Synthesis and Verification of Transformer Programs

Hongjian Jiang[1] , **Matthew Hague**[2] , **Philipp Rümmer**[3,4] and **Anthony W. Lin**[1,5]
[1]University of Kaiserslautern-Landau, Germany
[2]Royal Holloway, University of London, England
[3]Uppsala University, Sweden
[4]University of Regensburg, Germany
[5]MPI-SWS, Germany
hongjian.jiang@rptu.de, matthew.hague@rhul.ac.uk, philipp.ruemmer@it.uu.se, awlin@mpi-sws.org

Abstract

C-RASP is a simple programming language that was recently shown to capture concepts expressible by transformers. In this paper, we develop new algorithmic techniques for automatically verifying C-RASPs. To this end, we establish a connection to the verification of synchronous dataflow programs in Lustre, which enables us to exploit state-of-the-art model checkers utilizing highly optimized SMT-solvers. Our second contribution addresses learning a C-RASP program in the first place. To this end, we provide a new algorithm for learning a C-RASP from examples using local search. We demonstrate efficacy of our implementation for benchmarks of C-RASPs in the literature, in particular in connection to the following applications: (1) transformer program minimization, and (2) constrained learning of transformer programs (based on a partial specification).

1 Introduction*

Transformers [Vaswani *et al.*, 2017] are the underlying neural network architectures behind Large Language Models (LLMs), using attention to aggregate information in the input sequence. Although transformers are expressive and at the same time efficiently parallelizable [Barceló *et al.*, 2024; Huang *et al.*, 2025; Yang and Chiang, 2024; Yang *et al.*, 2024; Strobl *et al.*, 2024], they are rather tricky to analyze. In particular, verifying transformers is undecidable [Sälzer *et al.*, 2025; Sälzer *et al.*, 2026]. Moreover, equipped with Chain of Thoughts (i.e. scratchpads for doing intermediate computation), they become Turing-complete [Pérez *et al.*, 2021; Merrill and Sabharwal, 2024; Jiang *et al.*, 2026b].

To better understand the behavior of transformers, researchers developed declarative languages which enjoy precisely defined semantics and at the same time faithfully capture the behavior of transformers. In 2021 Weiss et al. [Weiss *et al.*, 2021] proposed the influential RASP (Restricted Access Sequence Processing Language) paradigm. Essentially, RASP is a programming language that operates on sequences via operations like selection, aggregation, and simple position-wise arithmetic operations. Over the years the RASP paradigm has been instantiated and formally proven to model specific classes of transformers.

C-RASP (Counting RASP) [Yang and Chiang, 2024] is the most recent variant of RASP that has been shown (both theoretically and experimentally) by [Huang *et al.*, 2025] to capture real-world transformers. C-RASP is essentially a refinement of *Counting Linear Temporal Logic* [Barceló *et al.*, 2024], but without features that are not expressible by real-world transformers. More precisely, C-RASP allows a term of the form $\#\varphi$, which counts the number of positions j to the left of the current position i where the formula φ holds. In this way, the Dyck-1 language of balanced parentheses over the alphabet $\{\,[\,,\,]\,\}$ is expressible in C-RASP; more on this in Section 2. In addition, C-RASP can express languages like MAJORITY (i.e. the input string contains more a's than b's), but cannot express the flip-flop language [Liu *et al.*, 2023] — essentially, the regular language $\Sigma^* b e^*$, for some letters b, e in the alphabet Σ — as well as PARITY (i.e. the number of a's in the input string is even), both of which are not trainable by transformers. [Huang *et al.*, 2025] showed that C-RASP is captured by length-generalizable transformers (in that a transformers can learn a target language, given a finite training dataset). In particular, C-RASP is conjectured by [Huang *et al.*, 2025] to capture *all* length-generalizable transformer languages; this is a formal version of the so-called *RASP-L conjecture* [Zhou *et al.*, 2024].

Despite tight connections between C-RASP and transformers, the role of C-RASP has mostly been to help understand the expressivity of transformers. Here, we study the role of C-RASP as an *interpretable surrogate model* of transformers. "Interpretability" is a general concept in AI (e.g. see [Krishnan *et al.*, 2024]), but for C-RASP (which has a formally defined semantics) it is best understood as the existence of an automatic method to analyze C-RASP programs. In particular, this is consistent with the term "interpretability" and "explainability" in *FXAI* (Formal eXplainable AI) [Marques-Silva and Ignatiev, 2023] and investigations in the formal aspects of AI (e.g. [Okudono *et al.*, 2020; Barceló *et al.*, 2020; Weiss *et al.*, 2024; Hong *et al.*, 2025; Arenas *et al.*, 2021]). To this end, there are two main challenges. The first concerns *automatic verification*, since hith-

*An extended version of the paper, including appendices, is available at arXiv:2602.16473.

erto no automatic method to analyze C-RASP exists. In fact, checking whether a given C-RASP recognizes a trivial language is undecidable [Yang *et al.*, 2026]. The second concerns *synthesis*, since there is no known method to actually learn a C-RASP program (from examples, or otherwise).

Contributions. We develop the *first techniques* that address both the verification and synthesis of C-RASP programs. We have implemented them and shown their efficacy on an assortment of benchmarks from the literature [Huang *et al.*, 2025; Yang *et al.*, 2025], in relation to the following applications: (1) transformer program minimization, and (2) constrained learning of transformer programs.

Our first contribution is the *first automatic method for verifying C-RASP programs*. In particular, we provide a reduction to the verification of *Lustre*, which is a synchronous datastream language with precise syntax and semantics. Since Lustre is supported by highly optimized model checkers (e.g. Kind2 Model Checker [Champion *et al.*, 2016] with plugins to fast SMT solvers like CVC5 and Z3), we obtain an automatic method for verifying C-RASP programs against properties like language equivalence, inclusion, and emptiness.

Our second contribution is the *first automatic method for synthesizing C-RASP programs* from positive and negative examples. The method exploits local search in the form of simulated annealing. This is orthogonal to stochastic gradient descent for training transformers, which yields no implementable translation since the current translations (e.g. see [Yang and Chiang, 2024; Yang *et al.*, 2025]) work only from a specific class of idealized transformers.

We have implemented both methods and demonstrate their efficacy against two applications: (i) C-RASP minimization (given a C-RASP, construct a smaller C-RASP) and (ii) constrained learning of C-RASP from a partial specification. In particular, we have tested our verifier against existing C-RASP benchmarks in the literature (e.g. [Huang *et al.*, 2025; Yang *et al.*, 2025]), and show highly promising experimental results (see Table 1). In summary, our tool uses for both synthesis and verification of each benchmark a total of at most 300 seconds. In contrast, transformer training (e.g., GPT2) using the HuggingFace `transformer` library in PyTorch for some of these benchmarks alone could take *up to several hours* to achieve the correct language!

Organization. We define C-RASP and provide examples in Section 2. In Section 3, we define Lustre and a reduction from C-RASP to Lustre. Our synthesis algorithm is in Section 4. Experimental results and applications are in Section 5. We conclude with discussion and related work in Section 6.

We refer the reader to the technical report [Jiang *et al.*, 2026a] and the artifact [Hongjian *et al.*, 2026] for implementation and benchmark details.

2 Preliminaries

Before giving the full definition of C-RASP, we begin with an example. Let Σ be a finite nonempty alphabet and Σ^+ denote nonempty words[†] $a_1 \dots a_n$. Let $\top$ and $\bot$ denote the Boolean values true and false. C-RASP programs are given by a sequence of rules that are evaluated over each position of an input word from Σ^+. Rules may evaluate to Boolean or integer values, referred to as Boolean and counting rules, respectively. The C-RASP program below recognizes the Dyck(-1) language; that is, the language of words over $\Sigma = \{[,]\}$ where each [is matched by an] in a well-nested fashion. For example [], [[]], and [[][]] are Dyck words, []] and][are not.

Example 1. *The following C-RASP program contains four rules and recognizes Dyck words. It is explained below.*

$$\begin{array}{llll} C_{[} & := & \#[& \qquad C_{]} := \#] \\ V & := & C_{[} < C_{]} & \qquad D := \#V = 0 \wedge C_{[} = C_{]} \end{array}$$

The counting rules rule $C_{[}$ and $C_{]}$ count the number of [and] characters respectively. We use the Boolean rule V to detect violations of well-nestedness. That is, V is true at a position of the input word if the number of [characters is less than the number of] characters. Finally, D is true only at the end of a Dyck word. It requires that the word contains no violations of the well-nested property and ultimately has the same number of [and]. The table below shows the full execution of the program over the word [[][]]][which is not a Dyck word.

	[	[	]	[	]	]	]	[	]
$C_{[}$	1	2	2	3	3	3	3	4	4
$C_{]}$	0	0	1	1	2	3	4	4	5
V	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
D	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$

A word is accepted if the value of the last rule (D) is $\top$ at the final position of the word.

In the sequel, we follow closely the definition of C-RASP from [Huang *et al.*, 2025], which also allows local and periodic positional encodings.

Definition 2.1 (C-RASP)**.** *A* C-RASP *program P over a finite alphabet Σ is a finite sequence of rules $P = (R_1, \dots, R_k)$ where each R_i is either a* Boolean rule *or a* Count rule *of the form $B := BExp$ or $C := CExp$ respectively. B and C are the names of the rules. To recognize languages, R_k is a Boolean rule that determines acceptance.*

Definition 2.2 (Boolean Expressions)**.** *The syntax of Boolean expressions is given by the following grammar, in which $a \in \Sigma$ is an atomic letter predicate, B is the name of a Boolean rule, $\diamond \in \{\wedge, \vee\}$ denotes Boolean conjunction and disjunction, $\bowtie \in \{=, \leq, <\}$ denotes a comparison between counting expressions, and $o < m \in \mathbb{N}$:*

$$\begin{aligned} BExp ::= \top \mid \bot \mid a \mid B \mid \neg\, BExp \mid BExp \ \diamond \ BExp \\ \mid CExp \ \bowtie \ CExp \mid m\%o. \end{aligned}$$

The semantics of Boolean expressions is as expected; in particular, B evaluates to the value of the referenced rule and $m\%o$ is true at positions j satisfying $j \bmod m = o$.

Definition 2.3 (Counting Expressions)**.** *The syntax of counting expressions is defined inductively as follows, where $k \in \mathbb{N}$, C is the name of a counting rule, and $r_s < r_e \in \mathbb{N}$:*

$$\begin{aligned} CExp ::= k \mid C \mid \#(BExp) \mid \#_{r_s, r_e}(BExp) \\ \mid CExp + CExp \mid CExp - CExp \\ \mid \min(CExp, CExp) \mid \max(CExp, CExp) \\ \mid BExp \ ?\ CExp : CExp. \end{aligned}$$

[†]We use "words" and "strings" synonymously in the paper, as as is standard in formal language theory.

Here, at position j, $\#(BExp)$ denotes the number of positions satisfying the Boolean expression at or before position j. The expression $\#_{r_s,r_e}(BExp)$ is a variant that only counts positions occurring between r_s and r_e positions before the current position. The conditional expression $e_b \ ? \ e_1 \ : \ e_2$ evaluates to e_1 if the Boolean expression e_b holds and to e_2 otherwise. Constants $k \in \mathbb{N}$ denote fixed integer values and C evaluates to the value of the referenced rule.

Counting expressions thus provide numerical summaries of Boolean rules, which can be compared in Boolean expressions to form complex logical conditions. The expressions $m\%o$ and $\#_{r_s,r_e}$ are extensions of C-RASP to handle LOCAL and PERIODIC positional predicates [Huang *et al.*, 2025].

C-RASP programs are evaluated over words. Given a word $w = a_1 \dots a_n \in \Sigma^+$ we inductively define the value of expressions. Let each rule be of the form $R_i = V := e_V$, that is, e_V is the expression associated to the rule named V. We define $\nu_w : (BExp \times \{1, \dots, n\} \to \{\top, \bot\}) \cup (CExp \times \{1, \dots, n\} \to \mathbb{Z})$ to be the value of the expression at position j in the word. Here we show only a selection of interesting cases, the full definition is given in the extended version.

$$\begin{aligned}
\nu_w(a, j) &:= \top \text{ if } a = a_j \text{ and } \bot \text{ otherwise} \\
\nu_w(B, j) &:= \nu_w(e_B, j) \\
\nu_w(m\%o, j) &:= j \bmod m = o \\
\nu_w(C, j) &:= \nu_w(e_C, j) \\
\nu_w(\#e, j) &:= \sum_{i \le j} \nu_w(e \ ? \ 1 : 0, i) \\
\nu_w(\#_{r_s,r_e} e, j) &:= \sum_{j - r_e \le i \le j - r_s} \nu_w(e \ ? \ 1 : 0, i) \\
\nu_w(e_1 \ ? \ e_2 : e_3, j) &:= \begin{cases} \nu_w(e_2, j) & \text{if } \nu_w(e_1, j) = \top \\ \nu_w(e_3, j) & \text{otherwise.} \end{cases}
\end{aligned}$$

A word $w = a_1 \dots a_n \in \Sigma^+$ is accepted by a C-RASP program $(R_1, \dots, R_k)$ if $\nu_w(B, n) = \top$ where $R_k = B := e$. [Remark: for simplicity, we do not allow an empty input, but this is easy to handle, e.g., by allowing an end-of-string symbol (see [Yang and Chiang, 2024]).]

3 Verification of C-RASP Programs

We verify properties of C-RASP programs via a translation to the synthronous dataflow language Lustre [Caspi *et al.*, 1987], which is tailored to implementing embedded systems. Properties of Lustre programs can be checked by highly optimized tools such as Kind2 [Champion *et al.*, 2016].

3.1 Lustre

We do not present Lustre in full, but show a fragment that is sufficient to show how C-RASP can be simulated. We will begin with an example to show how Lustre programs can recognize Dyck words. The program operates in a similar way to the C-RASP program of Example 1.

In the full definition, a Lustre program contains a set $N_1, \dots, N_n$ of nodes. For simplicity of presentation we will use a single node. Intuitively, nodes can be used to define functions that are reusable and aid readability. Each node has input, output, and local variables. Our node will have one input variable (the input word) and no output variables. We use two types: BOOL and INT. Each variable represents an infinite stream of values of the declared type and will broadly correspond to a single C-RASP rule.

Local and output variables are defined using standard mathematical and Boolean combinations of variables and constants. For example $X := \neg Y$ means that at position i in the stream, X takes the negation of the value of Y at position i. The pre operator accesses the value of a variable at the previous position. The $\to$ operator can be used to replace the initial value of a stream. A typical example of these two operators is $X := 1 \to \mathrm{pre}(X) + 1$, which defines the sequence $1, 2, 3, \dots$ where 1 replaces the initial (undefined) value of $\mathrm{pre}(X) + 1$. Kind2 supports a check keyword to verify that a given Boolean expression evaluates to true at all positions.

Example 2. *The following Lustre program recognizes Dyck words. It has an input variable I of type* INT *and local variables N_l, C_l, N_r, C_r, N_V, and C_V of type* INT *and V and D of type* BOOL*. D is designated as an output variable. For this example we will make the simplifying assumption that I only takes integer values* 0 *or* 1 *(corresponding to* [*and*]*). Our full encoding will need to enforce this assumption and use end of stream markers. The equations defining each variable are shown below and then explained.*

```
node DyckCounters(I: int) returns (D: bool);
var Nl, Nr, Cl, Cr, Nv, Cv : int; V : bool;
let
  Nl = if I = 0 then 1 else 0;
  Nr = if I = 1 then 1 else 0;
  Cl = (0 -> pre(Cl)) + Nl;
  Cr = (0 -> pre(Cr)) + Nr;
  V  = Cl < Cr;
  Nv = if V then 1 else 0;
  Cv = (0 -> pre(Cv)) + Nv;
  D  = (Cv = 0) and (Cl = Cr);
tel
```

The equations N_l, N_r and N_V convert Boolean values into an integer. C_l, C_r then count how many times the input was [*or*]*. V is true whenever the property $C_l \ge C_r$ is violated and C_V counts how many times a violation has occurred. Finally D is true at all positions that have not seen any violations in the past and are currently well-balanced. The table below shows an execution over the infinite stream* $00101000\dots$ *that represents* $[[][]]]\dots$*. The value of D is $\top$ when the prefix of the stream is a Dyck word.*

	0	0	1	0	1	1	1	…
N_l	1	1	0	1	0	0	0	…
N_r	0	0	1	0	1	1	1	…
C_l	1	2	2	3	3	3	3	…
C_r	0	0	1	1	2	3	4	…
V	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	…
N_V	0	0	0	0	0	0	1	…
C_V	0	0	0	0	0	0	1	…
D	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	…

Adding a check *statement* check $\neg(C_r = 3) \vee D$ *would require the property that whenever three*] *characters have been*

seen, then the word so far is a Dyck word. This property is true of the stream above, but not true of all input streams.

Definition 3.1 (Lustre (fragment))**.** *A Lustre program in our fragment is a node with: one input variable I of type* INT*; local variables $B_1, \ldots, B_{m_b}$ of type* BOOL*; local variables $C_1, \ldots, C_{m_c}$ of type* INT*; one equation for each local variable; and a* check *statement.*

Let $\mathcal{B} = \{B_1, \ldots, B_{m_b}\}$ and $\mathcal{C} = \{I, C_1, \ldots, C_{m_c}\}$ denote the Boolean and integer variables. Boolean variables are defined using equations $B := LBExp$ and integer variables with $C := LIExp$ where

$$\begin{aligned} LBExp ::= & \top \mid \bot \mid B \mid \neg LBExp \\ & \mid LBExp \diamond_b LBExp \mid LIExp \bowtie_b LIExp \\ & \mid \mathrm{pre}(LBExp) \mid LBExp \to LBExp \end{aligned}$$

with $B \in \mathcal{B}$, $\diamond_b \in \{\wedge, \vee\}$ and $\bowtie_b \in \{<, =, \leq\}$ and

$$\begin{aligned} LIExp ::= & k \mid C \mid LIExp \diamond_i LIExp \\ & \mid \text{if } LBExp \text{ then } LIExp \text{ else } LIExp \\ & \mid \mathrm{pre}(LIExp) \mid LIExp \to LIExp \end{aligned}$$

with $C \in \mathcal{C}$, $\diamond_b \in \{+, -, \mathrm{mod}\}$ and $k \in \mathbb{Z}$. The check *statement is of the form* check $LBExp$*.*

Variables hold infinite streams of integers or Booleans. Let $\nu : (\mathcal{B} \times \mathbb{N} \to \{\top, \bot\}) \cup (\mathcal{C} \times \mathbb{N} \to \mathbb{Z})$ be a valuation of the Boolean and integer variables. That is $\nu(V, i)$ is the value of the ith position of the stream assigned to the variable V. We generalize ν to expressions e. Interesting cases are given below and full generalization refers to the extended version.

$$\nu(\mathrm{pre}(e), i) = \begin{cases} \nu(e, i-1) & \text{when } i > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\nu(e_1 \to e_2, i) = \begin{cases} \nu(e_1, i) & \text{if } i = 0 \\ \nu(e_2, i) & \text{otherwise} \end{cases}$$

$$\nu(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, i) = \begin{cases} \nu(e_2, i) & \text{if } \nu(e_1, i) = \top \\ \nu(e_3, i) & \text{otherwise} \end{cases}$$

Each equation $V := e$ means V takes a value such that $\nu(V, i) = \nu(e, i)$ for all i. An assertion check e is violated if there exists some assignment ν consistent with the equations of the program such that for some i we have $\nu(e, i) = \bot$. We say a Lustre program is *valid* if no check statement is violated by any value of the input variable I.

While the semantics is largely straightforward, it is worth considering some examples of pre and $\to$. An expression $1 \to 2$ defines a stream $1, 2, 2, 2, \ldots$. However, notice that the expression $1 \to (1 \to 2)$ also defines $1, 2, 2, 2, \ldots$ as $\to$ only changes the first value. To define a stream $1, 2, 3, 4, 4, 4, \ldots$ we use $1 \to \mathrm{pre}(2 \to \mathrm{pre}(3 \to 4))$.

3.2 C-RASP to Lustre

We encode a C-RASP program $(R_1, \ldots, R_k)$ over an alphabet Σ into Lustre as follows. An outline of the translation can be seen by comparing Example 1 and Example 2. Consider the C-RASP expression $\#V$. This is simulated in Lustre by introducing new variables N_V and C_V with

$$\begin{aligned} N_V &:= \text{if } V \text{ then } 1 \text{ else } 0 \\ C_V &:= N_V \to pre(C_V) + N_V. \end{aligned}$$

The expression $\#V$ is then translated directly to C_V. Most other C-RASP constructs allow almost direct translations, with some additional subtlety for the counting operators.

We also need to bridge the gap between C-RASP operating over finite words and Lustre over infinite streams of integers. The translations are given below.

Representing the Input Word

We introduce two fresh symbols $\#$ (end of stream) and Ω (eternity). Let $\Sigma' = \Sigma \cup \{\#, \Omega\}$. Choose any injective map $\iota : \Sigma' \to \mathbb{Z}$. A finite word $a_1 \ldots a_n$ will be encoded by an infinite stream $\iota(a_1) \ldots \iota(a_n)\iota(\#)\iota(\Omega)\iota(\Omega)\iota(\Omega) \ldots$.

For convenience, let $I \in S$ for a set $S \subseteq \Sigma'$ denote $\bigvee_{a \in S} I = \iota(a)$. Similarly $I = a$ denotes $I = \iota(a)$.

We introduce a new Boolean Lustre variable B_I and assign it the following Boolean expression that checks at each position i if I is a correct input word. It asserts that each position encodes a character. Then, it asserts that the first symbol is not Ω and Ω is the only character seen after $\#$. Let $R_I =$

$$B_I := I \in \Sigma' \wedge \neg(I = \Omega) \to (\neg \mathrm{pre}(I \in \{\#, \Omega\}) \vee I = \Omega).$$

We introduce a second variable $B_{\hat{I}}$ to track whether B_I was false at any point in the word. Let $R_{\hat{I}} = B_{\hat{I}} := B_I \to (B_I \wedge \mathrm{pre}(B_{\hat{I}}))$. Finally, let $\mathcal{R}_I = \{R_I, R_{\hat{I}}\}$.

Translating C-RASP Rules

Let $(R_1, \ldots, R_k)$ be a C-RASP program. For each Boolean rule $R_i = B := e$, introduce a Lustre variable V_B that is of type BOOL. Similarly, introduce a variable V_C of type INT for each counting rule $R_i = C := e$. For each C-RASP rule $R_i = X := e$ we define $T(R_i)$ to be a set of equations including the rule $V_X := T(e)$ and possibly some auxiliary equations. We define $T(e)$ in full in the extended version(e.g. $T(\neg e) := \neg T(e)$). Interesting cases are shown here. For Boolean expressions (recall that $I = a$ denotes $I = \iota(a)$)

$$\begin{aligned} T(a) &:= I = a & T(B) &:= V_B \\ T(m\%o) &:= P \bmod m = o & T(C) &:= V_C \\ T(\#(e)) &:= C_{\#(e)} & T(\#_{r_s, r_e}(e)) &:= C_{\#_{r_s, r_e}(e)} \end{aligned}$$

where P is a fresh integer variable encoding the current position. It uses an auxiliary equation $P := 0 \to \mathrm{pre}(P) + 1$. $C_{\#(e)}$, $C_{\#_{r_s, r_e}(e)}$, C_{e_1}, and C_{e_2} are fresh integer variables defined via the auxiliary equations.

The auxiliary equations of $T(R_i)$ for the examples shown are as follows. For each instance of $C_{\#(e)}$ we create a fresh a integer variable C_e that is 0 when e is false and 1 when e is true. We add the equations

$$\begin{aligned} C_e &:= \text{if } T(e) \text{ then } 1 \text{ else } 0 \\ C_{\#(e)} &:= C_e \to \mathrm{pre}(C_{\#(e)}) + C_e \end{aligned}$$

For $C_{\#_{r_s, r_e}(e)}$ we use the shorthand pre^i to define a "safe" i-fold application of pre that evaluates to 0 if there are fewer than i previous positions. That is $\mathrm{pre}^0(e) = e$ and $\mathrm{pre}^i(e) = 0 \to \mathrm{pre}^{i-1}(e)$ when $i > 0$. We also create a fresh integer variable C_e defined as above. We then add

$$C_{\#_{r_s, r_e}(e)} := \sum_{r_e \leq i \leq r_s} \mathrm{pre}^i(C_e).$$

Checking Properties of C-RASP

We can verify a number of properties of C-RASP programs by translation to Lustre. We show here how to check language inclusion. A similar encoding can check equality

Given two C-RASP programs $P_1 = (R_1^1, \ldots, R_{k_1}^1)$ and $P_2 = (R_1^2, \ldots, R_{k_2}^2)$ we can check whether P_1 only accepts words that are also accepted by P_2. We assume that P_1 and P_2 have disjoint rule names and translate P_1 and P_2 to Lustre. The only shared variable between the programs is the input variable I. Suppose V_1 and V_2 are the variables corresponding to the output equations of the two programs. Notice that, in C-RASP, at least one character is needed to determine the value of the output equations. The following check constraint asserts that after the first position, either the input I is incorrect, the end of the stream marker has not been reached, or P_2 accepts whenever P_1 does.

$$\text{check} \top \rightarrow (\neg(B_{\hat{\iota}} \wedge I = \#) \vee \neg\text{pre}(V_1) \vee \text{pre}(V_2))$$

If the property does not hold, Kind2 reports a witnessing trace of I. This trace gives counter-example word after applying the inverse of ι and removing $\#$ and Ω.

We define $T_{\subseteq}(P_1, P_w)$ to be the Lustre program with the equations $\mathcal{R}_I \cup \bigcup_i T(R_i^1) \cup \bigcup_i T(R_i^2)$ and the check statement defined above.

Proposition 3.2. *Take C-RASP programs P_1 and P_2. P_1 accepts only words accepted by P_2 if $T_{\subseteq}(P_1, P_2)$ is valid.*

The translation from C-RASP to Lustre is polynomial except for $\#_{r_s,r_e}$. To translate $\#_{r_s,r_e}$ we use $pre^i(C_e)$ for each $r_e \leq i \leq r_s$. If r_s and r_e are encoded in binary, the translation is exponential. Hence, we assume that r_s and r_e are given in unary. We do not need a similar restriction for $m\%o$.

Proposition 3.3. *Take C-RASP programs P_1 and P_2. Assume each r_s and r_e appearing in the $\#_{r_s,r_e}$ operators is encoded in unary. The Lustre program $T_{\subseteq}(P_1, P_2)$ is of size polynomial in the size of P_1 and P_2.*

4 Synthesis of C-RASP Programs

We synthesize C-RASP programs with a local search procedure based on simulated annealing [Kirkpatrick *et al.*, 1983; Henderson *et al.*, 2003]. It starts from an initial program and repeatedly proposes a neighboring program obtained by applying syntactic mutations. Given a set of words labeled as positive and negative samples, we aim to synthesize a program that is consistent with the samples while remaining small and structurally simple.

State Space. A C-RASP program over an alphabet Σ is written as a finite sequence $P = (R_1, \ldots, R_k)$ where each R_i is either a Boolean rule or a Count rule, and R_k is the distinguished Boolean rule that determines language acceptance.

To bias the search toward a concise and interpretable model, we restrict the search to programs of a fixed *program shape*. Formally, a shape is a tuple

$$\sigma = (N_b, N_c, K),$$

where N_b and N_c are the number of Boolean and Counting rules, respectively, and K bounds the range of numeric constants that may appear in Counting expressions. Only programs that satisfy these bounds are considered during search.

Let $\mathcal{P}_\sigma$ denote the set of all well-formed C-RASP programs over Σ that contain exactly N_b Boolean rules and N_c counting rules and use only constants from $\{0, \ldots, K\}$. The local search explores the finite state space $\mathcal{P}_\sigma$.

Mutation Operator and Neighborhood. Local search proceeds by repeatedly modifying the current program using a stochastic mutation operator. The mutation operator induces the neighborhood relation explored by simulated annealing. The mutation operator defines a Markov transition

$$\mathsf{Mutate} : \mathcal{P}_\sigma \rightarrow \mathcal{P}_\sigma \rightarrow \mathbb{R}$$

mapping each program P to a distribution $\mathsf{Mutate}(P)$ over the successor programs; we require $\mathsf{Mutate}(P)(P') \geq 0$ and $\sum_{P' \in \mathcal{P}_\sigma} \mathsf{Mutate}(P)(P') = 1$ for all $P, P' \in \mathcal{P}_\sigma$. The neighborhood of a program P is defined as

$$\mathcal{N}_\sigma(P) = \{\, P' \in \mathcal{P}_\sigma \mid \mathsf{Mutate}(P)(P') > 0 \,\}.$$

Each neighbour differs from P by a single conservative rewrite of the right-hand side of one rule. Although individual mutations are local, the neighborhood is sufficiently rich that any two programs of the same program shape can be connected by a finite sequence of mutations.

At each iteration, the mutator selects either a Boolean rule or a Counting rule in a program P, rewrites the right-hand side of the rule, and removes rules that are unreachable from the final Boolean expression R_k. All mutations preserve well-formedness and acyclicity by construction.

Boolean Rule Mutation. Boolean rules are mutated by resampling or locally modifying their right-hand sides. The mutation operator only uses a non-nested fragment of expressions to bias the search toward simple programs. In particular, Boolean expressions are limited to expressions in which logical connectives and negation may be applied only to Boolean atoms, and nesting of Boolean operators is disallowed. This restriction ensures that the program remains simple, while more complex behavior can still be expressed through dependencies between expressions.

A Boolean atom occurring in the right-hand side of a Boolean rule B_i may be one of the following: an alphabet test $a \in \Sigma$; a reference to an earlier Boolean rule B_j with $j < i$; a position–period predicate $m\%o$, where $m \in \mathbb{N}$ and $o < m$; or a comparison $CExp_1 \bowtie CExp_2$, where $\bowtie \in \{=, \leq, <\}$.

In addition to full resampling, we allow *micro-mutations* that preserve expression depth while modifying only a local syntactic feature. Examples include swapping $\wedge$ and $\vee$, toggling negation, flipping the strictness of a comparison, or resampling a single leaf atom. These micro-mutations define small, incremental moves in the neighborhood.

Counting Rule Mutation. As with Boolean expressions, we work with a shallow fragment of the syntax defined in Definition 2.3. During mutation, the right-hand side of each counting rule is required to have syntactic depth at most 2. Counting expressions are therefore built from count atoms using at most one arithmetic operator or one conditional expression. A new right-hand side is either resampled entirely or modified using a micro-mutation. Resampled Counting expressions are drawn from the grammar

$$CExp ::= x \mid \mathsf{op}(x, y) \mid BExp \ ?\ x : y,$$

where x, y are count atoms and $\mathsf{op} \in \{+, -, \min, \max\}$.

Count atoms are drawn from a finite, type-directed pool: integer constants $k \in \mathbb{N}$; references to earlier counting rules C_j with $j < i$; or count of PERIODIC predicate $\#(\phi)$ or LOCAL predicate $\#_{r_s,r_e}(\phi)$. The predicate ϕ is a Boolean atom that does not contain comparisons or equalities. For LOCAL predicates, bounds are normalized so that $r_s \leq r_e$.

Objective Function. Each candidate program P is evaluated on a fixed set of labeled examples $D = D^+ \cup D^-$. The primary objective is to minimize the number of misclassified samples:

$$\mathrm{mis}(P) = \sum_{x \in D^+} \mathbb{I}[P(x) = 0] + \sum_{x \in D^-} \mathbb{I}[P(x) = 1].$$

Here, $\mathbb{I}[\cdot]$ is the indicator function, equal to 1 if P misclassifies x and 0 otherwise. Among programs with equal classification behavior, we prefer simpler programs. We penalize on expressions unreachable from R_k in the dependency graph, and the total size of the abstract syntax tree. The resulting scoring function is

$$E(P) = \lambda_{\mathrm{mis}} \cdot \mathrm{mis}(P) + \lambda_U \cdot \mathrm{unreach}(P) + \lambda_S \cdot \mathrm{size}(P),$$

where $\lambda_{\mathrm{mis}} \gg 1$. This demands that the search first minimizes $\mathrm{mis}(P)$ and, among programs with equal misclassified count, prefers structurally simpler programs.

Annealing Dynamics. At each iteration, the algorithm maintains a current program P and proposes a neighbouring program P'. Let $\Delta = E(P') - E(P)$. If $\Delta \leq 0$, the new program P' is always accepted. Otherwise, it is accepted with probability $\exp\left(-\frac{\Delta}{T}\right)$, where $T > 0$ is the current temperature. The temperature is initialized to T_0 and decreased using geometric cooling. To mitigate premature convergence, we optionally apply periodic reheating.

The complete simulated-annealing synthesis procedure is presented in Algorithm 1, which maintains the current program and the best program encountered so far, applies mutation-based proposals, and accepts or rejects them according to the probability above. Note that if $\Delta \leq 0$, then $\exp(-\Delta/T) \geq 1$ and the new program P' is accepted with probability 1 in line 7. Termination occurs when the iteration budget is exhausted or when no further improvement is observed under a sufficiently low temperature.

As a refinement (not shown in Algorithm 1), once a program with $\mathrm{mis}(P) = 0$ is discovered, the search may continue under a hard constraint that forbids reintroducing misclassification, while increasing structural penalties to accelerate simplification.

Algorithm 1 Simulated-Annealing Synthesis for C-RASP Programs

1: **Input:** initial program P_0; sample set D; temperature parameters (T_0, α, K, ρ); iteration budget N
2: **Output:** best program $P^\star$
3: $P \leftarrow P_0$, $P^\star \leftarrow P$, $T \leftarrow T_0$
4: **for** $i = 0$ to $N - 1$ **do**
5: Pick P' according to distribution $\mathsf{Mutate}(P)$
6: Pick $r \in [0, 1]$ uniformly at random
7: **if** $r \leq \exp(-(E(P') - E(P))/T)$ **then**
8: $P \leftarrow P'$
9: **end if**
10: **if** $E(P) < E(P^\star)$ **then**
11: $P^\star \leftarrow P$
12: **end if**
13: $T \leftarrow \alpha \cdot T$
14: **if** $K > 0$ and $(i + 1) \bmod K = 0$ **then**
15: $T \leftarrow \rho \cdot T$
16: **end if**
17: **end for**
18: **return** $P^\star$

5 Experiments

We implement a unified Scala framework integrating *C-RASP synthesis and verification* within a single modular toolchain. The framework — combining example-driven local search with verification — consists of three core components:

1. C-RASP synthesis based on simulated annealing,
2. a lightweight C-RASP evaluator for fast scoring on finite datasets, and
3. a verification back-end that translates C-RASP programs to Lustre and checks properties using Kind2.

Building on this integration, our framework supports two verification-guided synthesis applications: 1) Transformer Program Minimization and 2) Constraint Learning.

1. Given a specification program P_{spec}, the goal is to synthesize a program P such that $L(P) = L(P_{spec})$ while minimizing structural complexity. The procedure alternates between local search and equivalence checking; when equivalence fails, Kind2 provides a counterexample trace that is added to the trace set to guide further search. The process terminates once equivalence is established or a predefined limit is reached.
2. Given labeled examples $D = D^+ \cup D^-$ and a partial specification program P_{spec}, constraint learning aims to synthesize a program P that fits all examples and satisfies $L(P) \subseteq L(P_{spec})$. Local search is guided by D and each candidate is checked for inclusion; if the check fails, the resulting counterexample is added to D as a negative example and synthesis continues.

We evaluate on a benchmark suite of regular, counting-based, and context-free–style languages. For each task, synthesis is trained on 1000 balanced examples with string lengths in $[\ell_{\min}, 100]$, where $\ell_{\min}$ is the minimum valid length for the target language, using an 80%–20% train/test split to report both train accuracy (TA) and evaluation accuracy (EA). The objective function is parameterized with $\lambda_{mis} = 1000$, $\lambda_U = 200$, and $\lambda_S = 100$. Simulated annealing runs a budget of 10^5 iterations. The temperature (initially $T_0 = 1.0$) is cooled exponentially with rate $\alpha = 0.9995$, and reheated every 4000 iterations by a factor of 1.2.

SL Synthesis is the backbone of both applications and achieves high accuracy across benchmarks, including those requiring non-trivial counting (Table 1). Consistent with the known expressivity theory [Huang *et al.*, 2025; Yang *et al.*,

	SL Synthesis			Program Minimization			Constraint Learning		
Language	**TA (%)**	**EA (%)**	**Time (s)**	**R**	**Synth (s)**	**Verif (s)**	**R**	**Synth (s)**	**Verif (s)**
Dyck-1	99.75	99.5	3.92	21	27.7	41.7	12	–	–
AStarBStar	100	100	24.62	11	7.7	46.8	1	10.2	8.9
AnBnCn	100	100	80.05	22	88.7	42.4	2	–	–
AAStar	100	100	1.57	4	1.0	10.0	1	15.5	8.3
ContainsAB	100	100	17.35	6	2.2	12.5	1	145.0	10.42
Majority	100	100	6.02	9	4.8	19.9	1	61.4	8.9
Existential	93.12	92.5	4.62	6	2.2	12.5	4	48.0	13.6
PT-2	100	100	13.30	12	9.2	29.7	1	133.7	12.0
PT-3	100	100	59.06	21	91.3	46.4	3	–	–
PT-12	100	100	90.425	20	–	–	10	–	–
D_2	99.75	100	2.26	27	63.9	90.8	11	68.5	48.2
D_3	99.00	98.01	1.54	12	–	–	15	–	–
D_{12}	99.75	98.01	1.32	22	–	–	17	–	–
Tomita 1	100	100	3.45	5	1.4	11.2	1	28.9	8.3
Tomita 2	100	100	3.03	21	33.6	58.3	9	78.4	38.3
Tomita 4	100	100	21.32	21	50.2	52.0	2	–	–
Tomita 7	100	100	9.32	22	79.1	59.8	1	279.4	18.3
Next(Argmax)	100	100	7.20	27	153.5	46.9	2	185.7	6.3

Table 1: C-RASP synthesis results across benchmark languages (timeout: 300s). SL synthesis reports train accuracy (TA), evaluation accuracy (EA), and runtime; program minimization and constraint learning report refinement rounds (R), synthesis time (Synth), and verification time (Verif). "–" indicates that no 100%-accurate program was found within the timeout.

2025], languages beyond C-RASP's expressive power (Parity, Tomita 3, Tomita 5, etc.) could not be learned with a sufficiently high accuracy; within the alloted time, the best training accuracy is below 80%. Full results and the corresponding transformer training comparisons can be found in the technical report.

Program Minimization and Constraint Learning are *correct by construction*: a candidate is accepted only after Kind2 discharges the equivalence (resp. inclusion) checked against P_{spec}, and every failed check is recycled as a counterexample. Accuracy is therefore 100% on all inputs whenever the loop succeeds, so the table reports only refinement rounds and synthesis/verification time. In Program Minimization, local search quickly finds correct but non-minimal programs, while verification dominates the runtime by driving structural simplification, especially for richer counting structures. In Constraint Learning, refinement rounds are typically few since the labeled examples restrict the hypothesis space; verification cost stays low, while synthesis grows under tighter semantic constraints.

6 Discussion and Related Work

Formal models of transformers. The intricacy of formally modeling transformers arises from the following features (among others): precision (polynomial, logarithmic, and fixed), attention mechanisms (hardmax/softmax attention), positional encodings (periodic, local, relative, etc.), and uniformity (a uniform description independent of the input length). These gave rise to a number of different formal models (e.g. see [Strobl *et al.*, 2024]). Over the years, experimental results concerning learnability revealed that several variants of softmax transformers [Huang *et al.*, 2025; Yang *et al.*, 2025] that are tightly related to C-RASP capture concepts that are learnable by transformers. [Huang *et al.*, 2025] proposed a formal version of the RASP-L conjecture [Zhou *et al.*, 2024] that C-RASP captures *precisely* languages that are expressible by length-generalizable transformers.

RASPs. [Friedman *et al.*, 2023] proposed learning a RASP by pre-restricting the architecture and parameters for transformer training. The difficulty with this approach (especially for C-RASP) is to find the right "restriction" so that a C-RASP can be extracted; we are not aware of such work. Our synthesis algorithm directly finds a C-RASP without going through a transformer, which we showed to be competitive with transformer training on our benchmarks.

B-RASP [Yang *et al.*, 2024] is a version of RASP, which supports manipulations of bit sequences. B-RASP can express languages that are not efficiently learnable (see [Huang *et al.*, 2025]), including the flip-flop language [Liu *et al.*, 2023], which makes it not a suitable model for real-world transformers. The model overapproximates concepts expressible by fixed-precision transformers [Li and Cotterell, 2025].

Verification of neural networks. Such works (e.g. [Katz *et al.*, 2022; Huang *et al.*, 2020; Liu *et al.*, 2021; Huang *et al.*, 2017]) focus mostly on verifying neural networks with a fixed number of input nodes. Yearly competition (e.g. see [Brix *et al.*, 2024]) is now held on such verification tasks. Here, the verification problem is decidable (in fact, NP-complete), which is not the case for transformers [Sälzer *et al.*, 2025].

Future Work. Numerous open problems lie ahead including the extension of our methods to other properties, e.g., "robustness". This is a common property in neural networks verification (e.g. [Brix *et al.*, 2024]), but the right formulation (that is motivated by practical applications) for sequential models is yet to be explored, e.g., perhaps via metrics on strings like hamming/edit distances. Since practical transformers use "Chain of Thoughts (CoTs)" [Jiang *et al.*, 2026c], another direction is to incorporate CoTs in our framework.

Acknowledgments

This material is based in part upon work supported by the European Union‡  (ERC, LASD, 101089343, https://doi.org/10.3030/101089343) and by the Swedish Research Council through grants 2021-06327 and 2025-06185.

References

[Arenas *et al.*, 2021] Marcelo Arenas, Daniel Báez, Pablo Barceló, Jorge Pérez, and Bernardo Subercaseaux. Foundations of symbolic languages for model interpretability. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 11690–11701, 2021.

[Barceló *et al.*, 2020] Pablo Barceló, Mikaël Monet, Jorge Pérez, and Bernardo Subercaseaux. Model interpretability through the lens of computational complexity. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[Barceló *et al.*, 2024] Pablo Barceló, Alexander Kozachinskiy, Anthony Widjaja Lin, and Vladimir V. Podolskii. Logical languages accepted by transformer encoders with hard attention. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[Brix *et al.*, 2024] Christopher Brix, Stanley Bak, Taylor T. Johnson, and Haoze Wu. The fifth international verification of neural networks competition (VNN-COMP 2024): Summary and results. *CoRR*, abs/2412.19985, 2024.

[Caspi *et al.*, 1987] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 178–188, New York, NY, USA, 1987. Association for Computing Machinery.

[Champion *et al.*, 2016] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind2 model checker. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 510–517, Cham, 2016. Springer International Publishing.

[Friedman *et al.*, 2023] Dan Friedman, Alexander Wettig, and Danqi Chen. Learning transformer programs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

‡Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

[Henderson *et al.*, 2003] Darrall Henderson, Sheldon H. Jacobson, and Alan W. Johnson. The theory and practice of simulated annealing. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 287–319. Springer US, Boston, MA, 2003.

[Hong *et al.*, 2025] Chih-Duo Hong, Hongjian Jiang, Anthony W. Lin, Oliver Markgraf, Julian Parsert, and Tony Tan. Extracting robust register automata from neural networks over data sequences. *CoRR*, abs/2511.19100, 2025.

[Hongjian *et al.*, 2026] Jiang Hongjian, Matthew Hague, Philipp Rümmer, and Anthony Widjaja Lin. (artifact) synthesis and verification of transformer programs. Artifact on Zenodo, May 2026.

[Huang *et al.*, 2017] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2017.

[Huang *et al.*, 2020] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 37:100270, 2020.

[Huang *et al.*, 2025] Xinting Huang, Andy Yang, Satwik Bhattamishra, Yash Raj Sarrof, Andreas Krebs, Hattie Zhou, Preetum Nakkiran, and Michael Hahn. A formal framework for understanding length generalization in transformers. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[Jiang *et al.*, 2026a] Hongjian Jiang, Matthew Hague, Philipp Rümmer, and Anthony Widjaja Lin. Synthesis and verification of transformer programs (technical report), 2026.

[Jiang *et al.*, 2026b] Hongjian Jiang, Michael Hahn, Georg Zetzsche, and Anthony Widjaja Lin. Softmax transformers are turing-complete. In *The Fourteenth International Conference on Learning Representations*, 2026.

[Jiang *et al.*, 2026c] Hongjian Jiang, Michael Hahn, Georg Zetzsche, and Anthony Widjaja Lin. Softmax transformers are turing-complete. In *The Fourteenth International Conference on Learning Representations*, 2026.

[Katz *et al.*, 2022] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods Syst. Des.*, 60(1):87–116, 2022.

[Kirkpatrick *et al.*, 1983] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[Krishnan *et al.*, 2024] N. M. Anoop Krishnan, Hariprasad Kodamana, and Ravinder Bhattoo. *Interpretable Machine Learning*, pages 159–171. Springer International Publishing, Cham, 2024.

[Li and Cotterell, 2025] Jiaoda Li and Ryan Cotterell. Characterizing the expressivity of fixed-precision transformer language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[Liu *et al.*, 2021] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher A. Strong, Clark W. Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks. *Found. Trends Optim.*, 4(3-4):244–404, 2021.

[Liu *et al.*, 2023] Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Exposing attention glitches with flip-flop language modeling. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[Marques-Silva and Ignatiev, 2023] Joao Marques-Silva and Alexey Ignatiev. No silver bullet: interpretable ml models must be explained. *Frontiers in Artificial Intelligence*, 6, 2023.

[Merrill and Sabharwal, 2024] William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[Okudono *et al.*, 2020] Takamasa Okudono, Masaki Waga, Taro Sekiyama, and Ichiro Hasuo. Weighted automata extraction from recurrent neural networks via regression on state spaces. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 5306–5314. AAAI Press, 2020.

[Pérez *et al.*, 2021] Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing-complete. *J. Mach. Learn. Res.*, 22:75:1–75:35, 2021.

[Sälzer *et al.*, 2025] Marco Sälzer, Eric Alsmann, and Martin Lange. Transformer encoder satisfiability: Complexity and impact on formal reasoning. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[Sälzer *et al.*, 2026] Marco Sälzer, Chris Köcher, Alexander Kozachinskiy, Georg Zetzsche, and Anthony Widjaja Lin. The counting power of transformers. In *The Fourteenth International Conference on Learning Representations*, 2026.

[Strobl *et al.*, 2024] Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What formal languages can transformers express? A survey. *Trans. Assoc. Comput. Linguistics*, 12:543–561, 2024.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[Weiss *et al.*, 2021] Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *International Conference on Machine Learning*, pages 11080–11090. PMLR, 2021.

[Weiss *et al.*, 2024] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples (extended version). *Mach. Learn.*, 113(5):2877–2919, 2024.

[Yang and Chiang, 2024] Andy Yang and David Chiang. Counting like transformers: Compiling temporal counting logic into softmax transformers. In *First Conference on Language Modeling*, 2024.

[Yang *et al.*, 2024] Andy Yang, David Chiang, and Dana Angluin. Masked hard-attention transformers recognize exactly the star-free languages. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024.

[Yang *et al.*, 2025] Andy Yang, Michaël Cadilhac, and David Chiang. Knee-deep in c-RASP: A transformer depth hierarchy. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[Yang *et al.*, 2026] Andy Yang, Pascal Bergsträßer, Georg Zetzsche, David Chiang, and Anthony W. Lin. Length generalization bounds for transformers. In *Forty-third International Conference on Machine Learning*, 2026.

[Zhou *et al.*, 2024] Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Joshua M. Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? A study in length generalization. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.