$. Now after computing the fixpoint, when we unpack $\Lambda_\sigma^*[S]$, this will be a GRE recognizing the finite CFL slice.

My poster: Incremental structured decoding



Brzozowski Differentiation

Janusz Brzozowski (1964) introduced a derivative operator $\partial_a : \textsc{Reg} \to \textsc{Reg}$, which slices a given prefix off a language: $\partial_a L = \{b \in \Sigma^* \mid ab \in L\}$. The Brzozowski derivative over a GRE is effectively a normalizing rewrite system:

$$
\begin{aligned}
\partial_a\ \varnothing &= \varnothing & \delta(\ \varnothing\ ) &= \varnothing \\
\partial_a\ \varepsilon &= \varnothing & \delta(\ \varepsilon\ ) &= \varepsilon \\
\partial_a\ a &= \varepsilon & \delta(\ a\ ) &= \varnothing \\
\partial_a\ b &= \varnothing \text{ for each } a \neq b & \delta(\ R^*\ ) &= \varepsilon \\
\partial_a\ R^* &= (\partial_x R) \cdot R^* & \delta(\ \neg R\ ) &= \varepsilon \text{ if } \delta(R) = \varnothing \\
\partial_a\ \neg R &= \neg \partial_a R & \delta(\ \neg R\ ) &= \varnothing \text{ if } \delta(R) = \varepsilon \\
\partial_a\ R \cdot S &= (\partial_a R) \cdot S \vee \delta(R) \cdot \partial_a S & \delta(\ R \cdot S\ ) &= \delta(R) \wedge \delta(S) \\
\partial_a\ R \vee S &= \partial_a R \vee \partial_a S & \delta(\ R \vee S\ ) &= \delta(R) \vee \delta(S) \\
\partial_a\ R \wedge S &= \partial_a R \wedge \partial_a S & \delta(\ R \wedge S\ ) &= \delta(R) \wedge \delta(S)
\end{aligned}
$$

The key property we care about is, this formulation allows us to sample lazily from language intersections, without first materializing the product automaton.

Visit to UC Berkeley