DEPTH IS NOT ENOUGH: COMPOSITIONAL GENERALIZATION IN LOOPED TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Compositional generalization requires reusing familiar operations in new combinations. For computations whose dependencies form a Directed Acyclic Graph (DAG), this involves both locating each operation's operands (syntax inference) and computing with them (syntax execution). Looped Transformers have demonstrated impressive generalization abilities by repeatedly applying a shared block, but it is unclear to what tasks they can generalize and why. In this work we introduce a task ladder, from prefix scans to serialized arithmetic trees, that progressively increases the difficulty of syntax inference. Models are trained on small compositions and tested on larger ones. The reference computation of every task is fully known, but models must infer its syntax from the input and then execute it. This lets us supply the true operands to a trained model and test the cause of a compositional failure. On this ladder, looped models fit every task in the training range but extrapolate only when each operand lies at a fixed offset from its operation. Larger models, other positional encodings, chain-of-thought and a pretrained 3.5B looped model do not close this gap. We establish syntax inference as the cause of this generalization failure through test time interventions: restricting a frozen model's attention to the true operand positions substantially improves extrapolation. Adding iterations, by contrast, does not help: beyond a few iterations, accuracy stops improving. Computational depth is therefore not enough: looped models must also learn to locate operands in a way that holds beyond the training range. Our results highlight limitations of current looped models and improve our understanding of what enables compositional generalization.

1 INTRODUCTION

A program can use familiar operations yet require an unfamiliar computation. For example, multiplying longer integers with the standard written algorithm reuses the same single-digit products, additions, and carry operations, but requires coordinating them across more intermediate results (Dziri et al., 2023). Knowing the local operations is therefore not enough: their outputs must also be combined in the correct way. This distinction is central to compositional generalization. Riveland et al. (2026) organize compositionality along the expressivity of reusable *modules* and the complexity of the *syntax* that combines them. Work on discovering reusable modules (Schug et al., 2024) addresses the first, but failures such as longer multiplication arise in the second. We therefore study how a fixed repertoire of learned operations can be composed into structures larger than those encountered during training. Within this setting, we distinguish *syntax inference*, i.e., determining which intermediate results each operation should use (its *operands*), from *syntax execution*, i.e., carrying out the resulting composition.

We study computations that admit a Directed Acyclic Graph (DAG), with nodes representing operations and directed edges specifying the intermediate results they consume. This connects our setting to computational-graph analyses of compositional tasks (Dziri et al., 2023). In this work, *syntax* is the set of operation labels together with which operands each operation takes, in order. Operation labels are visible in our inputs, but no token names an operation's operands, so syntax inference amounts to locating them by position in the serialized input. We further say a model has *operand access* when it attends to those positions. In each task the rule determining operand positions is fixed, but we test its application to larger compositions than those seen at training time. A model

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

may therefore execute a composition correctly when its operands are supplied (successful syntax execution), yet fail to locate them itself (failed syntax inference).

As compositions grow, solving them may require additional sequential computation. Fixed-depth Transformers impose a fixed number of computational stages. Under precision and computability assumptions, their expressivity is bounded by the circuit class TC^0, whereas, for instance, general Boolean formula evaluation is NC^1-complete (Merrill & Sabharwal, 2023; Buss et al., 1992). Under the common assumption that these classes differ, fixed depth cannot support all such computations at arbitrary size. Chain-of-thought (CoT) generation offers one solution, introducing sequential computation through additional generated tokens (Merrill & Sabharwal, 2023). Looped Transformers offer another: repeating a shared block lets effective depth grow without new parameters, in hidden states instead of generated tokens, so looped models need no reasoning traces for training and do not lengthen the context window (Saunshi et al., 2025; Geiping et al., 2025). These expressivity results show what added depth can theoretically compute, not what training finds: they do not guarantee that a looped model learns an algorithm that generalizes. Nor do they separate syntax inference from syntax execution: additional computation may support either.

Real-world inputs, such as programs, formulas or chains of tool calls (Popescu et al., 2026), are not designed to match an architecture, so a model must perform syntax inference, locating operands itself. Despite the strong extrapolation results of looped models on algorithmic tasks (Bansal et al., 2022; Fan et al., 2025), their architecture is typically well-matched to the task's dependencies, so the model never has to infer the syntax. For instance, in prefix sum and maze tasks, each step needs only neighboring positions, which is exactly what a convolution attends to (Bansal et al., 2022). In the reachability experiments of Chen (2026), adjacency masks supply each node's neighbors directly. Thus in both cases it is only the generalization of syntax *execution* that is being tested. Conversely, past work reporting failure of generalization measures only out-of-distribution error, not its cause (Dziri et al., 2023; Zhou et al., 2024): when a model fails on a larger composition, we cannot tell whether it located the wrong operands or lacked the computation to execute them.

This work introduces a rigorous framework that precisely attributes failure of compositional generalization. Our central question is: *When models must infer the syntax they execute, is extra recurrent computation enough to generalize, and if not, what is missing?* We answer this in three steps: 1) building a task ladder on which the cause of a failure can be tested, 2) asking whether current looped models extrapolate on them, and, 3) where they fail, intervening to test whether computational depth or operand access is the cause. Our contributions are as follows:

- We introduce **a novel framework for attributing compositional failure** (Section 2). We construct a task ladder, from prefix scans to arithmetic trees, comprising simple local operations and representing a controlled increase in the difficulty of locating operands. Each task is defined by an exact reference computation formulated as a DAG. This reference computation links every operation to its operands, allowing us to separate two possible causes of failure: syntax inference and syntax execution.

- We show that **looped models fail to generalize once operands must be located** (Section 3). On this ladder, looped Transformers fit every task in the training range but extrapolate only while each operand lies at a fixed offset from its operation, and fail once its position varies. Larger models, other positional encodings, chain-of-thought models and a pretrained 3.5B recurrent model do not close this gap.

- We show that **compositional failure is due to operand access rather than limited depth** (Section 4). We vary one factor at a time—supplying the true operands at test time and increasing computational depth—in trained models. We find that, holding computation fixed, restricting a frozen model's attention to the true operand positions substantially improves extrapolation, whereas restricting it to equally many positions of the model's own choosing (top-k attention) does not. Holding the operand access fixed, no iteration budget restores extrapolation: beyond a few iterations, accuracy stops improving.

2 A FRAMEWORK FOR ATTRIBUTING COMPOSITIONAL FAILURE

2.1 PROBLEM SETUP

We consider a model that must evaluate a computation written as a sequence of tokens, $x = (x_1, \ldots, x_n) \in \mathcal{X}^n$, where $\mathcal{X}$ is the task's vocabulary. For example, the expression $(3+5) \times (9-2)$ is given in postfix order as $x = (3, 5, +, 9, 2, -, \times)$. Each token is an input value or an operation, and the input does not state which earlier tokens each operation takes as operands: locating them from the sequence is syntax inference.

Each token x_i specifies a *value* v_{x_i} and a local *function* f_{x_i}. Each local function has a fixed arity – the number of previous node outputs it consumes. For function f_{x_i} with arity a, the expected *output* at position i is

$$o_i = f_{x_i}\big(v_{x_i}; o_{b_1(i;x_{\le i})}, \ldots, o_{b_a(i;x_{\le i})}\big), \qquad b_k(i; x_{\le i}) < i. \tag{1}$$

Here, $b_k(i; x_{\le i})$ is the position of operand k: depending on the task, it depends only on the position i (e.g. the previous position) or also on the tokens up to i. The specific positions are set by the task's *binding rule*: in the earlier arithmetic example, $b_1(7; x_{\le 7}) = 3$ and $b_2(7; x_{\le 7}) = 6$. Operand order matters, for example, for subtraction. Equation (1) thus defines a DAG: each position is a node, with an edge from each operand $b_k(i; x_{\le i})$ to i. Since $b_k(i; x_{\le i}) < i$, nodes are serialized in topological order: every operand precedes its operator. Leaf nodes l encode a function of arity 0 that returns their own value v_{x_l}. Tasks defined by scans (A, B1, B2 below) additionally use an initial-state node 0. For single-operand scans we write $b(i)$.

Syntax inference is thus the problem of learning $b_k(i; x_{\le i})$ from training examples and applying it to every new input, while *syntax execution* is the application of f_{x_i} to the retrieved operands, propagating intermediate results across the depth of the computation. The local functions and binding rule remain fixed between training and testing, but extrapolation requires applying them to larger compositions.

We define *reference depth* as follows: first inductively define the depth of each node i as $d_i = 0$ for leaves and node 0 and $d_i = 1 + \max_{k \le a} d_{b_k(i;x_{\le i})}$ otherwise. Reference depth D is the maximum node depth, $D = \max_i d_i$. With correct operand access and local functions, synchronous updates evaluate the graph in at most D rounds (Appendix A). A model may need fewer: associative scans such as task A admit parallel shortcuts (Liu et al., 2023).

Supplying the syntax. Since the tasks we define have a known binding rule, the true operands can be supplied to a model through an *operand mask*, which restricts each position's attention to itself and its operands:

$$\mathcal{M}_i = \{i\} \cup \{b_1(i; x_{\le i}), \ldots, b_a(i; x_{\le i})\}. \tag{2}$$

The mask provides the model only with operand *position*, not their (intermediate) values.[1] A model trained with it need only learn syntax execution, and if inserting it into a model trained without it restores accuracy, that model failed at syntax inference (Section 4.1). Varying the number of iterations while holding operand access fixed tests instead whether it lacks computation (Section 4.2).

2.2 A TASK LADDER FOR LOCATING OPERANDS

Our ladder comprises four tasks, and each step on the ladder introduces a more complex binding rule for the model to learn. Figure 1 (bottom) gives an example of each, with arrows from operands to their operators. Figure 1 (top) further specifies which binding rule requirement each task adds to the previous one. Every node takes values in the same finite set at every depth (bits for A, S_3 for B1 and B2 or $\mathbb{Z}_{11}$ for C), so larger compositions add steps but not new values.

A: Prefix XOR. For binary inputs, the local operation and reference scan are

$$f_{x_i}(v_{x_i}; o_{i-1}) = o_{i-1} \oplus v_{x_i}, \qquad b(i) = i - 1, \qquad o_0 = 0, \qquad D = n.$$

[1] The mask is a set, so it does not say which operand is left and which is right, and subtraction depends on the order. Appendix B.1 describes how each model recovers the order, and why the tree oracle uses role biases instead of RoPE. Figure B.1 shows an oracle that keeps RoPE.

Operand ...	A	B1	B2	C: postfix trees			
				ragged	left cat.	right cat.	complete
must be retrieved	×	✓	✓	✓	✓	✓	✓
+ at a varying distance	×	×	✓	✓	×	✓	✓
+ located via a derived quantity	×	×	×	✓	×	✓	✓
+ at distances unseen in training	×	×	×	most	×	×	✓

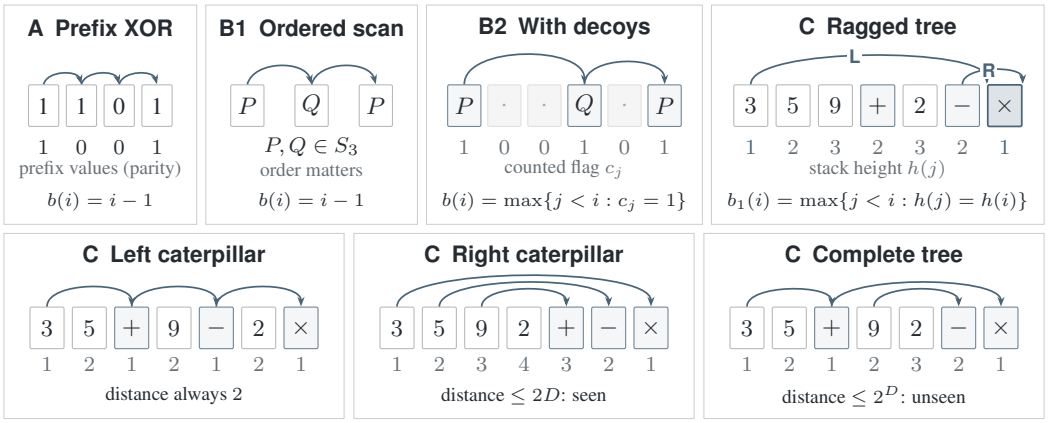

Figure 1: **The task ladder.** Top: each row adds a requirement on how an operation obtains its operand. *Retrieved*: the value cannot be obtained by counting over the prefix (A's parity can). *Varying distance*: the offset changes across positions (B2: 2–4; left caterpillars: always 2). *Derived quantity*: the position depends on a quantity computed from the prefix (C: stack height), not on a visible flag. *Unseen distances*: offsets above the training maximum of 16. Bottom: arrows connect each operation to its operands (for the ragged tree, only the root's). Gray numbers are what the binding rule reads: the flag c_j in B2 and the stack height $h(j)$ in C. "Seen" and "unseen" compare distances at test depths 5–8 with training depths 1–4.

For example, $x = (1, 1, 0, 1)$ gives $o = (1, 0, 0, 1)$. This prefix-sum, modulo-two task provides an established recurrent extrapolation baseline (Bansal et al., 2022). However, XOR is associative and commutative: $o_i = (\sum_{j=1}^{i} v_{x_j}) \bmod 2$. This means that each prefix value can alternatively be computed by aggregation *without* retrieving the preceding state o_{i-1}.

B1: Ordered scan. We replace bits with elements of S_3, the six permutations of three objects:

$$f_{x_i}(v_{x_i}; o_{i-1}) = o_{i-1} \circ v_{x_i}, \qquad b(i) = i - 1, \qquad o_0 = e, \qquad D = n,$$

where e is the identity permutation. For example, $x = (P, Q, P)$ gives $o = (P,\ PQ,\ PQP)$. As the smallest non-abelian group, S_3 makes composition order-sensitive. Thus the preceding state must be retrieved, while retaining a small value space and adjacent access. Related permutation-group tasks appear in studies of Transformer automata simulation (Liu et al., 2023).

B2: Ordered scan with decoys. We retain B1's operation, but each input permutation has a visible flag $c_i \in \{0, 1\}$ marking whether it is counted. Uncounted tokens (decoys) are irrelevant. At counted positions,

$$b(i) = \max\{j < i : c_j = 1\}, \qquad D = \sum_{i=1}^{n} c_i, \qquad c_0 = 1, \quad o_0 = e.$$

For example, omitting leading decoys, in $x = (P, \cdot, \cdot, Q, \cdot, P)$ with $c = (1, 0, 0, 1, 0, 1)$, the counted tokens at positions 4 and 6 take their predecessors from positions 1 and 4, giving $o_6 = PQP$ as in B1. Before each counted token, we insert one to three decoys, so the reference predecessor is always two to four positions earlier, in both training and evaluation. The operand is therefore at a varying distance, located by a visible flag.

C: Postfix arithmetic trees. Inspired by the tree-based arithmetic tasks used by Tang et al. (2025), we evaluate nested expressions whose depth and structure can be controlled. Leaves take values in $\mathbb{Z}_{11}$, and each internal node applies addition, subtraction, or multiplication modulo 11 to its left and right operand values. These are the local functions f_{x_i}, $x_i \in \{+, -, \times\}$. Reference depth D counts internal nodes on the longest root-to-leaf path.

We serialize trees in postfix order: each operator follows its left and right subtrees. Let $h(j)$ be the stack height after token j, increasing by one for a leaf and decreasing by one for an operator. The operand positions are

$$b_2(i) = i - 1, \qquad b_1(i) = \max\{j < i : h(j) = h(i)\}. \tag{3}$$

Unlike B2's visible flag, identifying the left operand requires deriving stack height from the input and selecting its most recent match. Appendix A gives the derivation for the binding rule.

Training trees are *ragged*: one branch is forced to depth D and other nodes expand randomly, so depth, stack height and operand distance $i - b_1(i)$ vary together. We also evaluate three families at matched depth (Figure 1, bottom). All belong to the training generator's support through depth 4, and left and right caterpillars have equal sampling probability by symmetry. In *left caterpillars* the left operand is always two tokens earlier and stack height never exceeds two. *Right caterpillars* have the same $2D+1$ tokens, but maximum operand distance $2D$ and stack height $D+1$. Through depth 8, both stay within the training bounds on length (31) and distance (16), but right caterpillars exceed the height bound (5). *Complete trees* have $2^{D+1} - 1$ tokens and exceed all three bounds beyond depth 4 (Table A.1). A gap that opens beyond depth 4 therefore cannot come from depth, length or an unseen distance, and would point to the binding rule applied at unseen heights.

Generalization setting and metric. Scans (tasks A, B1, B2) are trained up to reference depth $D = 8$ and tested up to 32. B2's decoys lengthen the sequence to $2D$–$4D$ tokens without adding composition steps. Trees (task C) are trained at depths 1–4 and evaluated through depth 8. Training supervises intermediate computation states as well as the final result. We report *root accuracy*, the accuracy of the final output (last counted scan state or tree root value). Cross-task comparisons concern patterns of extrapolation, not a common difficulty scale, as chance level and task structure vary. Protocol details, including evaluation draws (Table B.2), are in Appendix B.

3 Looped models fail once operands must be located

We now ask whether looped models extrapolate on the ladder, and whether any failure is specific to our model or persists across size, positional encoding and chain-of-thought.

Models. Our recurrent backbone follows the Fixed-Point Reasoner design (Movahedi et al., 2026). Given input embeddings $E(x)$, it updates a latent state $z^{(r)} \in \mathbb{R}^{n \times d_{\text{model}}}$ over R iterations:

$$z^{(r+1)} = F_\theta(z^{(r)}; E(x)), \qquad r = 0, \ldots, R-1. \tag{4}$$

We compare a RoPE-based Transformer (Su et al., 2024) (*base*), a variant that replaces RoPE with an in-loop causal convolution over a three-token window, inspired by recurrent convolutional models (Bansal et al., 2022) (*convolution*), and a variant trained with the operand mask of Equation (2) (*oracle*). For trees, the oracle additionally receives learned operand-role biases and no RoPE, so it is a structured execution control rather than a mask-only change (Table B.3). All three share one size and training protocol (Table B.1). At evaluation, each model halts adaptively once its state stops changing, up to 128 iterations. Thus, the number of iterations R is set by the model and is a priori distinct from the task's reference depth D. Implementation details are in Appendix B.

Failure along the ladder. Figure 2 compares the three models across the ladder (top row). The base model fits every task within the training range but deteriorates beyond it: at the largest tested depth it reaches approximately 0.53 on A (chance 0.5), 0.20 on B1 and 0.43 on B2 (chance 0.17), and 0.15 on ragged trees (chance 0.09). The convolutional variant generalizes almost perfectly on A and B1, where a local neighbourhood matches the reference access rule, but not on B2 or ragged trees, where it reaches approximately 0.33 and 0.23. A wider kernel does not reliably help on B2 (Figure D.5). The oracle model remains near perfect throughout.

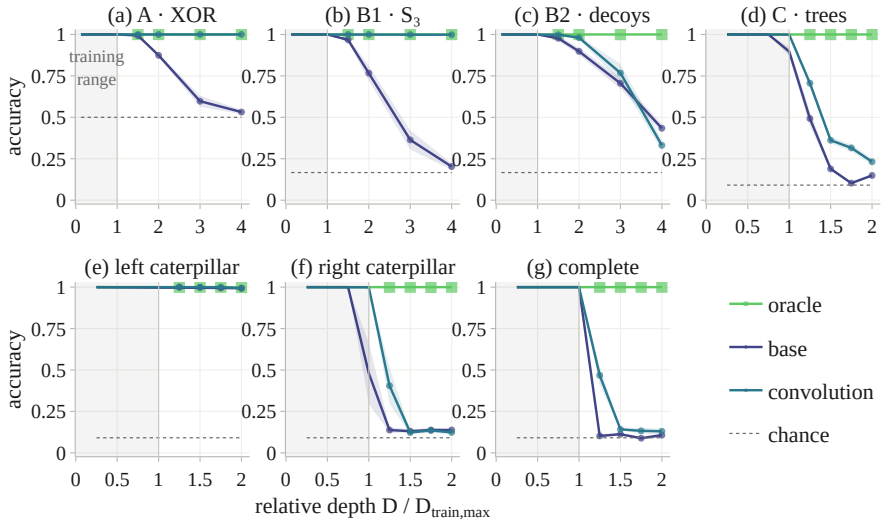

Figure 2: **Looped models fail once operands must be located.** Accuracy versus relative reference depth $D/D_{\text{train,max}}$; the shaded region is the training range. Top: the ladder, with ragged trees for C. Bottom: the tree subfamilies of task C (Section 2.2) at matched depth. Convolution supports A, B1 and left caterpillars but not B2 or C's other structures. The oracle model is trained with operand access. Configurations and aggregation details are in Appendix B.

The tree families reveal contrasting behavior (bottom row). At depth 8, base and convolutional models remain near 0.99 on left caterpillars, but reach 0.14 and 0.12 on right caterpillars and 0.11 and 0.13 on complete trees. Both caterpillars have the same length and only operand distances seen in training, but right caterpillars reach unseen stack heights, which locating the left operand requires (Equation (3)). The contrast is therefore consistent with a syntax-inference difficulty rather than a length effect. The base model already scores 0.48 on right caterpillars at depth 4, although left and right caterpillars are equally frequent in training, so part of this gap is a within-distribution shape effect. The convolutional model gives the clearer extrapolation contrast: 1.00 at depth 4, then collapse.

Failure persists across models and compute. Figure 3 compares the looped models with a fixed-depth Transformer and two chain-of-thought models, trained on a trace or on a scratchpad (Appendix D.4, Table D.2), and with Huginn, an open-weight recurrent language model (Geiping et al., 2025) adapted with LoRA (Appendix D.5, Table D.3). On trees, averaged over depths 5–8, the trace model and Huginn reach 0.23 and 0.31, against 0.23 for the base model (Figures D.6 and D.7). Both spend far more compute than the base model, about $40\times$ and $10^4\times$ its FLOPs, yet fall towards chance as depth grows, while only the oracle stays accurate. These comparisons are not matched to the base model: the CoT models have twice its parameters and Huginn is a pretrained 3.5B model.

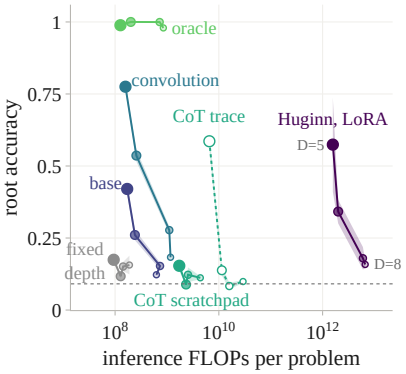

Figure 3: **Compositional accuracy against test-time FLOPs for other models.** Ragged trees, depths 5–8, at each looped model's smallest budget reaching 95% of its best (Appendix D.6). Markers shrink from depth 5 to 8; bands: ± 1 s.e.

We also change one aspect of the base model at a time (Appendix D, Table D.1). We report accuracy on scans at depth 32 and on trees averaged over depths 5–8. Higher capacity helps models extrapolate on the tasks with adjacent operands, reaching 0.98 on A and 0.97 on B1 at the best size, but not on those where the operand

must be located: 0.73 on B2 and 0.26 on trees (Figure D.2). Replacing RoPE with no positional encoding (NoPE) or with FIRE (Li et al., 2024) gives 0.16 on B2 and 0.29 on trees (Figure D.3), and a fixed-depth Transformer stays near chance on every task (Figure D.4). Across all these changes, only the oracle, which is given the operand positions, extrapolates on every task.

4 Operand access improves extrapolation, but depth does not

A failing model may lack either the right operands or enough computation. To distinguish the two, we intervene on trained models, varying one while holding the other fixed.

4.1 Holding computation fixed: supplying operand positions

We compare five settings. The *oracle* model is the one of Section 3, trained with the operand mask from the outset. The other four start from the same trained base checkpoint: unchanged (*base*), with the mask inserted at test time and weights frozen (+ *oracle, frozen*), after 50 further optimization steps under the mask on training-range examples (+ *oracle, adapted*), and with each position restricted to the k positions it attends to most, with k the size of the operand mask at an operation (2 for scans, 3 for trees) (+ *top-k, frozen*). All four keep the base model's architecture and iteration cap, so only which positions are accessible changes. Frozen insertion tests whether supplying the syntax is sufficient, adaptation lets the model adjust to the new access pattern, and top-k restricts attention to the same number of positions without supplying which ones, separating operand identity from restricting attention as such. Inserted masks retain RoPE and provide no role biases (Appendix C).

Figure 4 applies these settings to the base checkpoints. At scan depth 32, frozen insertion raises accuracy from approximately 0.21 to 0.95 on B1 and from 0.47 to 0.71 on B2. Furthermore, after 50 adaptation steps, both reach approximately 0.98, close to the oracle model. On ragged trees at depth 8 the change is $0.15 \rightarrow 0.35 \rightarrow 0.49$: substantial, but incomplete. Top-k stays at or below the base level on every task (0.21 on B1 at depth 32), although it keeps as many positions as the operand mask. This shows that the gain comes from *which* positions are accessible, not from restricting attention. Restricting attention to the model's own choices even lowers B2 from 0.47 to 0.29. Task A behaves differently: inserting the true predecessor lowers accuracy within the training range, from 1.00 to 0.56 at depth 8, and adaptation recovers only 0.70. This is consistent with the base model solving A by aggregation (Section 2.2) instead of retrieving the preceding state. As a consequence, supplying that state disrupts rather than helps.

The tree families show where recovery stops (bottom row). On left caterpillars, the base model and both operand interventions remain near 1.00. On right caterpillars, whose operand distances were seen in training, recovery is strong at depth 5 ($0.14 \rightarrow 0.47 \rightarrow 0.81$) but weakens by depth 8 ($0.14 \rightarrow 0.20 \rightarrow 0.31$). On complete trees, whose distances were not, recovery at depth 5 ($0.10 \rightarrow 0.63 \rightarrow 0.85$) vanishes by depth 8, where all three settings are near 0.12. Where the operand mask does not repair the model, the test no longer isolates syntax inference: the model's learned computation may not use the operands the mask supplies. Supplying operands therefore repairs nearly all of the failure on scans and most of it on trees one depth beyond training, but a shrinking share as trees grow deeper.

4.2 Holding the operand access fixed: adding iterations

We now keep each trained model unchanged and vary only the number of iterations it may run, with adaptive halting disabled. Figure 5 shows accuracy at the largest tested depth (trees: averaged over depths 5–8) against this budget. Models that extrapolate use the extra iterations: at scan depth 32 the oracle is at chance with 16 iterations and at 1.00 with 64. The failing models do not. Their accuracy stops rising after about 16 iterations and stays far below the oracle's; on B2 it even falls, for the base model from 0.55 at 16 iterations to 0.41 at 128. The same sweep gives $R^\star$, the smallest budget reaching 95% of a model's best accuracy at a given depth (Appendix D.1, Figure D.1). Where a model extrapolates, $R^\star$ grows with depth: the oracle needs 5, 10 and 20 iterations at scan depths 8, 16 and 32, and 8 to 13 on ragged trees from depth 5 to 8. Where the base model fails, $R^\star$ stops growing (B1: 8.7 at depth 16, 7.3 at depth 32): the model's output stops changing after

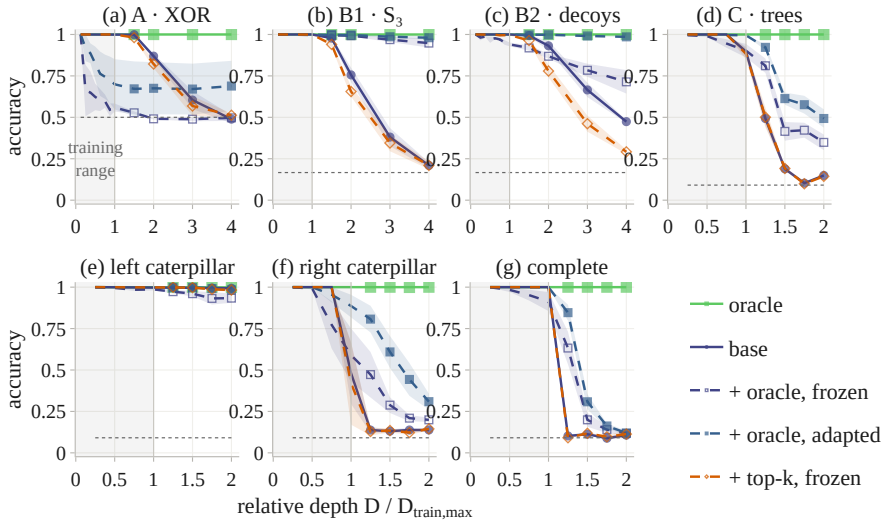

Figure 4: **Intervening at test-time on operand access improves trained models.** Each panel compares the base model with the operand mask inserted at test time (frozen) or after 50 training steps (adapted) and with the oracle model trained with the mask. Top-k attention keeps as many positions as the operand mask but is chosen by the model. Top: the ladder; bottom: the tree families, with adaptation fitted on ragged trees only. Scan panels use a separate evaluation sample, so base values differ slightly from Figure 2.

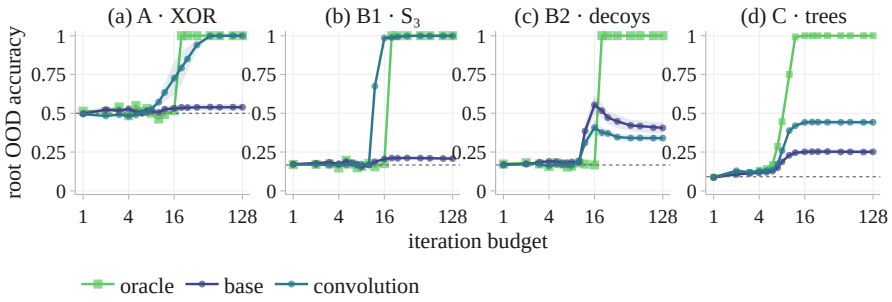

Figure 5: **More iterations do not restore extrapolation.** Root accuracy at the largest tested depth (scans $D = 32$, trees averaged over depths 5–8) against a fixed iteration budget from 1 to 128, with adaptive halting disabled. Scan values come from a separate evaluation sample, so they differ slightly from Figure 2. Bands: ± 1 s.e. over seeds; dotted lines: chance.

a few iterations, so further iterations cannot help. Additional iterations therefore do not restore extrapolation: supplying the operands (Section 4.1) largely does.

5 Related work

Compositional generalization and binding. Riveland et al. (2026) identify binding, assigning an entity to its role in a larger structure, as a candidate mechanism for syntax: a system that has learned which entities fill which roles can apply this structure to new inputs. In our tasks, the roles are the operands of each operation, and the binding rule assigns them. In practice, models often fail on larger compositions: language models fine-tuned on multi-step tasks such as multiplication are near perfect in distribution but fail on larger problems, even with step-by-step scratchpads (Dziri et al., 2023). The authors attribute this to matching subgraphs seen in training, and Zhou et al. (2024)

conjecture that Transformers extrapolate when a short program solves the task at every length. Our ladder isolates one ingredient of these failures: within each task the operations stay fixed from training to test, and from task to task their operands become harder to locate.

Looped models and generalization. Looped models extrapolate mostly where binding is fixed. Fan et al. (2025) choose tasks whose single step has a length-generalizable program and supervise the number of steps. Other successes build the binding rule into the architecture. Movahedi et al. (2026) find a causal convolution essential on permutation state tracking, where each step reads the previous position. In graph reachability, adjacency masks give each node its neighbors, and a model needs N iterations to follow N hops (Chen, 2026). On our ladder, the convolution and oracle models do the same, and extrapolate wherever the structure they build in matches the binding rule. However, in real-world tasks, models do not know the binding rule a priori. For example, in API tool calling tasks, a call often uses the result of an earlier call. There, looping reduces invalid references to earlier results, but only on compositions of training size (Popescu et al., 2026). We find that extra computation helps extrapolation only if the binding rule learned in training still locates the right operands at test time.

Locating operands with attention. Transformers learn to bind a variable to its value when the operand is named by content, such as a variable name appearing earlier in the sequence (Wu et al., 2025). In our tasks, no token names its operand: the operand's position must be inferred from the input sequence. Attention can also fail at larger sizes because softmax spreads over more tokens (Veličković et al., 2025), and boosting attention to the correct tokens at test time helps fine-tuned language models extrapolate (Spiegel et al., 2025). Both accounts predict that sharpening a model's own attention should help. Our top-k control tests this directly: restricting a frozen model's attention to its own top positions leaves extrapolation at or below the base level, whereas restricting it to the true operand positions substantially improves it.

6 Discussion and conclusion

Summary. We asked whether extra recurrent computation is enough to generalize when models must infer the syntax they execute. We show that looped models extrapolate only while each operand lies at a fixed offset from its operation. We further show that supplying the true operand positions substantially improves extrapolation, whereas adding more computational depth does not.

The remaining gap. Correct operand access, as shown in Section 4, does not fully repair a model trained without it: recovery is partial on right caterpillars and vanishes on complete trees at larger depth, where length, operand distance and the number of operations grow together. What supplied access can repair therefore depends on what the model has learned. The oracle results show that models can learn to execute deeper compositions when trained with supplied structure, which motivates models whose inferred syntax remains correct as compositions grow.

Limitations and future work. Our tasks are synthetic, with finite value domains and known reference dependencies, and our models remain relatively small. Capacity partly mitigates some failures, so we make no asymptotic claim about scale. Mask insertion further changes the recurrent dynamics as well as operand access, lowering in-distribution accuracy on A and B2 (Appendix C), and we did not apply the interventions to chain-of-thought models or Huginn. Finally, our interventions do not identify the models' internal representations or algorithms. We leave improvements along these lines as future work.

The distinction between syntax inference and execution arises in ordinary code: to evaluate `c = a * b`, a model must know what multiplication does (execution) and also find the most recent lines that assigned `a` and `b` (inference). Once variables are reassigned, this is B2's rule: the most recent position carrying a visible mark. A natural next step is to place such tasks on the ladder, by the rule that locates their operands, to predict whether a model is likely to extrapolate and which architectural change, such as a local convolution or supplied structure, it needs. The useful question is then not only how long a model should compute, but how it learns which intermediate results to connect, and whether those connections remain valid outside the training distribution.

AI USE STATEMENT

Large language model assistants were used to help implement experiment and analysis code, draft cluster scripts, produce figures from result files, search and summarize related work, and improve the clarity of the manuscript. All research questions, experimental designs and interpretations are the authors' own, and the authors take full responsibility for the content of the paper.

REPRODUCIBILITY STATEMENT

Task definitions and reference computations are given in Section 2 and Appendix A, including the generation of every tree family. Model architecture, training, halting, evaluation draws and compute are specified in Appendix B and mask construction and adaptation in Appendix C. Further comparisons are described in Appendices D.1–D.6. All data are generated online from fixed seeds, and every hyperparameter is set in a configuration file.

REFERENCES

Arpit Bansal, Avi Schwarzschild, Eitan Borgnia, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. End-to-end Algorithm Synthesis with Recurrent Networks: Logical Extrapolation Without Overthinking. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Samuel R. Buss, Stephen A. Cook, Arvind Gupta, and Vijaya Ramachandran. An Optimal Parallel Algorithm for Formula Evaluation. *SIAM Journal on Computing*, 21(4):755–780, 1992.

Hung-Hsuan Chen. Thinking Deeper, Not Longer: Depth-Recurrent Transformers for Compositional Generalization. arXiv preprint arXiv:2603.21676v1, 2026.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and Fate: Limits of Transformers on Compositionality. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Ying Fan, Yilun Du, Kannan Ramchandran, and Kangwook Lee. Looped Transformers for Length Generalization. In *International Conference on Learning Representations*, 2025.

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up Test-Time Compute with Latent Reasoning: A Recurrent Depth Approach. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional Interpolation for Relative Positions Improves Long Context Transformers. In *International Conference on Learning Representations*, 2024.

Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers Learn Shortcuts to Automata. In *International Conference on Learning Representations*, 2023.

William Merrill and Ashish Sabharwal. The Parallelism Tradeoff: Limitations of Log-Precision Transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023.

Sajad Movahedi, Vera Milovanović, Shlomo Libo Feigin, Alexander Theus, Thomas Hofmann, Valentina Boeva, T. Konstantin Rusch, and Antonio Orvieto. Fixed-Point Reasoners: Stable and Adaptive Deep Looped Transformers. arXiv preprint arXiv:2606.18206, 2026.

Andrei Cristian Popescu, Haitz Sáez de Ocáriz Borde, and Pietro Liò. Looped Language Models Improve Compositional Tool Calling. arXiv preprint arXiv:2608.18171, 2026.

Reidar Riveland, Alexandre Pouget, and Laura Driscoll. The compositionality continuum as a principle for studying the neural basis of intelligence. *Nature Neuroscience*, 29(9):2067–2080, 2026.

Nikunj Saunshi, Nishanth Dikkala, Zhiyuan Li, Sanjiv Kumar, and Sashank J. Reddi. Reasoning with Latent Thoughts: On the Power of Looped Transformers. In *International Conference on Learning Representations*, 2025.

Simon Schug, Seijin Kobayashi, Yassir Akram, Maciej Wołczyk, Alexandra Proca, Johannes von Oswald, Razvan Pascanu, João Sacramento, and Angelika Steger. Discovering modular solutions that generalize compositionally. In *International Conference on Learning Representations*, 2024.

Michal Spiegel, Michal Štefánik, Marek Kadlčík, and Josef Kuchař. Attend or Perish: Benchmarking Attention in Algorithmic Reasoning. arXiv preprint arXiv:2503.01909, 2025.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced Transformer with Rotary Position Embedding. *Neurocomputing*, 568:127063, 2024.

Bohan Tang, Huidong Liang, Keyue Jiang, and Xiaowen Dong. On the Importance of Task Complexity in Evaluating LLM-Based Multi-Agent Systems. arXiv preprint arXiv:2510.04311, 2025.

Petar Veličković, Christos Perivolaropoulos, Federico Barbero, and Razvan Pascanu. Softmax is not Enough (for Sharp Size Generalisation). In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 61190–61211, 2025.

Yiwei Wu, Atticus Geiger, and Raphaël Millière. How Do Transformers Learn Variable Binding in Symbolic Programs? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 67284–67299, 2025.

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 *International Conference on Learning Representations*, 2024.

A TASK GENERATION AND REFERENCE COMPUTATIONS

Scans. BOS supplies the initial state: 0 for A and the identity permutation for B1 and B2. Permutations are represented by their images, with $(p \circ q)[k] = p[q[k]]$. In B2, token identity indicates whether an element is counted. Before each counted element, including the first, we insert $u \sim \mathcal{U}\{1, 2, 3\}$ decoys, independently valued uniformly in S_3. The previous counted state is therefore two to four positions earlier. A batch shares a decoy layout, with independently sampled values. Decoys receive no supervision and are excluded from evaluation. An adjacent scan that treats them as identity updates is also a valid solution.

Trees. Leaves are uniform in $\mathbb{Z}_{11}$ and operators are uniform in $\{+, -, \times\}$. To generate a tree of depth D, one branch is forced to reach D, choosing its direction uniformly at each level. Other eligible nodes expand with probability p. Training draws $p \sim \mathcal{U}(0.2, 1)$ per batch. Ragged evaluation fixes $p = 0.5$. Examples in a batch share a shape but have independent values and operators. This avoids padding. Every operator is supervised with its subtree value.

Recovering postfix operands. Let $h(j)$ be the stack height after token j:

$$h(0) = 0, \qquad h(j) = h(j-1) + \begin{cases} 1, & x_j \text{ is a leaf,} \\ -1, & x_j \text{ is a binary operator.} \end{cases} \tag{5}$$

For an operator at position i,

$$b_2(i) = i - 1, \qquad b_1(i) = \max\{j < i : h(j) = h(i)\}. \tag{6}$$

The right subtree ($b_2(i)$) ends immediately before its operator. After the left subtree ends, the stack is at height $h(i)$. Processing the right subtree keeps it strictly above this height until the operator at i. The left root ($b_1(i)$) is therefore the most recent earlier match. For example,

$$x = (3, 5, +, 9, 2, -, \times), \qquad h = (1, 2, 1, 2, 3, 2, 1).$$

At position 7, the operand roots are 3 and 6, with values 8 and 7. The result is 1 modulo 11.

Tree families. Left and right caterpillars have, respectively, a leaf as the right and left child of each internal node. Complete trees have equally deep subtrees at every internal node. Training

Table A.1: Postfix structure at depth $D \geq 1$, measured in internal nodes along the longest path. Left offset is $i - b_1(i)$.

Family	Operations	Tokens	Max. height	Max. left offset
Left caterpillar	D	$2D + 1$	2	2
Right caterpillar	D	$2D + 1$	$D + 1$	$2D$
Complete	$2^D - 1$	$2^{D+1} - 1$	$D + 1$	2^D

depth $D \leq 4$ permits at most 31 tokens, stack height 5, and left offset 16. At $D = 8$, right caterpillars exceed this height but remain within the offset bound; complete trees reach 511 tokens and offset 256. These bounds describe possible training examples, not their frequency. Left and right caterpillars are equally frequent in training, since the direction of the forced branch is uniform.

Execution with supplied dependencies. If leaves and node 0 are available initially, each update evaluates every node whose ordered operands are available, and completed values are preserved, then a depth-D DAG is evaluated in at most D updates. Indeed, an update extends the evaluated set from depth r to depth $r + 1$. This is a sufficient schedule, not a lower bound: alternative algorithms can use parallel shortcuts (Liu et al., 2023). We distinguish reference depth D from recurrent iterations R (of the model solving the task).

B MODEL AND EXPERIMENTAL PROTOCOL

Backbone. The small recurrent models follow Movahedi et al. (2026). FPRM applies a depthwise convolution at the start of every loop in all its experiments, but our base model omits it. The

convolutional variant, however, uses one in place of RoPE (Table B.3). Inputs sum type and value embeddings. Each layer uses causal attention and a SwiGLU MLP, both with pre-RMSNorm. A final RMSNorm and a shared linear head predict values at every position. For a block of L layers, residual sub-updates and input re-injection are

$$z \leftarrow a_1 z + b_1 f(\mathrm{RMSNorm}(z)), \tag{7}$$

$$z^{(r+1)} = a_2 z_{\mathrm{block}} + b_2 E(x), \qquad b_2 = 1 - a_2 a_1^{2L}, \qquad b_1 = \frac{b_2(1-a_1)}{1-a_1^{2L}}. \tag{8}$$

The learned coefficients $a_1, a_2 \in (0,1)$ are initialized at $0.75, 0.25$ and the initial state is sampled independently from the input as $z^{(0)} \sim \mathcal{N}(0, 0.5^2 I)$, following the practice from Movahedi et al. (2026).

Table B.1: Base configuration. Convolutional and oracle variants differ by fewer than $1{,}000$ parameters. Data are generated online, with depth sampled uniformly each step. All runs use the final checkpoint.

Setting	Value
Layers / heads / hidden width / MLP width	2 / 4 / 128 / 512
RoPE parameters: A / B1–B2 / C	526,852 / 528,648 / 528,397
Optimizer	AdamW, $\beta = (0.9, 0.95)$
Learning rate / weight decay / gradient clip	3×10^{-4} / 0.01 / norm 1.0
Schedule	1,000 warm-up steps, cosine decay to zero
Scans: training depths / steps / batch / seeds	1–8 / 40,000 / 64 / 3
Trees: training depths / steps / batch / seeds	1–4 / 20,000 / 256 / 5

Training and stopping. Scans use $n \sim \mathcal{U}\{0, \ldots, 16\}$ iterations without tracking the gradients, followed by four tracked iterations. The loss is the cross-entropy at the labeled positions in the final step (Bansal et al., 2022). The labeled positions are every counted scan state (decoys excluded) and every tree operator. Leaves are unlabeled. Trees use adaptive halting in windows of eight iterations: subtree loss is applied at each boundary, states are detached between windows, and gradients are averaged over windows. At evaluation, looped models use the adaptive halting rule below, except in the fixed-budget sweep (Appendix D.1). We denote by F the recurrent update, i.e $F(z) = F_\theta(z, E(x))$. The coefficients a_1, b_1, a_2, b_2 define the recurrent map F_θ. The adaptive procedure additionally uses a damping factor η to control the step towards its proposed next state. Unlike the learned residual coefficients, η is adjusted during iteration: damping preserves the map's fixed points. Evaluation (and tree training) uses damped updates $z \leftarrow \eta F(z) + (1-\eta)z$ and halt when

$$\max_{q \in \mathrm{batch}} \frac{\|z_q - F(z_q)\|_\infty}{\|F(z_q)\|_\infty} < 0.1, \tag{9}$$

or after 128 iterations. Damping starts at $\eta = 1$ and is multiplied by 0.9 after five non-improving steps, following the FPRM defaults.

Evaluation. Root accuracy measures the final counted scan state or final postfix value; uniform chance is 0.5, 0.17, and 0.09 for A, B1–B2, and C. Curves show means and one standard error across training seeds. Table B.2 specifies the evaluation draws. The different scan draws explain small baseline differences between Figures 2 and 4.

Compute and selection. Optimisation settings are fixed across corresponding variants and sizes, without a hyperparameter search. Selected-budget comparisons use separate selection and scoring splits. Huginn's depth-specific procedure is given in Appendix D.5. GPU jobs use one H100 80GB. Separate scan interventions and some evaluations use CPUs. The broader project used approximately 950 GPU-hours, including failed and superseded runs.

Architectural variants. Table B.3 distinguishes positional information from supplied dependencies. The convolution is causal, depthwise, and left-padded, with kernel size three. It is applied before each repeated block as $z \leftarrow z + \mathrm{conv}(z)$. A further control is provided in Figure D.5 to test the impact of changing the kernel size.

Table B.2: Main scans use shared evaluation seeds; scan interventions use each run's evaluation seed. Trees use shared seeds. Scan depths are 1–8, 12, 16, 24, 32; tree depths are 1–8.

Comparison	Samples per depth and training seed
Main scans	1,024 examples; one initial state per example. For B2: 4 layouts $\times 256$ examples.
Scan interventions	512 examples; for B2: 4 layouts $\times 128$ examples. Conditions use identical examples and states.
Trees	16 shapes $\times 16$ contents $\times 2$ initial states; shape fixed for deterministic families. Main and intervention comparisons use identical draws.

Table B.3: Oracles are trained from scratch. The tree oracle supplies operand roles and removes RoPE; it is therefore a structured execution control rather than a mask-only modification of the base model.

Variant	Position in attention	Access and additional structure
Base (RoPE)	Token-index RoPE	Causal attention
Convolution	None	Causal attention; in-loop convolution
Scan oracle	RoPE	Self and previous counted state
Tree oracle	None	Self and operand roots; learned per-head edge-role biases
Inserted mask	Retained RoPE	Self and operands; no added parameters or role labels
Inserted top-k	Retained RoPE	The k highest-scoring positions of each query, chosen by the model; no mask

B.1 Operand order and RoPE

Operand order. The operand mask (Equation (2)) gives a tree operator the positions of its operands, but not which is left and which is right, and subtraction depends on the order. With RoPE, relative position recovers the order: $b_2(i) = i - 1$ is always adjacent and $b_1(i)$ lies further back. Without a positional term the two operands are interchangeable, so the tree oracle adds one learned scalar per head for each edge role (self, left operand, right operand) to the attention logits. A scan position has a single operand, so the scan oracle needs no roles and keeps RoPE.

RoPE. A separate five-seed control keeps RoPE and trains with the operand mask alone, without role biases (Figure B.1). Given the operands, RoPE handles every structure whose offsets were seen in training: the control stays at 1.00 on right caterpillars through depth 8, where the base model falls to 0.13, and the scan oracle keeps RoPE and reaches 1.00 at depth 32. It fails only far beyond the training offsets: on complete trees it falls from 0.91 at depth 5 to 0.43 at depth 8, where left offsets reach 256 against at most 16 in training (Table A.1). We chose in the main text to represent the main tree oracle that replaces RoPE with role biases, so that it tests execution instead of RoPE extrapolation. The two oracles thus differ in both positional encoding and role information.

Removing RoPE does not remove the base model's failure. The convolutional model has no positional term in attention, yet on right caterpillars it falls from 1.00 at depth 4 to 0.12 at depth 8, and NoPE and FIRE also fail (Appendix D.3). The failure is therefore in locating operands, not in unseen RoPE offsets.

C Test-time interventions

Operand mask and top-k. The operand mask (Equation (2)) permits

$$\mathcal{M}_i = \begin{cases} \{i, b(i)\}, & i \text{ is a counted scan position,} \\ \{i, b_1(i), b_2(i)\}, & i \text{ is a tree operator,} \\ \{i\}, & i \text{ is BOS, a decoy, or a leaf.} \end{cases} \tag{10}$$

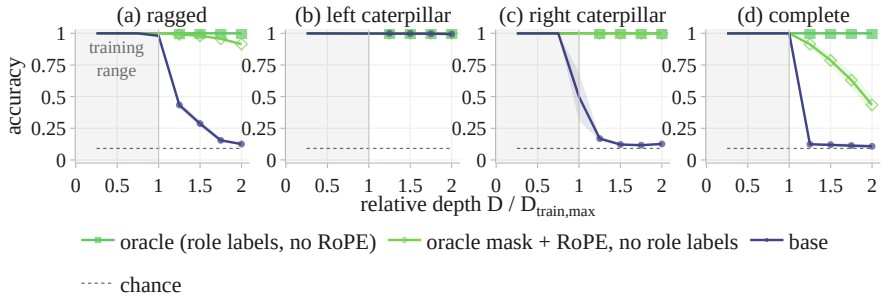

Figure B.1: **Tree oracle with RoPE and no role labels.** The operand mask with RoPE, the role-labeled oracle without RoPE, and the base model, on the four tree families. Accuracy at the budget selected on a held-out split, so base values differ slightly from Figure 2.

Disallowed logits are set to $-\infty$ before softmax in every head, layer, and iteration. Insertions retain the checkpoint's RoPE. The operand mask is fixed across iterations and initial-state draws. All intervention conditions are evaluated on paired examples and states. Top-k applies no mask: in every head, layer and iteration, each query keeps its k highest-scoring positions among itself and earlier positions under the checkpoint's own scores, with the softmax taken over those alone ($k = 2$ for scans, 3 for trees, the size of the correct mask at an operation; fewer where fewer exist). Weights are unchanged.

Adaptation. Adaptation under the operand mask starts from a checkpoint copy with a fresh AdamW state. It uses 50 steps, learning rate 3×10^{-5}, weight decay 0.01, and gradient clipping at 1. Scans use depths 1–8 and batch size 128. Trees use ragged examples at depths 1–4, $p = 0.5$, and batch size 64. Subtree or scan-state loss is differentiated through the full evaluation trajectory, unlike the truncated initial training. Each tree model is adapted once, then evaluated on all families. **No evaluation examples enter adaptation**.

Interventions experiments. The interventions of Section 4.1 show that which positions a model can attend to causally affects its accuracy. They do not show how the model locates operands internally: the mask also changes how attention weights are normalized and how the recurrent state evolves. Inserting the mask can also lower accuracy within the training range, on B2 (0.94 at depth 8) and markedly on A (0.56; Figure 4). Part of what adaptation repairs is therefore this mismatch between the learned computation and the new mask, not only the extrapolation failure.

D Additional experimental controls

Each control changes one aspect of the base model and is read beyond the training range, as in Table D.1: scans at depth 32, trees averaged over depths 5–8. Scan models use three seeds and tree models five, unless stated otherwise.

D.1 Recurrent budget

Figure 5 is read off a fixed-budget sweep: every checkpoint is run with adaptive halting disabled at each of 18 budgets from 1 to 128, on an evaluation sample separate from that of the main text. On the scans at depth 32, the oracle is at chance up to 16 iterations and at 1.00 from 64. The base model stops improving by 16 iterations on A and B1 (0.54 and 0.21 at 128). On B2 it peaks at 0.55 at 16 iterations and declines to 0.41 at 128, and the convolution from 0.41 to 0.34. On trees, all models saturate by 16 iterations: the oracle reaches 1.00, while the base and convolutional models plateau near 0.25 and 0.44. Separately, the scan checkpoints were run with the loop fixed at 128 and 512 iterations; these differ from adaptive halting by at most 0.02 and from each other by at most 0.01.

For each depth, $R^\star$ is the smallest budget of this sweep reaching 95% of the model's best accuracy at that depth. Figure D.1 gives $R^\star$ for every model and task. Where a model extrapolates, $R^\star$ grows

Table D.1: **Controls.** Root accuracy beyond the training range (scans at depth 32, trees averaged over depths 5–8). Rows are grouped by what differs from the base model. CoT, Huginn and the tree oracle differ in more than one aspect (Section 3). [a]Trees: best fixed budget in 1–128. [b]Best fitting size; convolution on A, B1 not scaled. [c]Pretrained 3.5B model, LoRA on our trees. [d]One size (1.05M). –: not run.

Changed	Model	A	B1	B2	C
–	Base	0.53	0.20	0.43	0.23
Iterations	more iterations[a]	0.54	0.21	0.41	0.25
Capacity	Base, best size[b]	0.98	0.97	0.73	0.26
	Conv., best size[b]	1.00	1.00	0.85	0.40
Position	No positional term	0.47	0.17	0.16	0.29
	FIRE	0.52	0.17	0.16	0.29
Recurrence	Fixed depth[b]	0.52	0.18	0.18	0.14
	CoT trace[d]	–	–	–	0.23
	Huginn, LoRA[c]	–	–	–	0.31
Access	Oracle	1.00	1.00	1.00	1.00
	Chance	0.50	0.17	0.17	0.09

with depth. Where it fails, however, $R^\star$ stops growing, because its best accuracy falls and is reached within a few iterations.

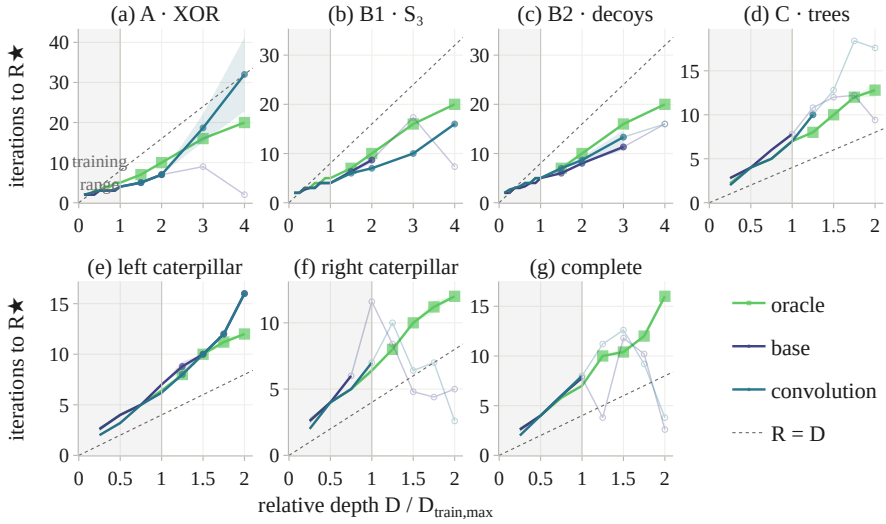

Figure D.1: **Iterations to $R^\star$.** $R^\star$ against relative depth $D/D_{\mathrm{train,max}}$, on the grid of Figure 2. Where the oracle extrapolates, $R^\star$ grows with depth; where the base model fails, it stops growing although budgets up to 128 are available. $R^\star$ is relative to each model's own best accuracy, so a low value for a failing model means its output stops changing early, not that it is accurate.

D.2 Capacity

The sweep varies the hidden width (64–512) and the number of layers in the repeated block (2, 4, 8), spanning 0.13M–34M parameters. Training steps, batch size and optimizer settings are those of Table B.1, so compute grows with size. Every scan configuration reaches at least 0.99 at training depth 8. On A and B1, width is what helps: at width 512 the base model reaches 0.98 and 0.97

at depth 32. On B2, the best sizes reach 0.73 (base) and 0.85 (convolution), and gains are not monotonic in size. The tree sweep is sparse: hollow markers denote sizes below 0.95 at training depth 4, whose extrapolation scores are not informative. Convolution on A and B1 already reaches 1.00 and was not scaled.

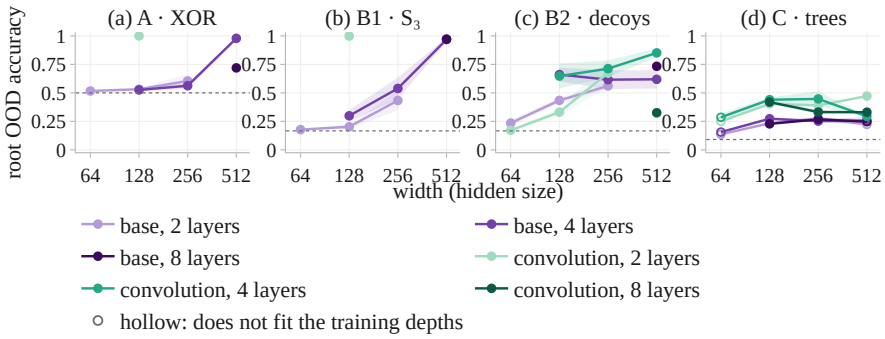

Figure D.2: Root accuracy beyond the training range against hidden width, one line per number of layers in the repeated block (darker: more layers). Hollow markers: insufficient fit to the training range.

D.3 Positional encodings

Two controls replace RoPE and keep everything else fixed. *No positional term* removes position from attention entirely, without the convolution. FIRE (Li et al., 2024) replaces RoPE with a causal per-head bias from a two-layer ReLU MLP of width 32, whose input is

$$\frac{\log(1+i-j)}{\max\{\log(1+i),\log(1+\ell)\}},$$

where ℓ is a learned floor. Both fit the training range and neither extrapolates better than RoPE. On the scans, both end at chance at depth 32. On trees, FIRE falls from 0.65 at depth 5 to 0.13 at depth 8, against 0.49 and 0.15 for RoPE.

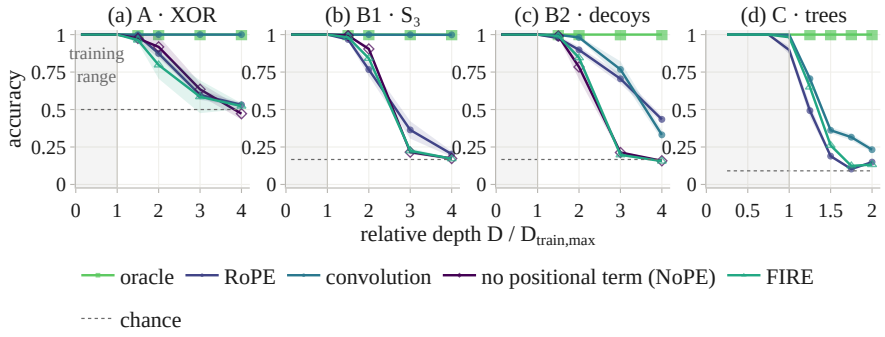

Figure D.3: Figure 2 (top row) with the two positional controls added.

Convolution width. On B2, the operand lies two to four positions back, and a causal kernel of width k reaches offsets up to $k-1$ in one iteration. Width 4 covers offsets 2 and 3 but leaves depth-32 accuracy unchanged (0.34 against 0.35 for width 3). Width 5, the first to cover every offset, does not reliably help (0.50 at depth 32) (Figure D.5).

D.4 FIXED-DEPTH AND CHAIN-OF-THOUGHT MODELS

Fixed depth. The layers of the repeated block are untied and applied once, keeping the embedding, RoPE attention, data, loss and parameter count. Across the same size grid (Figure D.4), this model stays at chance beyond the training range on every task, and most tree sizes do not fit the training range.

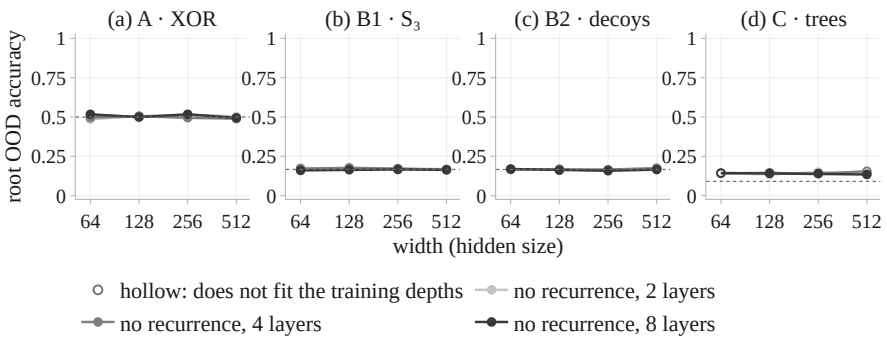

Figure D.4: As Figure D.2, for the fixed-depth model (no recurrence).

Chain of thought. We compare with chain of thought on ragged trees, the hardest task on the ladder, to test whether generating intermediate results as tokens extrapolates better than computing them in the loop. A causal decoder with the same backbone is trained with teacher forcing on an explicit trace, and its final value is scored after greedy decoding. The trace records one reduction at a time (Table D.2). A scratchpad variant rewrites the whole reduced expression at each step. Values, operators and separators each occupy one token. Sequences are capped at $1{,}200$ tokens, and decoding stops at EOS or the cap. CoT models use four layers and about 1.05M parameters, twice the base model, with three seeds. The trace model reaches 0.97 at depth 4, 0.59 at depth 5 and 0.10 at depth 8 whereas the scratchpad falls to 0.15 already at depth 5 (Figure D.6).

Table D.2: Tree CoT formats. The scratchpad rewrites the expression after reducing all ready subexpressions; the trace records one reduction at a time.

Format	Example (arithmetic modulo 11)
Prompt	`3 5 + 9 2 - *`
Scratchpad	`> 8 7 * > 1 <eos>`
Trace	`> 3 5 + = 8 > 9 2 - = 7 > 8 7 * = 1 <eos>`

D.5 OPEN-WEIGHT RECURRENT MODEL

We adapt the 3.5B-parameter `tomg-group-umd/huginn-0125` (Geiping et al., 2025) with LoRA, using postfix strings and loss at operator tokens (Table D.3). Predictions take the argmax over the eleven value symbols. At each depth, the recurrent budget is selected on half of that depth's evaluation set and scored on the other half. Each seed trained for about 48 H100-hours across checkpointed 24-hour jobs, and was stopped before the planned $20{,}000$ steps, so we evaluate the last checkpoints ($16{,}750$ and $17{,}500$ steps). By then the cosine schedule had reduced the learning rate below 7% of its peak, and training loss had stayed below 10^{-4} for over $4{,}000$ steps. Zero-shot, the model stays at chance. After adaptation it reaches 0.94 at depth 4, but falls to 0.57 at depth 5 and 0.16 at depth 8 (Figure D.7).

D.6 TEST-TIME COMPUTE

Figure 3 draws each model as a trajectory over tree depths 5–8, at the inference FLOPs it spent and the accuracy it reached. We count $2 \times (\text{parameters applied}) \times (\text{tokens processed})$, so a block repeated

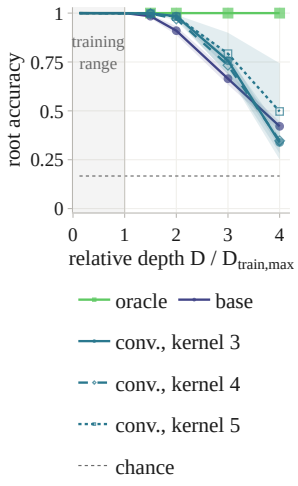

Figure D.5: B2 with convolution kernel 3, 4 and 5 (three seeds each).

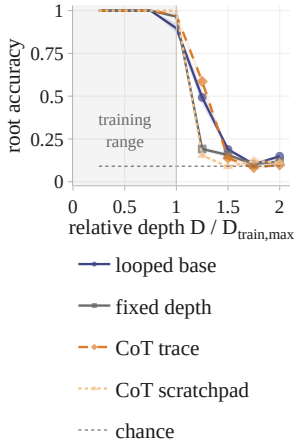

Figure D.6: Ragged trees: looped base model, fixed-depth model (best fitting size) and chain-of-thought trace and scratchpad.

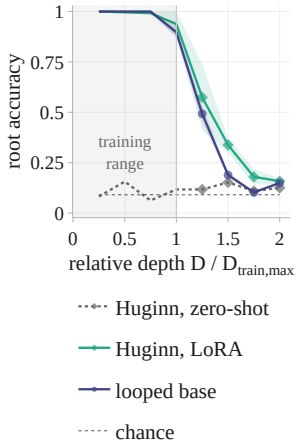

Figure D.7: Huginn on ragged trees, zero-shot and after LoRA, with the looped base model.

Table D.3: Huginn adaptation. Gradient accumulation gives the effective batch size. Results average the two available seeds.

Setting	Value
Data	Fixed 5,000 ragged trees per depth 1–4; $p = 0.5$
LoRA	Rank 32, $\alpha = 64$, dropout 0; all linear layers in prelude, core, and coda
AdamW	Learning rate 10^{-4}, $\beta = (0.9, 0.95)$, weight decay 0.01
Schedule / effective batch	200 warm-up steps, cosine decay / 256
Training recurrence	24 untracked + 8 tracked iterations
Completed steps	16,750 and 17,500 in two seeds (planned: 20,000)
Evaluation budgets	$\{4, 8, 16, 32, 64\}$

R times is charged R times its parameters, and Huginn's recurrent core four layers per recurrence. Attention-matrix FLOPs are omitted. Looped models are read at the smallest budget reaching 95% of their own best accuracy at that depth, so their values differ slightly from the adaptive-halting curves of Figure 2. CoT and fixed-depth models have no budget to choose. Huginn's token count uses our postfix length, a lower bound on its tokenized prompt.