ALTA: Compiler-Based Analysis of Transformers

Peter Shaw[1] **James Cohan**[2] **Jacob Eisenstein**[1] **Kenton Lee**[1]

Jonathan Berant[1] **Kristina Toutanova**[1]

[1] Google DeepMind [2] Google

Abstract

We propose a new programming language called ALTA and a compiler that can map ALTA programs to Transformer weights. ALTA is inspired by RASP, a language proposed by Weiss et al. (2021), and Tracr (Lindner et al., 2023), a compiler from RASP programs to Transformer weights. ALTA complements and extends this prior work, offering the ability to express loops and to compile programs to Universal Transformers, among other advantages. ALTA allows us to constructively show how Transformers can represent length-invariant algorithms for computing parity and addition, as well as a solution to the SCAN benchmark of compositional generalization tasks, without requiring intermediate scratchpad decoding steps. We make the ALTA framework — language specification, symbolic interpreter, and weight compiler — available to the community to enable further applications and insights.[1]

1 Introduction

There has been significant discussion and debate about the degree to which Transformers can perform compositional generalization and "System 2" reasoning, prompted by negative results on various evaluations for certain classes of Transformers (e.g., Dziri et al. (2023); Qiu et al. (2023); Shaw et al. (2021); Wu et al. (2024); Delétang et al. (2022); Mitchell et al. (2023); Valmeekam et al. (2023)). Do such negative results reflect some mutable aspect of how such models were trained, or more fundamental architectural limitations? To better understand the conceptual limitations of Transformers, it would be useful to have an interpretable framework for understanding whether and how Transformers can represent and learn solutions to various tasks of interest. Such a framework could also potentially help elucidate a path forward towards improving these capabilities.

We present a new framework for compiling interpretable, symbolic programs to Transformer model weights. The framework is based on a new programming language called ALTA, A Language for Transformer Analysis. The framework includes an interpreter for symbolically executing ALTA programs, and a compiler for converting ALTA programs to Transformer model weights. ALTA is inspired by prior work that introduced a programming language for Transformers called RASP (Weiss et al., 2021), and prior work that built a compiler from RASP programs to model weights called Tracr (Lindner et al., 2023). ALTA complements and extends this prior work, with two key conceptual differences.

First, ALTA supports dynamic control flow operations such as loops. While Zhou et al. (2023b) showed how RASP programs can be executed within the context of an auto-regressive decoder to implement some forms of loops by leveraging scratchpads (Nye et al., 2021; Wei et al., 2022), ALTA can implicitly support such operations without relying on intermediate decoding steps. This is useful

[1]Please see `https://arxiv.org/abs/2410.18077` for the latest version of this paper.

The 1st Workshop on System-2 Reasoning at Scale, NeurIPS 2024.

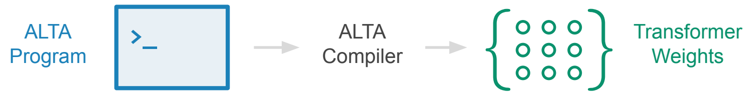

Figure 1: **Overview of ALTA.** We propose a new programming language called ALTA, and a "compiler" that can map ALTA programs to Transformer weights. ALTA is inspired by RASP, a language proposed by Weiss et al. (2021), and Tracr (Lindner et al., 2023), a compiler from RASP programs to Transformer weights. ALTA complements and extends this prior work, offering the ability to express loops and to compile programs to Universal Transformers, among other advantages.

to study because such additional decoding steps can be computationally inefficient and are non-differentiable, typically necessitating additional supervision. ALTA accomplishes this by compiling to Transformers with layer-wise weight sharing. From one perspective, Transformers with weight sharing are simply a special case of standard Transformers. However, they were also shown to have an inductive bias that improves performance on compositional tasks (Csordás et al., 2021; Ontanon et al., 2022; Yang et al., 2024), which warrants further study. ALTA also supports a conditional computation mechanism, enabling compilation of programs to Universal Transformers (Dehghani et al., 2019), thereby enabling new constructive expressivity results for this class of models.

Second, ALTA represents the computation of the MLP sub-layer as a sparse set of *transition rules*. We show that this enables compilation of complex programs to reasonably sized Transformers. In contrast, Tracr compiles functions over multiple variables expressed in RASP to dense lookup tables encoded in the MLP parameters, which can suffer from combinatorial explosion in the number of possible variable combinations. The ALTA compiler can leverage the sparsity expressed in the set of transition rules to reduce the number of MLP hidden dimensions required, in some cases by many orders of magnitude. Additionally, representing the MLP sub-layer computation as a set of sparse transition rules supports new insights into the generalization potential of MLP layers, and therefore of Transformers, which we explore both theoretically and empirically.

We highlight two primary applications of this framework. First, we show new constructive expressivity results for Transformers and Universal Transformers, including showing how Transformers can implement length-invariant algorithms for computing parity and addition, and a shift-reduce parsing algorithm that solves the SCAN (Lake & Baroni, 2018) benchmark of compositional generalization tasks. Second, we provide tools to analyze cases where the expressibility of an algorithm is established, but end-to-end training on a given training set fails to induce behavior consistent with the desired algorithm. Specifically, we propose to use intermediate supervision from ALTA execution traces over a given training set as a learning signal. In some cases, we show this additional supervision is sufficient to learn the desired algorithm, but in other cases failures can elucidate limitations of the underlying architecture due to, e.g., the type of positional encoding used. To complement this empirical assessment, we also introduce the analytical notion of whether a program is *minimal* with respect to a training set, based on whether certain components of a program could be removed without affecting the training set predictions. We provide theory showing that if a program is not minimal with respect to a training set, then the compiled model will contain parameters that can be freely changed without affecting any predictions on the training set, i.e. some parameters are under-specified by the training set, even with intermediate supervision. We demonstrate cases where this analysis predicts that test set performance would be under-specified by a given training set, and show agreement with empirical results from training with intermediate supervision. We hope these tools can help provide insights to bridge the gap between expressibility and learnability of algorithms in Transformers.

We make the ALTA framework, including the language specification, symbolic interpreter, and compiler from programs to Transformer weights, available to the community to support further applications and insights.

2 Proposed Framework

Here we give an overview of how ALTA programs are specified, their computational model, and how they can be compiled to Transformers. More details on the ALTA program API is in Appendix A.1, and compilation details and examples are in Appendix A.2.

```python
vars = {
    # Initialize parity with input.
    "parity": var(range=2, input_init_fn=lambda x: x),
    # Whether parity has been updated.
    "done": var(range=2, position_init_fn=lambda x: x == 0),
    # Position of current element.
    "idx": var(range=NUM_POS, position_init_fn=lambda x: x),
    # Index of preceding element.
    "idx_left": var(range=NUM_POS, position_init_fn=lambda x: max(0, x - 1),
}

attention_heads = {
    # Values of 'parity' and 'done' for preceding element.
    "parity_left": qkv("idx_left", "idx", "parity")
    "done_left": qkv("idx_left", "idx", "done")
}

def ffn_fn(z):
  if not z["done"] and z["done_left"]:
    # Update parity based on parity of preceding element.
    z["parity"] = z["parity_left"] ^ z["parity"]
    z["done"] = 1

return program_spec(vars=vars, heads=attention_heads, ffn_fn=ffn_fn,
                    output="parity", halt_spec=halt_spec("done", 1),
                    input_range=2, position_range=NUM_POS)
```

Figure 2: **Example ALTA Program.** The parity program computes whether a given binary sequence contains an even or odd number of "1" tokens. For an input of length N, the `parity` variable of the final input element will equal the parity of the overall sequence after $N-1$ layers, and computation will halt. The program specification contains all of the necessary information to compile the program to a Transformer.

2.1 Overview

We give an example of an ALTA program in Figure 2. An ALTA program specification includes three key components: a set of variables, a set of attention heads, and a "MLP function". We explain each of these below in the context of how they affect the execution of an ALTA program. The computational model of an ALTA program aligns closely with the computational model of a Transformer (Vaswani et al., 2017). In this paper we focus on *encoder-only* Transformers for simplicity.[2] We also focus on Transformers with layer-wise weight sharing, i.e. all attention and MLP parameters are shared across layers. We also support Universal Transformers which have an input-dependent number of layers. [3]

The ALTA framework includes an *interpreter*, which symbolically executes a program, and a *compiler* which compiles programs to Transformer weights. The input to an ALTA program $P \in \mathcal{P}$ is a sequence of integers inputs ($\in \mathcal{X}$) and the output is a sequence of integers of equal length ($\in \mathcal{Y}$). The interpreter implements a function $I : \mathcal{P} \times \mathcal{X} \rightarrow \mathcal{Y}$. The interpreter is useful for development and understanding the computational model in an abstract way. The compiler implements a function C such that $\theta = C(P)$ where $T(\mathbf{x}, \theta) \approx I(P, \mathbf{x})$ for all $\mathbf{x} \in \mathcal{X}$ and where T denotes the output of a Transformer encoder. The equality holds up to the limits of numerical approximation for well formed programs.

2.2 Variables

Similarly to Lindner et al. (2023), we adopt the *residual stream* view of Transformers as proposed by Elhage et al. (2021). In this view, the attention and MLP sub-layers within the Transformer read and

[2]However, we note that ALTA programs can alternatively be executed in the context of a *decoder-only* Transformer. This involves adding a causal attention mask and outer auto-regressive decoding loop, but does not otherwise affect the definition and compilation of ALTA programs.

[3]Notably, not all types of computation expressible by Transformers can be represented in ALTA, i.e., the range of the ALTA compiler is a relatively small subspace of all possible parameter values. For example, ALTA has limited support for numeric computation and does not support modeling of probabilistic output distributions. However, ALTA provides broad support for implementing various types of deterministic algorithms.

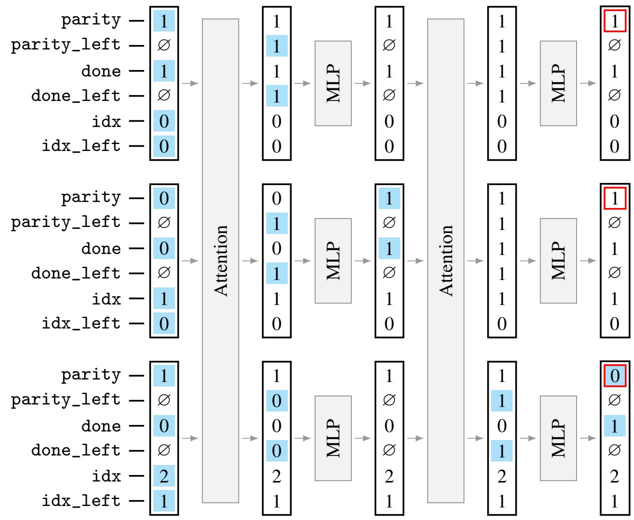

Figure 3: Visualization of the interpreter's symbolic residual stream for the parity program shown in Figure 2, for the input sequence $[1, 0, 1]$. The computed output sequence after the second layer is $[1, 1, 0]$, which corresponds to the parity of the input sequence up to the current position, with the final value containing the parity of the entire input sequence. Output values are outlined in red, and values that have changed are highlighted in blue.

write to the residual stream, which is represented by the activations between these sub-layers. While Transformers represent the residual stream for each element as a vector, our interpreter represents the residual stream for each element symbolically as a mapping of variables to values. The residual stream of the interpreter for the parity program of Figure 2 is shown in Figure 3. The set of variables and their possible values are specified by the program. There are three kinds of variables: categorical variables have bounded integer values, numerical variables have real-valued values, and set variables have sets of bounded integer values. [4] We establish a bijective mapping between variable assignments and activation vectors. Each possible value of a categorical variable is assigned a standard basis vector in the activation space, i.e., a one-hot encoding. Set variables are similarly represented, but with a multi-hot encoding. The scalar value of a numerical variable is directly represented in a single dimension.[5]

2.3 Execution

In this section, we explain the execution of an ALTA program in the interpreter, and summarize how each operation is encoded in a compiled Transformer, with more details in Appendix A. We denote the value of variable `foo` for element i at sub-layer k as $z^k_{\langle i,\texttt{foo}\rangle}$. Let $z^k_{\langle :,\texttt{foo}\rangle}$ denote the vector of values for `foo` across all elements at sub-layer k.

Initialization Given an input sequence $\mathbf{x} = \langle x_1, x_2, \cdots, x_{|\mathbf{x}|}\rangle$, we initialize every variable at every position. Specifically, for a variable `foo`, we initialize $z^0_{\langle i,\texttt{foo}\rangle}$ as a function of x_i, a function of the positional index i, or as a constant, based on how the initialization for `foo` is specified in the program. This operation is encoded in the parameters of the Transformer's embedding tables for input and position values. The number of possible input values and possible positional indexes must be specified to the compiler. Alternatively, positional embeddings can be omitted if no variable is initialized as a function of position.

Encoder Loop We then proceed to iteratively execute the self-attention sub-layer and the MLP sub-layer, which share parameters across all layers. A dynamic halting criteria, as proposed for

[4]Variables representing the output of attention heads can also take on a *null* or *undefined* value (§2.3)

[5]This mapping can be seen as establishing an approximate isomorphism with respect to the sub-layer operations of the interpreter and those of a compiled Transformer.

the Univeral Transformer (Dehghani et al., 2019), can optionally be specified by the program, which consists of specifying which variable and corresponding value indicate that computation has completed for a given element. Alternatively, a maximum number of layers can be specified as an argument to the interpreter or when running a compiled Transformer. Similarly to Tracr (Lindner et al., 2023), our compiled Transformers do not include layer normalization operations, which simplifies compilation. Otherwise, the self-attention and MLP sub-layers align with those of a standard Transformer, and are described below.

Self-Attention For each attention head, the interpreter computes a selection matrix, containing a "weight" for every pair of inputs, and uses this to aggregate information across positions. A simplifying assumption of ALTA, similarly to RASP, is that this matrix is binary. Attention heads in ALTA are specified by pointers to query, key, value, and output variables. Each head must have a unique output variable. The query variable must be a categorical or set variable, the key variable must be categorical, and the value and output variables must both be either categorical or numerical. For each attention sub-layer k, every attention head updates the value of some output variable:

$$z^{k+1}_{\langle :,\texttt{out}\rangle} = \texttt{aggregate}(\texttt{select}(z^k_{\langle :,\texttt{query}\rangle}, z^k_{\langle :,\texttt{key}\rangle}), z^k_{\langle :,\texttt{value}\rangle}),$$

where `query`, `key`, `value`, and `out` are the variable names specified by the given attention head. The definition of `select` is similar to that used by RASP.[6] The `select` operation returns a square selection matrix, $S_{i,j}$, where $S_{i,j} = [\![z^k_{\langle i,\texttt{key}\rangle} = z^k_{\langle j,\texttt{query}\rangle}]\!]$ if `query` is categorical and $S_{i,j} = [\![z^k_{\langle i,\texttt{key}\rangle} \in z^k_{\langle j,\texttt{query}\rangle}]\!]$ if `query` is set-valued. The `aggregate` operation returns a new sequence of values $z^{k+1}_{\langle :,\texttt{out}\rangle}$. Each value $z^{k+1}_{\langle i,\texttt{out}\rangle}$ is determined by aggregating over the set of *selected values*, $\{z^k_{\langle j,\texttt{value}\rangle} | S_{i,j} = 1\}$ specified by the selection matrix row $S_{i,:}$. When `value` is numeric, `aggregate` is defined the same as in RASP, outputting the average of the selected values, and will be undefined if no value is selected. When `value` is categorical, the output will be undefined if there is not exactly one value selected. [7]

These operations can be encoded in the parameters of the query, key, value, and output projections for a given attention head.[8] Each attention sub-layer is followed by a residual connection. We also support the option of using relative position representations (Shaw et al., 2018) by specifying a mask of relative positions that each attention head can attend to, which is applied to the selection matrix following the `select` operation. This binary mask is compiled to relative position biases using the parameterization of Raffel et al. (2020).

MLP Function The MLP sub-layer implements a mapping from a set of variable assignments to a new set of variable assignments, which is applied at every element. For compilation and analysis purposes, ALTA programs internally represent this operation as a set of *transition rules*, which can be interpreted as logical implications with conjunctive antecedents. For example, here is one of the transition rules for the example program in Figure 2:

$$z^{k+1}_{\langle \cdot,\texttt{parity}\rangle} = 1 \leftarrow z^k_{\langle \cdot,\texttt{done}\rangle} = 0 \ \wedge \ z^k_{\langle \cdot,\texttt{done\_left}\rangle} = 1 \ \wedge \ z^k_{\langle \cdot,\texttt{parity\_left}\rangle} = 1 \ \wedge \ z^k_{\langle \cdot,\texttt{parity}\rangle} = 0$$

When the *antecedent* of a rule is satisfied by the MLP input (i.e., all of the conditions hold with respect to the variable assignments at the MLP input), then the *consequent* determines the value of some *output variable* for the next sub-layer. [9] The set of transition rules can be specified in two

[6]The main difference is that we do not allow specifying a custom binary predicate as an argument to `select`. While this may seem to restrict the expressivity of the query, key, and value projections in the Transformer, we note that the MLP sub-layer prior to the attention sub-layer can set the query, key, and value variables to arbitrary values. By using set variables, it is still possible to specify arbitrary binary selection matrices. This choice simplifies compilation.

[7]The interpreter will raise an exception if any undefined variable is used as input to any operation in subsequent layers, as the encoding of an undefined variable is not well specified in a compiled model.

[8]A scalar hyperparameter controls the degree to which the softmax operation approximates generating a binary selection matrix.

[9]By construction, for a given output variable, there should never be more than one rule satisfied by the MLP input. If no rule is satisfied, then the value of that output variable is unchanged from the MLP input. We also include transition rules that ensure that every attention output variable is set to a *null* value so that it can be updated by the next attention sub-layer without conflicting with the residual connection. No attention output variable can otherwise be the output variable of any transition rule.

ways when defining ALTA programs. First, as shown in Figure 2, one can simply write a Python function with the signature shown. The set of transition rules can then be determined by executing this function for every possible set of variable assignments. In cases where this is not feasible, or where it is desirable to have more control over the set of transition rules, we offer an alternative API for specifying the set of transition rules more directly (see §A.1).[10] Leveraging the sparsity represented in a set transition rules rather than compiling a lookup table consisting of all variable combinations can significantly reduce the number of MLP dimensions required.[11]

The set of transition rules is represented in the parameters of the MLP layers. We generate a 4-layer MLP with clipped ReLU activations. The first 2 layers are only responsible for converting numerical and set variables into a one-hot representation, representing the possible values of these variables. For numerical variables, these correspond to a specified set of discrete buckets. Note that if a program contains only categorical variables, these 2 layers could be omitted. The final 2 layers of the MLP are based on the set of transition rules. The parameters of these layers are compiled such that the hidden activations are a binary vector where each value corresponds to whether a particular transition rule was satisfied by the MLP input. Each row of the first matrix is a function of the antecedent of a particular rule, and each column of the second matrix is a function of the consequent of a particular rule. Each MLP sub-layer is followed by a residual connection.

Output Each ALTA program specifies an output variable, which must be categorical. If execution terminates after k sub-layers, and the output variable is `output`, then the program returns $z^k_{\langle :, \texttt{output} \rangle}$. Selecting the subset of dimensions associated with the output variable is encoded in the parameters of the output projection. The Transformer then computes a softmax over this one-hot vector, and outputs the argmax.

3 Expressibility and Learnability

While there are many potential applications for ALTA, we focus on two applications in this paper: new constructive expressivity demonstrations, and analysis of whether such algorithms are learnable given a particular training set, with varying amounts of supervision. We give an overview of these applications here and provide additional discussion and details in Appendix B, with results in §4.

We show new constructive expressivity results for Transformers and Universal Transformers across several tasks in §4. However, even though we can establish that an algorithm is expressible by a given class of Transformers, training a model from this class on input and output examples of a particular algorithm can fail to induce a model that generalizes outside of the training set. It can be difficult to diagnose the reason for such failures. We provide two tools to help analyze the gap between expressibility and learnability.

First, we propose training with *trace supervision*, i.e., using intermediate supervision from ALTA execution traces over a given training set as a learning signal. In some cases, we show this additional supervision is sufficient to learn the desired algorithm, but in other cases failures can elucidate limitations of the underlying architecture due to, e.g., the type of positional encoding used. We report results on the parity task in §4, and details of the trace supervision procedure in §B.2.

Second, we introduce the analytical notion of whether a program is *minimal* with respect to a training set, based on whether certain components of a program could be removed without affecting the training set predictions. Given a program, training set, and test set, we can determine a *minimal version* of the program with respect to the training set (§B.3 describes the exact procedure used), and then assess whether this minimal version *generalizes*, i.e., has the same behavior as the original program on a test set. In §C, we provide a theoretical analysis focused on the MLP parameters in the trace supervision setting. In summary, for a given set of execution traces $\mathcal{D}$ and transition rules $\mathcal{R}$, we show that the compiled MLP parameters are a coordinate-wise local minima of a regularized reconstruction loss over $\mathcal{D}$ if and only if $\mathcal{R}$ is minimal with respect to $\mathcal{D}$ (see Theorems 1 and 2 for

[10] In either case, the representation of numerical variable values in the antecedent of a transition rule is based on the set of discrete buckets specified for the given variable.

[11] For example, consider a string x of length N represented by a set of categorical variables, x_1, x_2, $\cdots$, x_N, each with K possible values. We want to determine if x is in some vocabulary consisting of V strings. A naive lookup table approach requires K^N hidden dimensions, but this function can be represented with only V transition rules.

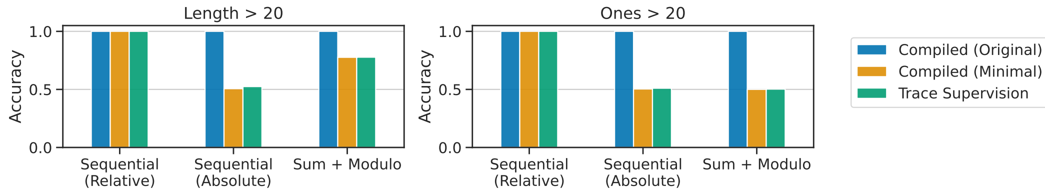

Figure 4: Generalization by length and number of ones for different parity programs in various settings. The training set consists of inputs of up to length 20 (and therefore up to 20 "ones") while the test set consists of inputs with lengths 21 to 40. While the original compiled models have perfect accuracy, the minimal versions of the programs exhibit different generalization behaviors and closely mirror the behaviors of the trace supervision experiments.

the formal statements and proofs). Our theoretical analysis suggests that a Transformer trained with trace supervision from a program on a training set is more likely to generalize if the minimal version of the program generalizes, and we evaluate this empirically in §4.

4 Experiments and Analysis

We detail experiments and analysis on several tasks, with further details and results in Appendix D.

Parity The parity task requires the model to compute whether a binary sequence contains an even or odd number of ones, and serves as a popular benchmark for analyzing Transformer generalization (Bhattamishra et al., 2020; Chiang & Cholak, 2022; Ruoss et al., 2023; Delétang et al., 2022; Hahn, 2020; Anil et al., 2022; Zhou et al., 2022, 2023b). We study three ALTA programs for computing parity detailed in §D.1. First, the *Sequential (Absolute)* program computes parity by iterating through each position (one per layer), flipping a parity bit every time a one is encountered. While this program uses absolute positions to enable propagating information to neighboring tokens, we also consider a *Sequential (Relative)* version that uses relative positions instead. Finally, the *Sum + Modulo* program computes parity in a single layer, using an attention head to compute the total number of ones, and then computing a mod 2 operation in the following MLP sub-layer.

For each program, Figure 4 compares the generalization behavior of Transformers compiled from the original version, compiled from the minimal version, and trained from trace supervision. The minimal versions of these programs exhibit different generalization behavior. The minimal *Sequential (Absolute)* does not generalize to examples longer than those seen during training, because the embeddings for positions not seen during training are not specified. The minimal *Sum + Modulo* program does not generalize to examples with more ones than those seen during training, as it does not contain transition rules related to the numerical values corresponding to larger numbers of ones than those seen during training. However, it can handle examples longer than the training examples if the numbers of ones were seen during training. Only the minimal *Sequential (Relative)* program generalizes to all input lengths. Notably, the trace supervision results exhibit the same generalization behavior as the minimal programs, as predicted in Section 3. The *Sequential (Relative)* program is also notable because it provides a constructive demonstration of how a Universal Transformer can express a length-invariant solution to parity with a finite set of parameters, without relying on intermediate decoding steps.[12]

We also evaluated several different Transformer variants using standard, end-to-end supervision. Theoretically, a transformer trained with weight sharing and at least as many layers as the longest examples in the train set could learn the minimal *Sequential (Relative)* algorithm and generalize to examples of length up to the number of layers. However, in practice all variants exhibit behavior similar to that of the minimal *Sum + Modulo* program, i.e., they exhibit some degree of length generalization but do not generalize to examples with more ones than were seen during training. See

[12] While Chiang & Cholak (2022) previously demonstrated how a Transformer can express a solution to parity for arbitrary lengths, their approach requires encoding activations and parameters with a degree of numerical precision that scales with the maximum input length. The *Sequential (Relative)* program does not have this limitation, but does require more layers of computation. Zhou et al. (2023b) also provide a length-invariant construction, but their approach requires intermediate decoding steps. (See §D.1 for details)

§D.1 for additional results and experiment details, and discussion of this result in relation to the MDL-inspired hypothesis of Zhou et al. (2023b), and related work attempting to characterize the inherent *simplicity bias* of Transformers.

Addition Another common benchmark for evaluating Transformer generalization has been multi-digit addition (Nogueira et al., 2021; Liu et al., 2022; Zhou et al., 2024; Shen et al., 2023; Zhou et al., 2023b; Lee et al., 2024; Kazemnejad et al., 2024; Ruoss et al., 2023). In §D.2 we detail an ALTA program with dynamic halting that can add two positive integers of unbounded size. This program compiles to a Universal Transformer with relative position representations. The number of layers required to compute the sum is $N + 2$, where N is the number of digits in the larger of the two inputs. Notably, the minimal version of this program with respect to a training set that includes only inputs with ≤ 3 digits can generalize to unbounded input lengths.

SUBLEQ SUBLEQ is a single instruction language that has been shown to be Turing-complete when given access to infinite memory (Mavaddat & Parhami, 1988). Giannou et al. (2023) previously showed how a Looped Transformer can implement an interpreter for a variant of SUBLEQ. In §D.3 we demonstrate an ALTA program for implementing an interpreter for a less restrictive version of SUBLEQ in a Universal Transformer.

SCAN The SCAN (Lake & Baroni, 2018) suite of compositional generalization tasks requires mapping natural language commands (e.g., "*jump twice*") to action sequences (e.g., `JUMP JUMP`). Certain train and test splits have been shown to be challenging for Transformer-based models (Keysers et al., 2020; Furrer et al., 2020; Qiu et al., 2022b; Kazemnejad et al., 2024). Empirically successful solutions have involved symbolic decompositions of some form (Shaw et al., 2021; Chen et al., 2020; Herzig & Berant, 2021; Qiu et al., 2022a; Zhou et al., 2023a). In §D.4, we demonstrate an ALTA program that solves the SCAN task and compile this program to a Transformer. First, the program executes a shift-reduce parse of the input sequence, representing the parse as a tree. Second, the ALTA program decodes the output sequence by traversing the parse tree. The program represents the necessary variable-length data structures (a stack, parse tree, and buffer) using a variable number of input tokens. Compiled models require $< 2,000$ MLP hidden dimensions despite there being $> 10^{60}$ possible variable combinations in the program, highlighting the importance of sparsity enabled by representing the MLP computation as a set of transition rules.

Notably, the *minimal version* of our program with respect to the training set generalizes to the test set, for all of the most challenging length-based and Maximum Compound Divergence (MCD) (Keysers et al., 2020) splits. Our ALTA program for SCAN thus gives a constructive demonstration of how Transformers can represent algorithms exhibiting systematic generalization: a finite set of transition rules and attention operations can be recombined in novel ways to process novel inputs.

5 Discussion

We have introduced the ALTA framework, and applied it to analyze the expressibility and learnability of various algorithms for Transformers. While this paper has focused on analysis related to expressibility and learnability, there are other potential applications of ALTA. For example, future work could explore more flexible methods for leveraging trace supervision as a learning signal, using ALTA to help develop interpretability tools, or developing models that combine compiled and learned components. Of course, the ALTA framework has some important limitations. In particular, the framework provides limited support for numerical computations and modeling probabilistic output distributions. Also, the properties of compiled models may not reflect those of models learned in practice. We offer an extended discussion of limitations and potential opportunities in Appendix E. We hope insights from ALTA can help the community better understand how Transformers can represent and learn various algorithms, and inspire new methods and techniques.

References

Emmanuel Abbe, Samy Bengio, Aryo Lotfi, and Kevin Rizk. Generalization on the unseen, logic reasoning and degree curriculum. In *International Conference on Machine Learning*, pp. 31–60. PMLR, 2023.

Ekin Akyürek, Bailin Wang, Yoon Kim, and Jacob Andreas. In-context language learning: Arhitectures and algorithms. *arXiv preprint arXiv:2401.12973*, 2024.

Dana Angluin, David Chiang, and Andy Yang. Masked hard-attention transformers and boolean rasp recognize exactly the star-free languages. *arXiv preprint arXiv:2310.13897*, 2023.

Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. *Advances in Neural Information Processing Systems*, 35:38546–38556, 2022.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

Satwik Bhattamishra, Arkil Patel, Varun Kanade, and Phil Blunsom. Simplicity bias in transformers and their ability to learn sparse boolean functions. In *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. *Advances in Neural Information Processing Systems*, 33: 1690–1701, 2020.

David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention. *arXiv preprint arXiv:2202.12172*, 2022.

David Chiang, Peter Cholak, and Anand Pillay. Tighter bounds on the expressivity of transformer encoders. In *International Conference on Machine Learning*, pp. 5544–5562. PMLR, 2023.

Noam Chomsky. *Syntactic Structures*. Mouton, Berlin, Germany, 1957.

Róbert Csordás, Kazuki Irie, and Juergen Schmidhuber. The devil is in the detail: Simple tricks improve systematic generalization of transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 619–634, 2021.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, et al. Neural networks and the Chomsky hierarchy. *arXiv preprint arXiv:2207.02098*, 2022.

P Kingma Diederik. Adam: A method for stochastic optimization. *(No Title)*, 2014.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, et al. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36, 2023.

Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. How can self-attention networks recognize dyck-n languages? In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 4301–4306, 2020.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, et al. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. URL `https://transformer-circuits.pub/2021/framework/index.html`.

Guhao Feng, Bohang Zhang, Yuntian Gu, Haotian Ye, Di He, and Liwei Wang. Towards revealing the mystery behind chain of thought: A theoretical perspective. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=qHrADgAdYu`.

Dan Friedman, Alexander Wettig, and Danqi Chen. Learning transformer programs. *Advances in Neural Information Processing Systems*, 36, 2023.

Dan Friedman, Abhishek Panigrahi, and Danqi Chen. Representing rule-based chatbots with transformers. *arXiv preprint arXiv:2407.10949*, 2024.

Daniel Furrer, Marc van Zee, Nathan Scales, and Nathanael Schärli. Compositional generalization in semantic parsing: Pre-training vs. specialized architectures. *arXiv preprint arXiv:2007.08970*, 2020.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pp. 11398–11442. PMLR, 2023.

Peter Grunwald. A tutorial introduction to the minimum description length principle. *arXiv preprint math/0406077*, 2004.

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Jonathan Herzig and Jonathan Berant. Span-based semantic parsing for compositional generalization. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 908–921, 2021.

Amirhossein Kazemnejad, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. The impact of positional encoding on length generalization in transformers. *Advances in Neural Information Processing Systems*, 36, 2024.

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. Measuring compositional generalization: A comprehensive method on realistic data. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=SygcCnNKwr`.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing, 2018. URL `https://arxiv.org/abs/1808.06226`.

Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pp. 2873–2882. PMLR, 2018.

Nayoung Lee, Kartik Sreenivasan, Jason D. Lee, Kangwook Lee, and Dimitris Papailiopoulos. Teaching arithmetic to small transformers. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=dsUB4bst9S`.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=rR03qFesqk`.

David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Tom McGrath, and Vladimir Mikulik. Tracr: Compiled transformers as a laboratory for interpretability. *Advances in Neural Information Processing Systems*, 36, 2023.

Ziming Liu, Ouail Kitouni, Niklas S Nolte, Eric Michaud, Max Tegmark, and Mike Williams. Towards understanding grokking: An effective theory of representation learning. *Advances in Neural Information Processing Systems*, 35:34651–34663, 2022.

Farhad Mavaddat and Behrooz Parhami. Urisc: the ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education*, 25(4):327–334, 1988.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations*, 2024.

Melanie Mitchell, Alessandro B Palmarini, and Arseny Moskvichev. Comparing humans, gpt-4, and gpt-4v on abstraction and reasoning tasks. *arXiv preprint arXiv:2311.09247*, 2023.

Richard Montague. Universal grammar. *Theoria*, 36(3):373–398, 1970.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

Michael Nielsen. A visual proof that neural nets can compute any function. *URL: http://neuralnetworksanddeeplearning. com/chap4. html*, 2016.

Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of transformers with simple arithmetic tasks. *arXiv preprint arXiv:2102.13019*, 2021.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Santiago Ontanon, Joshua Ainslie, Zachary Fisher, and Vaclav Cvicek. Making transformers solve compositional tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3591–3607, 2022.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021.

Jacob Pfau, William Merrill, and Samuel R Bowman. Let's think dot by dot: Hidden computation in transformer language models. *arXiv preprint arXiv:2404.15758*, 2024.

Linlu Qiu, Peter Shaw, Panupong Pasupat, Pawel Nowak, Tal Linzen, Fei Sha, and Kristina Toutanova. Improving compositional generalization with latent structure and data augmentation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4341–4362, 2022a.

Linlu Qiu, Peter Shaw, Panupong Pasupat, Tianze Shi, Jonathan Herzig, Emily Pitler, Fei Sha, and Kristina Toutanova. Evaluating the impact of model scale for compositional generalization in semantic parsing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 9157–9179, 2022b.

Linlu Qiu, Liwei Jiang, Ximing Lu, Melanie Sclar, Valentina Pyatkin, Chandra Bhagavatula, Bailin Wang, Yoon Kim, Yejin Choi, Nouha Dziri, et al. Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement. *arXiv preprint arXiv:2310.08559*, 2023.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986. *Biometrika*, 71(599-607):6, 1986.

Anian Ruoss, Grégoire Delétang, Tim Genewein, Jordi Grau-Moya, Róbert Csordás, Mehdi Bennani, Shane Legg, and Joel Veness. Randomized positional encodings boost length generalization of transformers. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1889–1903, 2023.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 464–468, 2018.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 922–938, 2021.

Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pp. 4596–4604. PMLR, 2018.

Ruoqi Shen, Sébastien Bubeck, Ronen Eldan, Yin Tat Lee, Yuanzhi Li, and Yi Zhang. Positional description matters for transformers arithmetic. *arXiv preprint arXiv:2311.14737*, 2023.

Lena Strobl, Dana Angluin, David Chiang, Jonathan Rawski, and Ashish Sabharwal. Transformers as transducers. *arXiv preprint arXiv:2404.02040*, 2024.

Hannes Thurnherr and Jérémy Scheurer. Tracrbench: Generating interpretability testbeds with large language models. In *ICML 2024 Workshop on Mechanistic Interpretability*, 2024.

Nikita Tsoy and Nikola Konstantinov. Simplicity bias of two-layer networks beyond linearly separable data. *arXiv preprint arXiv:2405.17299*, 2024.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of large language models-a critical investigation. *Advances in Neural Information Processing Systems*, 36:75993–76005, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

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

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 1819–1862, 2024.

Andy Yang and David Chiang. Counting like transformers: Compiling temporal counting logic into softmax transformers. *arXiv preprint arXiv:2404.04393*, 2024.

Liu Yang, Kangwook Lee, Robert D Nowak, and Dimitris Papailiopoulos. Looped transformers are better at learning learning algorithms. In *The Twelfth International Conference on Learning Representations*, 2024.

Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3770–3785, 2021.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=ByxRM0Ntvr`.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL `https://openreview.net/forum?id=WZH7099tgfM`.

Hattie Zhou, Azade Nova, Hugo Larochelle, Aaron Courville, Behnam Neyshabur, and Hanie Sedghi. Teaching algorithmic reasoning via in-context learning. *arXiv preprint arXiv:2211.09066*, 2022.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? a study in length generalization. *arXiv preprint arXiv:2310.16028*, 2023b.

Yongchao Zhou, Uri Alon, Xinyun Chen, Xuezhi Wang, Rishabh Agarwal, and Denny Zhou. Transformers can achieve length generalization but not robustly. *arXiv preprint arXiv:2402.09371*, 2024.

A Framework Details

In this section we provide additional details about the ALTA Framework, as introduced in §2. First, we detail the program API in §A.1, which is referenced by the ALTA programs in this paper. Second, we provide more details and examples of how programs are compiled to Transformer weights in §A.2.

A.1 Program API Details

Here we detail the functions of the ALTA API used to build the programs shown in this paper. We refer the reader to our open-source implementation for further details.

Variables and Attention Heads The module contains several methods for defining variable specifications:

- `var` defines a categorical variable, the most common variable type. The cardinality must be specified.
- `numerical_var` defines a numerical variable. A set of discrete buckets must be specified, and values are rounded to the closest bucket in the MLP sub-layer of compiled models.
- `set_var` defines a set variable. A set of sets of possible values must be specified.

For each of these functions, it is also necessary to specify how the variable is initialized.

There are two methods for defining attention heads:

- `qkv` defines an attention head, with arguments that specify the query, key, and value variables. Optionally, a set of relative positions can also be passed, as well as an explicit specification for the output variable.
- `relative_v` is a shorthand function for defining an attention head that attends to a relative position. They query and key are implicitly set to a single-valued categorical variable.

The output variable specification can be optionally specified explicitly, and is otherwise inferred from the type of the value variable. The output variable for each head is also included in the overall set of program variables.

```python
def get_transition_rules(variables, attention_heads):
  x = MLPBuilder(variables, attention_heads)
  for done in x.get("done"):
    if done != 1:
      for done_left in x.get("done_left"):
        if done_left == 1:
          x.set("done", 1)
          for parity_left, parity in x.get("parity_left", "parity"):
            x.set("parity", parity_left ^ parity)
  return x.rules
```

Figure 5: Function for directly specifying the set transition rules for the parity program of Figure 2.

MLP Functions There are two ways to specify the MLP function, as mentioned in §2. For simplicity, the programs listed in this paper specify the MLP function as a Python function, which is passed a dictionary-like object for accessing and updating variable values. Alternatively, the set of transition rules can be specified directly, allowing more control to manage the scope of variables included in the antecedent of each rule, and avoid the combinatorial explosion of possible variable values for more complex programs. Figure 5 gives an example of specifying the transition rules for the parity program of Figure 2 using the `MLPBuilder` class. The class has two methods. The `get` method returns a generator over possible variable values. The class automatically tracks which variables and variable values are in scope. The `set` method generates a transition rule, generating the antecedent of the rule automatically based on the current scope. In almost all cases, this results in a more compact set of transition rules than specifying the MLP function as a Python function. All of the results in this paper related to analyzing the minimal versions of programs or computing the number of MLP hidden dimensions in compiled models are based on versions of programs where the transition rule set has been specified directly.

A.2 Compiler Details

In this section we will give examples of the compilation process introduced in §2, using the parity program of Figure 2 as an example.

Notation Let v_i^k denote the activation vector in a compiled model for element i at sub-layer k.

Encoding Variable Assignments Consider a set of variable assignments:

$$\begin{aligned} \texttt{parity} &= 1 \\ \texttt{parity\_left} &= 0 \\ &\cdots \\ \texttt{idx} &= 2 \\ &\cdots \end{aligned}$$

For brevity, we consider only a subset of program variables here. This set of assignments is represented as the following vector in a compiled model:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \cdots \\ 0 \\ 0 \\ 1 \\ \cdots \end{bmatrix} \begin{matrix} \texttt{parity} = 0 \\ \texttt{parity} = 1 \\ \texttt{parity\_left} = 0 \\ \texttt{parity\_left} = 1 \\ \\ \texttt{idx} = 0 \\ \texttt{idx} = 1 \\ \texttt{idx} = 2 \\ \\ \end{matrix}$$

Initialization The input embedding is computed as: $z_i^0 = W^X_{x_i,:} + W^I_{i,:}$, where x_i is the input token ID as position i, and W^X and W^I represent the embedding matrices for token and positional embeddings, respectively. For brevity, we only consider only up to 3 positional indices. We also omit some variables in the matrices below, and transpose them for clarity of the row and column labels. Note that variables representing attention outputs, e.g. `parity_left`, are initialized to a *null* value.

$$(W^X)^\top = \begin{matrix} x_i = 0 & x_i = 1 \\ \end{matrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ \cdots & \cdots \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \cdots & \cdots \end{bmatrix} \begin{matrix} \texttt{parity} = 0 \\ \texttt{parity} = 1 \\ \texttt{parity\_left} = 0 \\ \texttt{parity\_left} = 1 \\ \\ \texttt{idx} = 0 \\ \texttt{idx} = 1 \\ \texttt{idx} = 2 \\ \\ \end{matrix} \qquad (W^I)^\top = \begin{matrix} i = 0 & i = 1 & i = 2 \\ \end{matrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \cdots & \cdots & \cdots \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \cdots & \cdots & \cdots \end{bmatrix} \begin{matrix} \texttt{parity} = 0 \\ \texttt{parity} = 1 \\ \texttt{parity\_left} = 0 \\ \texttt{parity\_left} = 1 \\ \\ \texttt{idx} = 0 \\ \texttt{idx} = 1 \\ \texttt{idx} = 2 \\ \\ \end{matrix}$$

Self-attention The self-attention operation sums over a set of attention heads:

$$z_i^{k+1} = z_i^k + \sum_h o_i^h,$$

where o_i^h is the output of head h parameterized by matrices W_h^Q, W_h^K, W_h^V, and W_h^O:

$$
\begin{aligned}
o_i^h &= W_h^O \sum_j \alpha_{ij} \cdot W^v z_i^k \\
l_{ij} &= (W_h^Q z_i^k)^\top W_h^K z_i^k \\
\alpha_{ij} &= \frac{e^{l_{ij}}}{\sum_k e^{l_{ik}}}.
\end{aligned}
$$

For simplicity, we ignore the scaled dot product term. Simlarly to Tracr (Lindner et al., 2023), we also adopt the parameterization of Elhage et al. (2021) for the attention output, which can be shown to be equivalent to that of the original Transformer, but allows for a clearer exposition.

The parity program of Figure 2 has two attention heads. Here we describe the W^Q, W^K, W^V, and W^O matrices corresponding to the attention head with query `idx_left`, key `idx`, value `parity`, and output `parity_left` (ommitted cells are zeros):

$$
(W^Q)^\top = \left[\begin{array}{ccc} \cdots & \cdots & \cdots \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{array}\right] \begin{array}{l} \\ \texttt{idx = 0} \\ \texttt{idx = 1} \\ \texttt{idx = 2} \\ \texttt{idx\_left = 0} \\ \texttt{idx\_left = 1} \\ \texttt{idx\_left = 2} \end{array}
\qquad
(W^K)^\top = \left[\begin{array}{ccc} \cdots & \cdots & \cdots \\ \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \begin{array}{l} \\ \texttt{idx = 0} \\ \texttt{idx = 1} \\ \texttt{idx = 2} \\ \texttt{idx\_left = 0} \\ \texttt{idx\_left = 1} \\ \texttt{idx\_left = 2} \end{array}
$$

$$
(W^V)^\top = \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ \cdots & \cdots \end{array}\right] \begin{array}{l} \texttt{parity = 0} \\ \texttt{parity = 1} \\ \texttt{parity\_left = 0} \\ \texttt{parity\_left = 1} \\ \\ \end{array}
\qquad
W^O = \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ \cdots & \cdots \end{array}\right] \begin{array}{l} \texttt{parity = 0} \\ \texttt{parity = 1} \\ \texttt{parity\_left = 0} \\ \texttt{parity\_left = 1} \\ \\ \end{array}
$$

where λ is a hyperparameter that controls the degree to which the selection matrix approximates a binary-valued matrix, set to 100 by default.

MLP Sub-layer Our approach to compiling MLP parameters takes loose inspiration from Nielsen (2016), which provides a helpful visual exposition of how MLPs can encode arbitrary functions. The MLP function of the parity program can be represented with 7 transition rules, using the notation of Section 2:

$$
\begin{aligned}
&R_1 : z^{k+1}_{\langle \cdot, \texttt{parity} \rangle} = 1 \leftarrow z^k_{\langle \cdot, \texttt{done} \rangle} = 0 \,\wedge\, z^k_{\langle \cdot, \texttt{done\_left} \rangle} = 1 \,\wedge\, z^k_{\langle \cdot, \texttt{parity\_left} \rangle} = 1 \,\wedge\, z^k_{\langle \cdot, \texttt{parity} \rangle} = 0 \\
&R_2 : z^{k+1}_{\langle \cdot, \texttt{parity} \rangle} = 0 \leftarrow z^k_{\langle \cdot, \texttt{done} \rangle} = 0 \,\wedge\, z^k_{\langle \cdot, \texttt{done\_left} \rangle} = 1 \,\wedge\, z^k_{\langle \cdot, \texttt{parity\_left} \rangle} = 1 \,\wedge\, z^k_{\langle \cdot, \texttt{parity} \rangle} = 1 \\
&R_3 : z^{k+1}_{\langle \cdot, \texttt{done} \rangle} = 1 \leftarrow z^k_{\langle \cdot, \texttt{done} \rangle} = 0 \,\wedge\, z^k_{\langle \cdot, \texttt{done\_left} \rangle} = 1 \\
&R_4 : z^{k+1}_{\langle \cdot, \texttt{parity\_left} \rangle} = \varnothing \leftarrow z^k_{\langle \cdot, \texttt{parity\_left} \rangle} = 0 \\
&R_5 : z^{k+1}_{\langle \cdot, \texttt{parity\_left} \rangle} = \varnothing \leftarrow z^k_{\langle \cdot, \texttt{parity\_left} \rangle} = 1 \\
&R_6 : z^{k+1}_{\langle \cdot, \texttt{done\_left} \rangle} = \varnothing \leftarrow z^k_{\langle \cdot, \texttt{done\_left} \rangle} = 0 \\
&R_7 : z^{k+1}_{\langle \cdot, \texttt{done\_left} \rangle} = \varnothing \leftarrow z^k_{\langle \cdot, \texttt{done\_left} \rangle} = 1
\end{aligned}
$$

Section C provides a detailed description of the MLP sub-layer and the compilation process. We just give the compiled parameters W^1, b^1, and W^2 for the parity program here. As described in Section A, there are also two initial MLP layers that are responsible for converting numerical and set variables into one-hot representations, but those are not necessary for the parity program we are considering which uses only categorical variables. We also omit the cells corresponding to `idx` and `idx_left` in the matrices and vector below, which are zeros.

$$
(W^1)^\top = \begin{array}{c}
\begin{matrix} R_1 & R_2 & R_3 & R_4 & R_5 & R_6 & R_7 \end{matrix} \\
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots
\end{bmatrix}
\end{array}
\begin{array}{l}
\\
\texttt{parity} = 0 \\
\texttt{parity} = 1 \\
\texttt{parity\_left} = 0 \\
\texttt{parity\_left} = 1 \\
\texttt{done} = 0 \\
\texttt{done} = 1 \\
\texttt{done\_left} = 0 \\
\texttt{done\_left} = 1 \\
\\
\end{array}
$$

$$
(b^1)^\top = \begin{array}{c}
\begin{matrix} R_1 & R_2 & R_3 & R_4 & R_5 & R_6 & R_7 \end{matrix} \\
\begin{bmatrix} -3 & -3 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}
\end{array}
$$

$$
W^2 = \begin{array}{c}
\begin{matrix} R_1 & R_2 & R_3 & R_4 & R_5 & R_6 & R_7 \end{matrix} \\
\begin{bmatrix}
-1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots
\end{bmatrix}
\end{array}
\begin{array}{l}
\\
\texttt{parity} = 0 \\
\texttt{parity} = 1 \\
\texttt{parity\_left} = 0 \\
\texttt{parity\_left} = 1 \\
\texttt{done} = 0 \\
\texttt{done} = 1 \\
\texttt{done\_left} = 0 \\
\texttt{done\_left} = 1 \\
\\
\end{array}
$$

MLP Example Consider a set of variable assignments:

$$
\begin{aligned}
\texttt{parity} &= 1 \\
\texttt{parity\_left} &= 1 \\
\texttt{done} &= 0 \\
\texttt{done\_left} &= 1 \\
&\cdots
\end{aligned}
$$

Let z denote the vector encoding of these assignments:

$$
z = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ \cdots \end{bmatrix}
\begin{array}{l}
\texttt{parity} = 0 \\
\texttt{parity} = 1 \\
\texttt{parity\_left} = 0 \\
\texttt{parity\_left} = 1 \\
\texttt{done} = 0 \\
\texttt{done} = 1 \\
\texttt{done\_left} = 0 \\
\texttt{done\_left} = 1 \\
\\
\end{array}
$$

Here are the hidden activations in the MLP layer, which is binary vector corresponding to which rules are satisfied by the input:

$$
h(z) = \sigma(W^1 z + b^1) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}
\begin{array}{l} R_1 \\ R_2 \\ R_3 \\ R_4 \\ R_5 \\ R_6 \\ R_7 \end{array}
$$

Here is the output of the MLP sub-layer before and after the residual connection:

$$W^2 h(z) = \begin{bmatrix} 1 \\ -1 \\ 0 \\ -1 \\ -1 \\ 1 \\ 0 \\ -1 \\ \dots \end{bmatrix} \begin{matrix} \texttt{parity} = 0 \\ \texttt{parity} = 1 \\ \texttt{parity\_left} = 0 \\ \texttt{parity\_left} = 1 \\ \texttt{done} = 0 \\ \texttt{done} = 1 \\ \texttt{done\_left} = 0 \\ \texttt{done\_left} = 1 \\ \end{matrix} \qquad W^2 h(z) + z = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ \dots \end{bmatrix} \begin{matrix} \texttt{parity} = 0 \\ \texttt{parity} = 1 \\ \texttt{parity\_left} = 0 \\ \texttt{parity\_left} = 1 \\ \texttt{done} = 0 \\ \texttt{done} = 1 \\ \texttt{done\_left} = 0 \\ \texttt{done\_left} = 1 \\ \end{matrix}$$

Which corresponds to the expected output assignments:

$$\begin{aligned} \texttt{parity} &= 0 \\ \texttt{parity\_left} &= \varnothing \\ \texttt{done} &= 1 \\ \texttt{done\_left} &= \varnothing \\ &\dots \end{aligned}$$

where the attention output variables are set to *null* values so they are ready to be updated at the next self-attention layer.

B Expressivity and Learnability Analysis Details

In this section, we provide additional discussion and details related to analyzing the expressibility of Transformers and the learnability of various algorithms, as introduced in §3. First, we discuss prior work related to Transformer expressivity and how ALTA can help provide new insights in §B.1. Second, we detail the procedure for training with trace supervision in §B.2. Third, we details the procedure for determining the minimal version of a program with respect to a training set in §B.3. Finally, we provide our theoretical analysis connecting the notion of minimality to properties of a loss landscape around the compiled MLP parameters in §C.

B.1 Expressivity Discussion

There has been considerable interest in establishing the theoretical expressivity of various classes of Transformers (Pérez et al., 2021; Chiang et al., 2023; Yun et al., 2020), including those with dynamic computation, with a recent focus on intermediate decoding steps (Feng et al., 2023; Merrill & Sabharwal, 2024). Of particular relevance to ALTA, while Pérez et al. (2021) show that Transformer encoder-decoders with hard attention, positional encoding and parameters and activations that are rational numbers of arbitrary precision are Turing-complete, they also show that even Universal Transformers with fixed precision are not Turning-complete. Theoretical works like this one often do not demonstrate specific constructions of how Transformers can express algorithms of interest with limited resources. Tools like RASP, Tracr, and ALTA make it easier to construct Transformers that express various algorithms than directly specifying specific sets of parameter values. For example, RASP and its extensions have supported many new constructive expressivity results for Transformers (Angluin et al., 2023; Kazemnejad et al., 2024; Yang & Chiang, 2024; Strobl et al., 2024; Friedman et al., 2024). In Section 4, we show that ALTA can enable new constructive demonstrations of how Transformers and Universal Transformers can express various algorithms, within some finite bounds on resources such as the sizes of parameter and activation matrices, and the number of layers of computation required.

B.2 Procedure for Training with Trace Supervision

Given a program P and a set of model inputs $\mathcal{X}$, we run P for every input in $\mathcal{X}$, and extract the variable assignments at the input and output of every sub-layer, for every position. These traces can be used to derive *intermediate supervision*, which encourages the behavior of the model to align with that of the program being used to provide supervision. In our experiments, for simplicity, we focus on training the MLP parameters. After using the ALTA interpreter to collect pairs of variable assignments at the input and output of the MLP layers, we map these assignments to vectors, using the mapping described in §2. Finally, we can train a MLP layer based on these pairs of input and output vectors. Specifically, we use an L2 loss to encourage the MLP to produce the output vector given the input. We then use the ALTA compiler to compile parameters other than those used for the MLP layer. By combining the learned MLP parameters with those provided by the compiler, we have a full set of Transformer parameters. See §D.1 for the hyperparameters and training details for parity task.

For future work, it may be possible to provide supervision from execution traces in a way that makes fewer assumptions about how variables are encoded in the residual stream, i.e. by encouraging the residual stream to encode such values in a way that allows them to be accurately predicted from the residual stream with a linear classifier. This would enable training models with intermediate supervision without any compiled parameters, but is out of scope for this work. It would then be interesting to assess whether there is any potential for transfer learning from training with such supervision in cases where a ground truth program is known to cases where a ground truth program is unknown or not feasible to express.

B.3 Procedure for Determining Minimal Versions of Programs

Given a program P and set of model inputs $\mathcal{X}$, we can determine the *minimal version* of P with respect to $\mathcal{X}$. For simplicity, we focus on the set of transition rules and the input embedding operations for this analysis. First, as motivated by our theoretical analysis in §C, we remove any transition rules which are never satisfied when running P on the inputs in $\mathcal{X}$. While the formal definition of a *minimal rule set* also puts conditions on the contraints of satisfied rules, we assume that there are no constraints to simplify our analysis. Second, we analyze the set of input IDs and positions seen when executing P over $\mathcal{X}$. We restrict the variable initialization functions of the minimal program to output default values for variables outside of this minimal set of observed token IDs and positions.

This analytical test has two advantages relative to training with trace supervision, with respect to understanding the potential learnability of a program from a given training set. It does not require training models or selecting hyperparameters, and provides an interpretable assessment of the potentially underspecified aspects of a program.

Notably, this notion of minimality can also help generalize some of the ideas introduced by Zhou et al. (2023b) with respect to RASP. As Zhou et al. (2023b) were interested in understanding length generalization, they proposed some restrictions on RASP programs that would otherwise be "difficult to learn". First, they proposed restrictions on programs containing certain operations over positional indices. Second, they excluded certain programs from consideration on an intuitive basis, such as a program for solving the parity task using sum and modulo operations. In both cases, such programs would not be minimal with respect to some finite length training set, according to our proposed criteria, as we show in Section 4.

C Theoretical Analysis of Minimal Rule Sets and MLP Parameters

Broadly, we are interested in how the criteria of whether a program is *minimal* with respect to some training set relates to the learnability of such a program with respect to that training set. In this section we explore this question from the perspective of the MLP parameters in the trace supervision setting. Specifically, we identify conditions under which the compiled MLP parameters are a strict coordinate-wise local optimum of a regularized reconstruction loss. Extensions to consider all parameters of the compiled transformer, end-to-end training objectives, or non-coordinate-wise local optima is left to future work.

C.1 MLP Specification

Here we introduce notation that differs from that in other sections, but is more appropriate for the goals of this section. Let us focus on two components of an ALTA program P for this analysis. First, a set of possible variable assignments, $\mathcal{V}$, and a set of rules, $\mathcal{R}$. The rules implicitly define an MLP function, $f_{\mathcal{R}} : \mathcal{V} \rightarrow \mathcal{V}$, which is a sub-component of the overall program. Let us also consider only programs with categorical variables for this analysis, and exclude transition rules related to attention outputs, for simplicity.

Variable Assignments An assignment $\mathbf{V} \in \mathcal{V}$ is a tuple of N_V elements, with $\mathbf{V} = \langle V_1, V_2, \cdots, V_{N_v}\rangle\rangle$, where $V_i \in \{0, 1, \ldots, D_i - 1\}$ and D_i is the dimensionality of variable V_i. The number and dimensionality of variables is specified by the program specification.

Transition Rules Let $\mathcal{R} = \langle R_1, R_2, \cdots, R_{N_R}\rangle$, where R_i is a transition rule. Transition rules can be represented as a logical implication where the antecedent, denoted R_i^a, has the following conjunctive form:

$$(k_1 = v_1) \wedge (k_2 = v_2) \wedge \cdots \wedge (k_{N_{R_i}} = v_{N_{R_i}}),$$

where each k_i refers to a variable and each v_i refers to some particular assignment of that variable. A transition rule is *satisfied* by an assignment $\mathbf{V} \in \mathcal{V}$ if and only if $V_k = v$ for all (k, v) in the antecedent, R_i^a. The consequent of a transition rule consists of a single *output variable*, denoted k_{R_i}, and a new assignment for that variable, denoted v'_{R_i}. Per construction, for every R_i we require that the output variable appears in the antecedent of the rule. Let v_{R_i} denote to the value associated with the output variable in the antecedent. The rule R_i can therefore be interpreted as updating the value of the variable k_{R_i} from v_{R_i} to v'_{R_i} if R_i is satisfied.

Finally, we also require that for any assignment $\in \mathcal{V}$, that no more than one rule is satisfied for a given output variable. In other words, if two rules share the same *output variable* index, then they should never both be *satisfied* for any assignment $\in \mathcal{V}$.

Logical MLP Function We can define the logical MLP function $g_R : \mathcal{Z} \to \mathcal{Z}$ in terms of $\mathcal{R}$. Let $\mathbf{V}' = f_R(\mathbf{V})$. Then:

$$V'_k = \begin{cases} v'_{R_i}, & \text{if } \exists R_i \in \mathcal{R} \text{ that is satisfied by } \mathbf{V} \\ V_k, & \text{otherwise.} \end{cases}$$

C.2 Compiling MLP Parameters

As described in §A.2, in our compiled models, we represent variable assignments as vectors, and the MLP function in the parameters of an MLP. Here we introduce notation and a more detailed description of the compilation procedure to support our theoretical analysis.

Assignment Embeddings In our compiled models, variable assignments are represented as N_Z-dimensional vectors. Let $\mathcal{Z} = \mathbb{R}^{N_Z}$ represent the space of such vectors, which include the inputs and outputs of MLP layers.

We can define a mapping $e_{\mathcal{V}} : \mathcal{V} \to \mathcal{Z}$. First, we define a bijective function $m(k, v)$ which for a given variable index $k \in \{0, 1, \ldots, N_v - 1\}$ and variable value v, returns an index in $\{0, 1, \ldots, N_Z - 1\}$. Let $\langle z_1, z_2, \cdots, z_{N_z}\rangle = e_{\mathcal{V}}(\mathbf{V})$. Then, $z_i = [\![\exists k, v \mid V_k = v \wedge m(k, v) = i]\!]$.

MLP Function The compiled MLP function $f_\theta : \mathcal{Z} \to \mathcal{Z}$ is defined by parameters $\theta \in \Theta$, where $\theta = \langle W^1, b^1, W^2\rangle$, and $W^1 \in \mathbb{R}^{N_Z \times N_R}$, $b^1 \in \mathbb{R}^{N_R}$, and $W^2 \in \mathbb{R}^{N_R \times N_Z}$. The bias term in the second layer is set to all zeros in practice, and omitted here for simplicity.

For $z \in \mathcal{Z}$, we can the define $f_\theta(z) = W^2 h_\theta(z) + z$ and $h_\theta(z) = \sigma(W^1 z + b^1)$, where $\sigma(z) = \max(0, \min(1, z))$ is a clipped ReLU activation function applied element-wise to the vector z.

MLP Parameters We can define the compiled parameters $\hat{\theta} \in \Theta$ such that $e_{\mathcal{V}}(f_R(\mathbf{V})) = f_{\hat{\theta}}(e_{\mathcal{V}}(\mathbf{V}))$ for all $\mathbf{V} \in \mathcal{V}$. Let $\hat{\theta} = \langle \hat{W}^1, \hat{b}^1, \hat{W}^2\rangle$.

$$\hat{W}^1_{i,j} = [\![\exists\, (k, v) \in R_i^a \text{ s.t. } m(k, v) = j]\!] \tag{1}$$

$$\hat{b}^1_i = 1 - N_{R_i} \tag{2}$$

$$\hat{W}^2_{i,j} = \begin{cases} -1, & \text{if } m(k_{R_j}, v_{R_j}) = i \\ 1, & \text{if } m(k_{R_j}, v'_{R_j}) = i \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

This construction ensures that for any assignment $\mathbf{V}$, we have $h_{\hat{\theta}}(e_{\mathcal{V}}(\mathbf{V}))[i] = [\![R_i \text{ is satisfied by } \mathbf{V}]\!]$. In other words, each element in the hidden layer activations corresponds to whether a particular rule was satisfied by the variable assignment represented in the input. Each column of $\hat{W}^2$ encodes an "update" to the assignments related to a particular rule.

C.3 Minimal Rule Set

A necessary condition for a *minimal program* is to have a *minimal rule set*, which describes minimality conditions relating only to the rule set and the corresponding MLP layer. This is a subset of the minimality conditions discussed in §B.3, encompassing only those conditions necessary to obtain guarantees about a reconstruction loss landscape around the parameters of the compiled MLP.

Definition 1. *(minimal rule set) Given a dataset of N variable assignments $\mathcal{D} \in \mathcal{V}^N$, a set of rules $\mathcal{R}$ is a **minimal rule set** over $\mathcal{D}$ if:*

- *It is not possible to remove any individual rule in $\mathcal{R}$ and not change the output of $f_{\mathcal{R}}$ for some example $\mathbf{V} \in \mathcal{D}$.*
- *It is not possible to remove any individual constraint in any rule in $\mathcal{R}$ and not change the output of $f_{\mathcal{R}}$ for some example $\mathbf{V} \in \mathcal{D}$.*

C.4 Setting

We consider a setting related to that of training models with trace supervision, as discussed in §B.2. For program P and set of model inputs $\mathcal{X}$, let $\mathcal{D} \in \mathcal{V}^N$ be the set of variable assignments at the input to the MLP for every position and layer when we run P over $\mathcal{X}$. The vector encodings of these variable assignments are denoted $\mathcal{Z}_{\mathcal{D}} = \{e_{\mathcal{V}}(\mathbf{V}) \mid \mathbf{V} \in \mathcal{D}\}$, with each $e_{\mathcal{V}}(\mathbf{V}_n)$ indicating the vertical concatenation of the one-hot encodings of all variables in $\mathbf{V}_n$.

Now let us define a *reconstruction loss* over individual predictions, $L_R(Z_n, \hat{Z}_n) \mapsto \mathbb{R}_+$, which quantifies how well the predicted variable encodings $\hat{Z}_n$ match the variable encodings Z_n specified by R. With some abuse of notation we will also use the shorthand $L_R(\Theta; \mathbf{V}_n) := L_R(e_{\mathcal{V}}(f_R(\mathbf{V}_n)), f_\theta(e_{\mathcal{V}}(\mathbf{V}_n)))$. Note that by construction of $e_{\mathcal{V}}$, we are guaranteed that $Z_i \in \{0,1\}^{N_Z}$. We will require that L_R increases at a rate of at least β as each component of $\hat{Z}$ moves away from Z:

Definition 2. *A loss $L_R(Z, \hat{Z}) \mapsto \mathbb{R}_+$ is a β**-reconstruction loss** for $\beta > 0$ if for each $\hat{Z}_i < Z_i$, the partial derivative is $\frac{\partial L_R}{\partial \hat{Z}_i}(\hat{Z}) \leq -\beta$ and for each $\hat{Z}_i > Z_i$, the partial derivative is $\frac{\partial L_R}{\partial \hat{Z}_i}(\hat{Z}) \geq \beta$.*

For example, for the L_1 loss $||Z - \hat{Z}||_1$ is a β-reconstruction loss with $\beta = 1$.

Finally, as it is difficult to analyze whether a point is a strict local optimum, we will instead analyze whether a point is a coordinate-wise local optimum.

Definition 3. *(coordinate-wise local optimum) For a multivariate function $f : \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_n \to \mathbb{R}$, the value $\hat{X} = [\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n]$ is a strict coordinate-wise local optimum iff for each i, the right derivative with respect to X_i evaluated at $\hat{X}$ is positive, $\frac{\partial_+ f}{\partial X_i}(\hat{X}) > 0$, and the left derivative with respect to X_i evaluated at $\hat{X}$ is negative $\frac{\partial_- f}{\partial X_i}(\hat{X}) < 0$.*

C.5 Main theorems

Suppose we learn Θ by optimizing a regularized sum of L_R losses over a dataset,

$$L(\Theta; \mathcal{D}, \alpha) = \alpha L_1(\Theta) + \sum_i L_R(\Theta; V_i), \tag{4}$$

with $\alpha > 0$ penalizing the sum of L_1 norms $||W^2||_1 + ||W^1||_1 + ||b^1||_1$. We want to show that when the weights $\hat{\Theta}$ are generated by compiling a program that is minimal with respect to $\mathcal{D}$, then $\hat{\Theta}$ is a coordinate-wise local optimum of L for $\alpha < \beta$.

Theorem 1. *Let $L(W^1, b^1, W^2) = \alpha L_1(W^1, b^1, W^2) + \sum_n^{\mathcal{D}} L_R(W^1, b^1, W^2; \mathbf{V}_n)$ be a loss such that L_R is a β-reconstruction loss summed over dataset $\mathcal{D}$ and $\alpha \in (0, \beta)$ is the coefficient on an element-wise L_1 regularizer. If $\langle \hat{W}^1, \hat{b}^1, \hat{W}^2 \rangle$ is the compilation of a rule set $\mathcal{R}$ that is minimal for $\mathcal{D}$, then $\langle \hat{W}^1, \hat{b}^1, \hat{W}^2 \rangle$ is a strict coordinate-wise local optimum of $L(W^1, b^1, W^2)$.*

Theorem 2. *If $\langle \hat{W}^1, \hat{b}^1, \hat{W}^2 \rangle$ is the compilation of a rule set $\mathcal{R}$ that is* not *minimal for $\mathcal{D}$, then $\langle \hat{W}^1, \hat{b}^1, \hat{W}^2 \rangle$ is* not *a strict coordinate-wise local optimum of any $L(W^1, b^1, W^2)$ as defined in Theorem 1.*

C.6 Proof of Theorem 1

To prove Theorem 1, we will prove lemmas for each group of parameters. To review the notation,

$$f(Z_n^{\text{in}}; \theta) = \hat{Z}_n^{\text{out}} = W^2 h(Z_n^{\text{in}}; \theta) + Z_n^{\text{in}} \tag{5}$$

$$h(Z_n^{\text{in}}; \theta) = \sigma(g(Z_n^{\text{in}}; \theta)) \tag{6}$$

$$g(Z_n^{\text{in}}; \theta) = W^1 Z_n^{\text{in}} + b^1, \tag{7}$$

with $\sigma(x) = \max(0, \min(x, 1))$. We then have the following partial derivatives,

$$\frac{\partial f_i}{\partial W_{i,j}^2} = h_j(Z_n^{\text{in}}; \theta) \qquad \frac{\partial f_i}{\partial h_j} = W_{i,j}^2 \tag{8}$$

$$\frac{\partial h_i}{\partial g_i} = [\![g_i(Z_n^{\text{in}}; \theta) \in (0,1)]\!] \qquad \frac{\partial h_i}{\partial g_{j \neq i}} = 0 \tag{9}$$

$$\frac{\partial g_i}{\partial b_j^1} = [\![i = j]\!] \qquad \frac{\partial g_i}{\partial W_{i,j}^1} = Z_{n,j}^{\text{in}}. \tag{10}$$

Lemma 1. *Each $\hat{W}_{i,j}^2$ is a strict coordinate-wise local optimum of L.*

Proof. We first distribute and then apply the chain rule to L_R,

$$\frac{\partial L}{\partial W_{i,j}^2} = \alpha \text{sign}(W_{i,j}^2) + \sum_n \frac{\partial}{\partial W_{i,j}^2} L_R(\theta; \mathbf{V}_n) \tag{11}$$

$$\frac{\partial}{\partial W_{i,j}^2} L_R(\theta; \mathbf{V}_n) = \frac{\partial L_R(\theta; V_n)}{\partial \hat{Z}_{n,i}^{\text{out}}} \frac{\partial \hat{Z}_{n,i}^{\text{out}}}{\partial W_{i,j}^2} \tag{12}$$

$$= \frac{\partial L_R(\theta; \mathbf{V}_n)}{\partial \hat{Z}_{n,i}^{\text{out}}} h_j(Z^{\text{in}}; \theta). \tag{13}$$

The term $h_j(Z_n^{\text{in}}) \in \{0, 1\}$ indicates whether rule j is satisfied by $\mathbf{V}_n$. By minimality, there exists at least one such n where $h_j(Z_n^{\text{in}}) = 1$; we can ignore all other n. Because L_R is a β-reconstruction loss and we are guaranteed that $\hat{Z}_n^{\text{out}} = Z_n^{\text{out}}$ for the compiled parameters, then the directional derivatives of the reconstruction loss must satisfy $\frac{\partial_- L_R(\theta; \mathbf{V}_n)}{\partial \hat{Z}_{n,i}^{\text{out}}} \leq -\beta$ and $\frac{\partial_+ L_R(\theta; \mathbf{V}_n)}{\partial \hat{Z}_{n,i}^{\text{out}}} \geq \beta$. Because $\alpha < \beta$, the regularizer cannot change the sign of the directional derivatives, so the conditions of strict coordinate-wise local optimality are satisfied. □

Lemma 2. *Each $\hat{b}_j^1$ is a strict coordinate-wise local optimum of L.*

Proof. We can reason about the effect of shifting $\hat{b}_j^1$ by ϵ by exploiting the properties of the nonlinearity $\sigma(x) = \min(1, \max(0, x))$:

$$\hat{Z}_{n,i}^{\text{out}} \Big|_{b_j^1 := \hat{b}_j^1 + \epsilon} = Z_{n,i}^{\text{in}} + W_{i,j}^2 \sigma(W_j^1 Z_n^{\text{in}} + b_j^1 + \epsilon) + \sum_{j' \neq j} W_{i,j'}^2 \sigma(W_{j'}^1 Z_n^{\text{in}} + b_{j'}^1) \tag{14}$$

$$= \begin{cases} \hat{Z}_{n,i}^{\text{out}} + W_{i,j}^2 \epsilon, & (Z_{n,i}^{\text{out}} = 0 \wedge W_{i,j}^2 \epsilon > 0) \vee (Z_{n,i}^{\text{out}} = 1 \wedge W_{i,j}^2 \epsilon < 0) \\ \hat{Z}_{n,i}^{\text{out}}, & \text{otherwise.} \end{cases} \tag{15}$$

Now we can evaluate $\hat{Z}_{n,i}^{\text{out}} \Big|_{b_j^1 := \hat{b}_j^1 + \epsilon}$ under various conditions:

- Suppose all conditions of rule j have been met and $W_i j \neq 0$. Then when $Z^{\text{out}}_{n,i} = 0$, the rule can only have affected the output by flipping $Z^{\text{in}}_{n,i}$ from 1 to 0, so we know $W^2_{i,j} = -1$; when $Z^{\text{out}}_{n,i} = 1$, the rule can only have affected the output by flipping $Z^{\text{in}}_{n,i}$ from 0 to 1, so we know $W^2_{i,j} = 1$. Either way, we can conclude $\epsilon < 0$, so when the conditions of rule j are met then $\frac{\partial_- \hat{Z}^{\text{out}}_{n,i}}{\partial \hat{b}^1_j} = 1$ and $\frac{\partial_+ \hat{Z}^{\text{out}}_{n,i}}{\partial \hat{b}^1_j} = 0$.
- If $W_i j \neq 0$ and all but one of the conditions of rule j are met, then when $Z^{\text{out}}_{n,i} = 0$, the MLP could affect the output by flipping $Z^{\text{in}}_{n,i}$ from 0 to 1, so we know $W^2_{i,j} = 1$; when $Z^{\text{out}}_{n,i} = 0$, the MLP could affect the output by flipping $Z^{\text{in}}_{n,i}$ from 1 to 0, so we know $W^2_{i,j} = -1$. Either way, we can conclude $\epsilon > 0$, so when all but one of the conditions of rule j are met then $\frac{\partial_+ \hat{Z}^{\text{out}}_{n,i}}{\partial \hat{b}^1_j} = 1$ and $\frac{\partial_- \hat{Z}^{\text{out}}_{n,i}}{\partial \hat{b}^1_j} = 0$.
- If $W_i j = 0$ or if two or more conditions of rule j are unmet then $\frac{\partial_+ \hat{Z}^{\text{out}}_{n,i}}{\partial \hat{b}^1_j} = \frac{\partial_- \hat{Z}^{\text{out}}_{n,i}}{\partial \hat{b}^1_j} = 0$.

By minimality we are guaranteed that for each rule j, there exists: (a) an index i such that $W_{ij} \neq 0$; (b) an example $\mathbf{V}_n$ such that all conditions of j are met; (c) an example $\mathbf{V}_{n'}$ such that all but one conditions are met. If (a) or (b) were not satisfied, we could drop the rule; if (c) were not satisfied we could drop one of the conditions. Now, by the chain rule and the definition of a β-reconstruction loss, these examples contribute $-\beta$ to the left derivative and β to the right derivative of L_R. Because $\beta > \alpha$, the signs of the left and right derivatives are not affected by the regularizer. They remain negative and positive respectively, satisfying the definition of coordinate-wise local optimality. □

Lemma 3. *$\hat{W}^1_{j,k}$ is a coordinate-wise local optimum of L.*

Proof. The analysis is similar to that of $\hat{b}^1_j$:

$$\frac{\partial}{\partial W^1_{j,k}} L_R(\theta; \mathbf{V}_n) = \sum_i \frac{\partial L_R(\theta; \mathbf{V}_n)}{\partial \hat{Z}^{\text{out}}_{n,i}} \frac{\partial \hat{Z}^{\text{out}}_{n,i}}{\partial h_j} \frac{\partial h_j}{\partial g_j} \frac{\partial g_j}{\partial W^1_{j,k}} \tag{16}$$

$$= \sum_i \frac{\partial L_R(\theta; \mathbf{V}_n)}{\partial \hat{Z}^{\text{out}}_{n,i}} \times \hat{W}^2_{i,j} \times [\![g_j(Z^{\text{in}}_n; \theta) \in (0,1)]\!] \times Z^{\text{in}}_{n,k} \tag{17}$$

$$= Z^{\text{in}}_{n,k} \times [\![g_j(Z^{\text{in}}_n; \theta) \in (0,1)]\!] \sum_i \frac{\partial L_R(\theta; \mathbf{V}_n)}{\partial \hat{Z}^{\text{out}}_{n,i}} \times \hat{W}^2_{i,j} \tag{18}$$

$$= Z^{\text{in}}_{n,k} \times \frac{\partial}{\partial b^1_j} L_R(\theta; \mathbf{V}_n). \tag{19}$$

Recall that $W^1_{j,k} \in \{0,1\}$ and $Z^{\text{in}}_{n,k} \in \{0,1\}$. If $W^1_{j,k} = 1$ then by minimality there exists some n such that $Z^{\text{in}}_{n,k} = 1$. In this case, $\frac{\partial}{\partial W^1_{j,k}} L_R(\theta; V_n) = \frac{\partial}{\partial b^1_j} L_R(\theta; V_n)$, which was already shown to satisfy the conditions of strict coordinate-wise local optimality in Lemma 2. If $W^1_{j,k} = 0$ then the left and right directional derivatives are set by the regularizer to $(-\alpha, \alpha)$. □

C.7 Proof of Theorem 2

Proof. There are two ways in which a ruleset might violate minimality with respect to $\mathcal{D}$.

- **We can remove rule R_j without affecting the output of $f_{\mathcal{R}}$ on any example.** All rules affect the output when their conditions are met, so we can infer that the conditions of R_j are never met. In this case, there is guaranteed to be a parameter $\hat{W}^2_{i,j} \neq 0$, setting the value of $\hat{Z}^{\text{out}}_i$ when the conditions of R_j are met. If R_j is never active on $\mathcal{D}$, then it does affect the reconstrution loss L_R.

- **There is a condition of R_j that we can remove without affecting the output of $f_{\mathcal{R}}$ on any example.** This condition is represented as a nonzero element in $\hat{W}^1_{j,k}$ (and also in b_j, which is not necessary for the proof). By construction this parameter does not affect the reconstruction loss.

In both cases, the gradient with respect to the nonzero parameter ($\hat{W}^2_{i,j}$ and $\hat{W}^1_{j,k}$) is set only by the regularizer. The right and left derivatives have the same sign, violating the conditions of strict coordinate-wise local optimality. □

D Program Details and Additional Results

D.1 Parity

Background The ability of transformers to learn parity has been studied extensively (Hahn, 2020), particularly the degree to which they exhibit length generalization (Bhattamishra et al., 2020; Chiang & Cholak, 2022; Ruoss et al., 2023; Delétang et al., 2022). Empirically successful solutions have relied on scratchpads (Anil et al., 2022; Zhou et al., 2022). Zhou et al. (2023b) used a variant of RASP to investigate why Transformers struggle with length generalization on parity and why scratchpads help.

Programs Here we provide the ALTA code for the parity programs that we study. The code for the *Sequential (Relative)* program is in Figure 7 and the code for the *Sum + Modulo* program is in Figure 6.

```
vars = {
    "parity": var(range=2),
    "start": numeric_var(input_init_fn=lambda x: float(x == START),
                         values=(0, 1)),
    "start_or_one": var(range=2, input_init_fn=lambda x: x in {1, START}),
    "query": var(range=2, default=1),
}

attention_heads = {
    "x": qkv("query", "start_or_one", "start",
             output_spec=numeric_var(values=BUCKETS)),
}

def ffn_fn(z):
  num_ones = round(1 / z["x"]) - 1
  z["parity"] = int(num_ones % 2 != 0)

return program_spec(
    variables=vars, heads=attention_heads, ffn_fn=ffn_fn,
    output_name="parity", input_range=3, position_range=None
)
```

Figure 6: Program for computing parity using a sum and modulo operation.

Alternative Programs There is another algorithm for parity proposed by Chiang & Cholak (2022), which was inspired by algorithms for MLPs from Rumelhart et al. (1986). Similarly to the Sum + Modulo program, this algorithm first computes a sum operation using an attention head, but they avoid explicitly computing the modulo operation by instead using attention heads that separately attend to even and odd elements. This enables the algorithm to be implemented with a fixed set of parameters that is invariant to the maximum input length, aside from the positional encodings. However, we did not explore this algorithm because it is not possible to implement in a way where the minimal ALTA program is invariant to the maximum input length considered. This is because ALTA requires discretization of numerical variables in order to perform numerical computations, a current limitation of the framework. Implementing the computations required would therefore require a number of transition rules that scales with the maximum input length considered, similarly to how the Sum + Modulo program requires a number of buckets that scales with the maximum

```
variables = {
     "parity": var(range=2, input_init_fn=lambda x: 0 if x == START else x),
     "done": var(range=2, input_init_fn=lambda x: x == START),
}
attention_heads = {
     "parity_left": v_relative("parity", -1),
     "done_left": v_relative("done", -1),
}

def ffn_fn(z):
  if z["done"] != 1 and z["done_left"] == 1:
     z["parity"] = z["parity_left"] ^ z["parity"]
     z["done"] = 1

return program_spec(
     variables=variables, heads=attention_heads,ffn_fn=ffn_fn,
     output_name="parity", input_range=3, position_range=None,
     halt=halt_spec("done", halt_value=1),
)
```

Figure 7: Program for computing parity sequentially using relative positions.

number of ones considered. Regardless, Chiang & Cholak (2022) showed this algorithm is difficult to learn in practice, and it also requires specialized positional encodings. Furthermore, their algorithm requires encoding parameters and activations with a degree of numerical precision that scales with the maximum input length. In contrast, the *Sequential (Relative)* program compiles to a Universal Transformer that only needs to encode 4 binary variables in the residual stream, and consists of only 7 transition rules, i.e., requires only 7 hidden MLP dimensions. However, the construction of Chiang & Cholak (2022) requires only 2 layers, where as the *Sequential (Relative)* program requires $N - 1$ layers, where N is the maximum input length.

Train and Test Sets The train and test sets are the same for all experiments (including both trace and end-to-end supervision). The train set consists of examples between lengths 0 and 20, and the test set contains examples between lengths 0 and 40. The sets include roughly an equal number of examples per number of ones.

Table 1: Trace supervision hyperparameters.

	Program		
Hyperparameter	**Sequential (Relative)**	**Sequential (Absolute)**	**Sum + Modulo**
Hidden Layers	2	4	4
Hidden Layer Size	128	4,096	4,096
Batch Size	256	256	256
Steps	50,000	50,000	400,000
Learning Rate	1e-2	1e-4	1e-4
Activation Fn	ReLU	ReLU	ReLU
Optimization Fn	Adafactor	Adam	Adam
Noise Std Dev	0.1	0.1	0.1

Trace Supervision Details Hyperparameters for training with trace supervision are listed in Table 1. All are standard hyperparameters for training neural networks except "Noise Std Dev." We added a small amount of Gaussian noise to the neural network input at training time to make it robust to numeric imprecision at inference time.

The *Sequential (Absolute)* and *Sum + Modulo* experiments used four hidden layers instead of the standard two, as we explored various hyperparameters to ensure the MLP had the capacity to fit the training set. Similarly, we used Adam (Diederik, 2014) for both experiments instead of Adafactor (Shazeer & Stern, 2018), as it helped fit the training set.

The *Sum + Modulo* program used for trace supervision differs slightly from the program in Figure 6. `num_ones` is stored as an intermediate categorical variable, as we found it easier for the MLP to learn to map a categorical variable to the correct parity output than a numeric variable.

End-to-end Training Details We trained transformers with various configurations using standard supervision — varying the number of layers, whether weight sharing is used, and the type of positional encoding. Constant in all standard supervision experiments are the following hyperparameters: embeddings with dimension 512, hidden layer sizes of 2048, 6 attention heads with dimension 64, an Adafactor optimization function, GeLU (Hendrycks & Gimpel, 2016) activation functions, a learning rate of 5e-4, 50,000 steps, and a byte vocabulary.

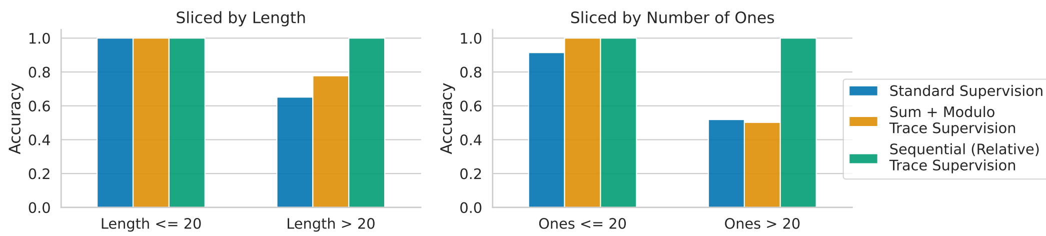

Figure 8: In distribution and out of distribution accuracy by length and number of ones for transformers trained using intermediate supervision and standard, end-to-end supervision. Transformers trained with standard supervision exhibit behavior similar to the *Sum + Modulo* intermediate supervision model. I.e., they exhibit no generalization to numbers of ones greater than those seen during training, but they do exhibit some length generalization.

Table 2: Length generalization accuracy for Transformers trained with intermediate supervision on parity.

	Accuracy		
Program	**Length** ≤ 20	**Length** > 20	**Ones** > 20
Sequential (Relative)	100%	100%	100%
Sequential (Absolute)	100%	52%	51%
Sum + Modulo	100%	78%	50%

Table 3: Length generalization accuracy for Transformers trained with standard supervision on parity.

			Accuracy		
Layers	**Weight Sharing**	**Positional Encoding**	**Length** ≤ 20	**Length** > 20	**Ones** > 20
8	No	Relative	100%	60%	51%
8	Yes	Relative	100%	65%	52%
40	No	Relative	100%	55%	51%
40	Yes	Relative	100%	54%	48%
8	No	Absolute	100%	49%	50%
8	Yes	Absolute	100%	50%	52%
40	No	Absolute	100%	50%	51%
40	Yes	Absolute	100%	52%	51%

Results Table 2 presents the accuracy for each intermediate supervision experiment and Table 3 presents the accuracy for each standard supervision configuration on different slices of the data. Figure 8 compares the intermediate and standard supervision results, showing that standard supervision exhibits behavior similar to the *Sum + Modulo* program.

Figures 9, 10, and 11 break down the standard supervision results in more detail.

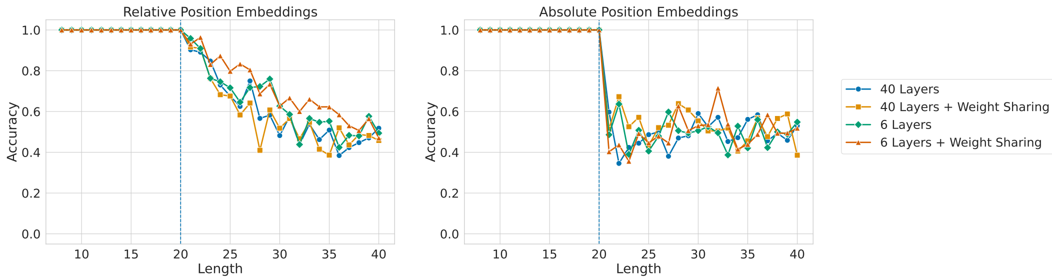

Figure 9: Accuracy by length for Transformers trained with standard supervision. There is some length generalization with relative position embeddings (left), but none with absolute (right).

Figure 9 shows that there is no length generalization when using absolute positional embeddings, while with relative positional embeddings there is some length generalization which decreases as length increases.

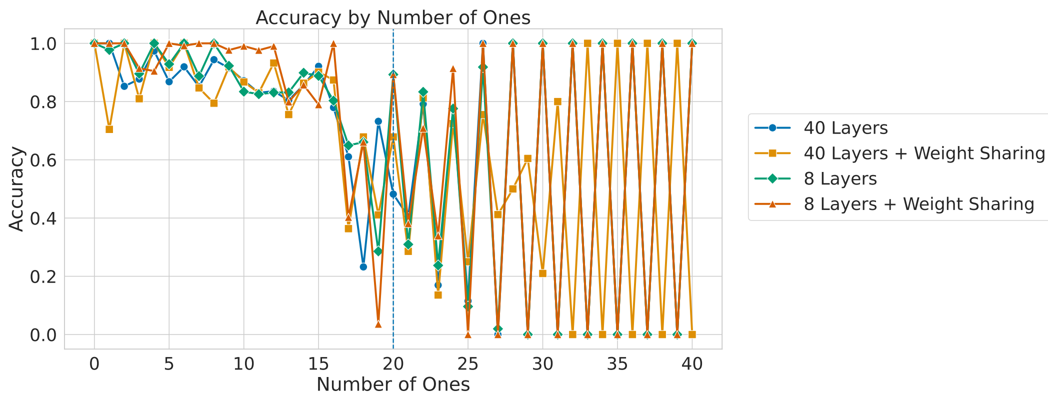

Figure 10: Accuracy by number of ones for Transformers trained with standard supervision using relative position embeddings. There is no generalization beyond 20 ones, which is the maximum number seen during training.

Figure 10 shows that the length generalization we observe with relative positional embeddings is due entirely to longer examples sometimes containing the same number of ones as shorter examples in the training distribution, consistent with results in Anil et al. (2022). After 20 ones, accuracy starts oscillating between 0 and 1. The oscillations are due to the model predicting either 0 or 1 for all examples once examples contain more ones than were seen during training, which is shown in Figure 11.

Simplicity Bias Prior work has attempted to understand the implicit inductive bias of Transformers in relation to a simplicity bias given some measure of complexity (Zhou et al., 2023b; Abbe et al., 2023; Bhattamishra et al., 2023; Tsoy & Konstantinov, 2024). For instance, inspired by the Minimum Description Length (MDL) principle (Rissanen, 1978; Grunwald, 2004), Zhou et al. (2023b) hypothesized that Transformers are biased towards learning behavior corresponding to the simplest RASP program that fits the training set, if one exists. Properties of the set of transition rules in an ALTA program can potentially provide new measures of Transformer complexity. For example, we can empirically compare the degree to which Transformers with layerwise weight sharing have an implicit simplicity bias towards learning "simpler" programs according to the number of transition rules. While the minimal *Sequential (Relative)* program contains 7 rules and the minimal *Sum + Modulo* program contains 31 rules, our results show that end-to-end model behavior is more consistent with the *Sum + Modulo* program. This indicates that Transformers, in this context, do not have an inherent simplicity bias that aligns with the number of transition rules expressed in ALTA. In this particular context, the end-to-end behavior is more consistent with the algorithm that requires the fewest layers to execute. Pfau et al. (2024) showed that Transformers struggle to effectively leverage intermediate

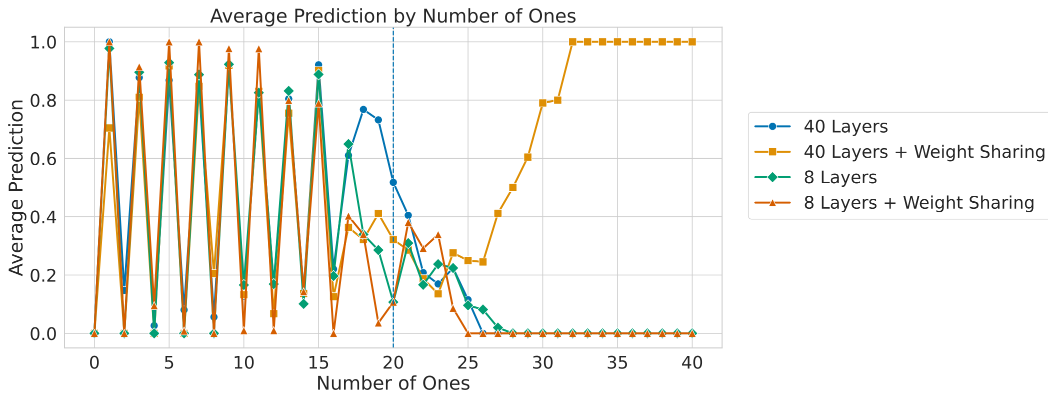

Figure 11: Average prediction by number of ones for Transformers trained with standard supervision using relative position embeddings. Once examples have slightly more than 20 ones, the maximum number seen during training, the models resort to predicting either 0 or 1 for all examples.

decoding steps to learn sequential algorithms without explicit supervision. Our results suggest a similar finding for leveraging weight-shared layers to learn sequential algorithms.

D.2 Addition

This task has been studied extensively (Nogueira et al., 2021; Liu et al., 2022), particularly with respect to length generalization (Zhou et al., 2024; Shen et al., 2023; Zhou et al., 2023b; Lee et al., 2024; Kazemnejad et al., 2024; Ruoss et al., 2023). While new positional encodings such as FIRE Li et al. (2024) can improve performance, Transformers still struggle with length generalization on this task, unless provided with carefully constructed supervision over intermediate decoding steps (Zhou et al., 2023b, 2024).

Our program for computing addition is given in Figure 13. The compiled MLP width (i.e., number of transition rules) is 884.

D.3 SUBLEQ

Our ALTA program for implementing a SUBLEQ interpreter is given in Figure 15. The input tokens define the set of memory registers, and program execution starts at position 0. SUBLEQ stands for SUBtract and branch if Less-than or EQual to zero. Commands in SUBLEQ are specified by three memory addresses A, B, and C. Executing a command consists of subtracting the value at memory address A from the value at address B, and writing the result to address B. If the result is less than or equal zero, then the program jumps to the command at address C, or will halt if $C < 0$. Giannou et al. (2023) showed how a Looped Transformer can implement an interpreter for a restricted form of SUBLEQ that did not allow self-modifying code, i.e., memory registers specifying SUBLEQ instructions could not be modified during program execution. Our program implements a SUBLEQ interpreter without such restrictions, i.e. without differentiating between program and memory registers. Additionally, our implementation executes 1 SUBLEQ instruction every 3 layers, as opposed to every 9 layers for the implementation of Giannou et al. (2023). However, our approach requires O(N^3) MLP hidden dimensions (i.e., transition rules), where N is the number of possible memory values. The construction of Giannou et al. (2023) requires only O(log(N)) MLP hidden dimensions.

D.4 SCAN

Background The SCAN suite of compositional generalization tasks (Lake & Baroni, 2018) require mapping natural language commands (e.g., "*jump twice and look left*") to action sequences (e.g., `JUMP JUMP LTURN LOOK`). The suite is inspired by the linguistic notion of systematic compositionality, i.e., the ability to recombine a finite set of elements in novel ways (Chomsky, 1957; Montague, 1970). Certain splits have been shown to be challenging for Transformer-based models (Keysers

```python
def init(x):
  # Initializes pointers.
  if x["token_right"] == END_TOKEN:
    x["ptr_b"] = 1
    x["ptr_out"] = 1
  if x["token_right"] == ADD_TOKEN:
    x["ptr_a"] = 1

def iterate(x):
  # Execute one step of addition.
  raw_sum = x["value_carry"] + x["ptr_a_token"] + x["ptr_b_token"]
  if x["ptr_out"]:
    x["value_out"] = raw_sum % 10
  x["value_carry"] = raw_sum // 10
  # Move all pointers to the left.
  # Attention heads attending to the right will be undefined.
  if x["token"] != END_TOKEN:
    x["ptr_out"] = x["ptr_out_right"]
    x["ptr_a"] = x["ptr_a_right"]
    x["ptr_b"] = x["ptr_b_right"]

def finalize(x):
  # Finalize output by adding the final carry to the output.
  if x["ptr_out"]:
    x["value_out"] = x["value_carry"]
  x["step"] = STEP_DONE

def ffn_fn(x):
  if x["step"] == STEP_INIT:
    init(x)
    x["step"] = STEP_ITERATE
  elif x["step"] == STEP_ITERATE:
    if x["ptr_a_token"] == START_TOKEN:
      x["step"] = STEP_FINALIZE
    else:
      iterate(x)
  elif x["step"] == STEP_FINALIZE:
    finalize(x)
```

Figure 12: MLP function for addition program.

et al., 2020; Furrer et al., 2020; Qiu et al., 2022b; Kazemnejad et al., 2024), especially the length split and the Maximum Compound Divergence (MCD) splits proposed by Furrer et al. (2020). Notably, no Transformer-based model has been shown to reliably solve these tasks, without relying on some symbolic decomposition of the task (Zhou et al., 2023a) or training data augmented by a symbolic system (Qiu et al., 2022a). All successful solutions have also involved symbolic parsing of some form (Shaw et al., 2021; Chen et al., 2020; Herzig & Berant, 2021). While prior work has studied the expressivity of various classes of Transformers with respect to formal languages, it has primarily focused on recognizing and generating Dyck languages (Yao et al., 2021; Bhattamishra et al., 2020; Hahn, 2020; Weiss et al., 2021; Ebrahimi et al., 2020), and thus has not previously shown a constructive demonstration of how a Transformer can solve the SCAN task.

Program Our approach follows Shaw et al. (2021) in formalizing the SCAN task as translation given a quasi-synchronous context-free grammar. Notably, the SCAN grammar is unambiguous and can be parsed in linear time. First, the program executes a shift-reduce parse of the input sequence, representing the parse as a tree. Second, the ALTA program decodes the output sequence by traversing the parse tree. The program represents the necessary variable-length data structures (a stack, parse tree, and buffer) using a variable number of input tokens. We include additional "memory" tokens in the input to ensure there are a sufficient number of tokens to represent these structures. We give an example of shift-reduce parsing for SCAN in Table 4.

In Figure 17 we show a program for parsing SCAN inputs using a shift-reduce parser. This program is a fragment of the overall SCAN program, which also decodes the output from the parsed representation, but is too verbose to include in this paper. We refer the reader to our open-source code for the full SCAN program.

```
variables = {
  "token": pb.var(INPUT_RANGE, input_init_fn=lambda x: x),
  # This variable tracks the current processing step.
  "step": pb.var(NUM_STEPS),
  # These are pointers to which digit is currently being processed.
  # They are `1` at the position of the current digit to process, and `0`
  # otherwise.
  "ptr_a": pb.var(2),
  "ptr_b": pb.var(2),
  # This pointer is `1` at the position to write the next output to,
  # and `0` otherwise.
  "ptr_out": pb.var(2),
  # This tracks the "carry" value form the previous iteration.
  "value_carry": pb.var(10),
  # This tracks the final output value for a given digit.
  "value_out": pb.var(10),
  # Static variables used as attention query.
  "one": pb.var(var_range=2, default=1),
}
attention_heads = {
  # For these relative attention heads, we always want to attend to the
  # position immediately to the right.
  "token_right": v_relative("token", 1),
  "ptr_a_right": v_relative("ptr_a", 1),
  "ptr_b_right": v_relative("ptr_b", 1),
  "ptr_out_right": v_relative("ptr_out", 1),
  # For these attention heads, we want to attend to the positions associated
  # with the current pointers.
  "ptr_a_token": qkv("one", "ptr_a", "token"),
  "ptr_b_token": qkv("one", "ptr_b", "token"),
}
return program_spec(
  variables=variables, heads=attention_heads, ffn_fn=ffn_fn,
  output_name="value_out", input_range=INPUT_RANGE, position_range=None,
  halt_spec=pb.halt_spec("step", halt_value=STEP_DONE),
)
```

Figure 13: Program for adding two positive integers. The MLP function is defined in Figure 12.

Table 4: Example of shift-reduce parsing for SCAN, for the input "jump twice". Our ALTA program represents the state of the stack and parse tree using a variable number of input tokens.

Action	Stack	Parse Tree	Input Buffer
Initialize	$\langle\,\rangle$	$\langle\,\rangle$	⟨ jump, twice ⟩
Shift	⟨ jump ⟩	$\langle\,\rangle$	⟨ twice ⟩
Reduce	⟨ NT ⟩	⟨ NT → jump ⟩	⟨ twice ⟩
Shift	⟨ NT, twice ⟩	⟨ NT → jump ⟩	$\langle\,\rangle$
Reduce	⟨ NT ⟩	⟨ NT → jump, NT → NT twice ⟩	$\langle\,\rangle$

End-to-end Training Details We trained *encoder-decoder* Transformers end-to-end on the SCAN length and MCD splits. We varied the number of layers, whether weight sharing was used, and the type of positional encodings used in the encoder. When using weight sharing, we trained with up to 256 encoder layers and 256 decoder layers (as our ALTA program requires at most 512 total layers). Without weight sharing, we only trained with up to 64 encoder layers and 64 decoder layers due to memory constraints.

Constant in all experiments were the following hyperparameters: embeddings with dimension 128, hidden layer sizes of 512, 8 attention heads with dimension 128, an Adafactor optimization function, GeLU activation functions, a learning rate of 5e-4, 100,000 steps, and a SentencePiece vocabulary (Kudo & Richardson, 2018).

Results Figure 18 plots test accuracy when using weight sharing and relative positional encodings. Accuracy does not improve as the number of layers increases. (If anything, there is a slight inverse

```
def encode(value):
  # Encodes a register value as positive integer.
  return value - MIN_VALUE

def decode(value):
  # Decodes a register value from positive integer.
  return value + MIN_VALUE

def _update_position(z, position_a):
  z["position_a"] = position_a
  z["position_b"] = position_a + 1
  z["position_c"] = position_a + 2

def ffn_fn(z):
  if z["state"] == STATE_1:
    if decode(z["a"]) < 0 or decode(z["b"]) < 0:
      # `a` or `b` are not valid register positions.
      z["state"] = STATE_DONE
      return
    z["state"] = STATE_2
  elif z["state"] == STATE_2:
    # Compute mem[b] - mem[a].
    mem_b = decode(z["mem_b"]) - decode(z["mem_a"])
    z["jump"] = int(mem_b <= 0)

    # Update memory value at position `b`.
    if z["position"] == z["b"]:
      z["mem"] = encode(mem_b)

    z["state"] = STATE_3
  elif z["state"] == STATE_3:
    # Determine next instruction.
    if z["jump"]:
      # Jump to instruction `c`.
      if decode(z["c"]) < 0:
        # Break if `c` is negative.
        z["state"] = STATE_DONE
      else:
        _update_position(z, z["c"])
        z["state"] = STATE_1
    else:
      # Proceed to next instruction.
      _update_position(z, z["position_a"] + 3)
      z["state"] = STATE_1
```

Figure 14: MLP function for interpreting SUBLEQ instructions.

relationship between number of layers and test accuracy.) This is the case in all experiments, regardless of the type of positional encoding used in the encoder and whether weight sharing is used. (See Table 5 for all results.)

The fact that increasing the number of layers does not improve generalization indicates that when trained with standard supervision, Transformers are unable to take advantage of extra layers to learn the sequential ALTA program that generalizes perfectly. All end-to-end experiments fit the training set, even with just two layers, so there exists some other algorithm that fits the training set that requires at most two layers. We speculate that in all cases, the end-to-end supervised Transformers are unable to learn the sequential algorithm with perfect generalization because of a bias towards learning algorithms that require fewer layers to execute. These results are consistent with our results on Parity, in which Transformers trained with end-to-end supervision do not learn the sequential algorithm with perfect generalization, instead seeming to mimic an algorithm that requires just one layer (see §D.1).

```
mem_range = (MAX_VALUE - MIN_VALUE) + 1
vars = {
    # Value of register.
    "mem": var(mem_range, input_init_fn=lambda x: x),
    # Position of register.
    "pos": var(mem_range, position_init_fn=encode),
    # Position of current instruction.
    "pos_a": var(mem_range, default=encode(0)),
    "pos_b": var(mem_range, default=encode(1)),
    "pos_c": var(mem_range, default=encode(2)),
    # Program state.
    "state": var(NUM_STATES),
    # Whether to jump at next instruction.
    "jump": var(2),
}

attention_heads = {
    # Values of registers at `pos_a`, `pos_b`, and `pos_c`.
    "a": qkv("pos_a", "pos", "mem"),
    "b": qkv("pos_b", "pos", "mem"),
    "c": qkv("pos_c", "pos", "mem"),
    # Value of registers at `a` and `b`.
    "mem_a": qkv("a", "pos", "mem"),
    "mem_b": qkv("b", "pos", "mem"),
}

variables = get_variables(mem_range)
attention_heads = get_attention_heads()

return program_spec(
    variables=variables, heads=attention_heads, ffn_fn=ffn_fn,
    output_name="mem", input_range=mem_range, pos_range=NUM_POSITIONS,
    halt=halt_spec("state", halt_value=STATE_DONE),
)
```

Figure 15: Program for interpreting SUBLEQ instructions. The MLP function is specified in Figure 14.

Table 5: Test accuracy for all Transformers trained with standard, end-to-end supervision on SCAN. Regardless of the configuration and split, increasing the number of layers does not improve test accuracy.

			Number of Layers						
Split	**Weight Sharing**	**Encoder Positional Encoding**	**2**	**6**	**16**	**32**	**64**	**128**	**256**
Length	Yes	Relative	11.1%	9.9%	8.9%	7.1%	7.0%	7.4%	5.0%
Length	Yes	Absolute	8.4%	8.6%	6.5%	4.8%	5.5%	5.7%	3.8%
Length	No	Relative	9.0%	9.2%	10.6%	10.2%	10.2%	–	–
Length	No	Absolute	9.1%	6.1%	6.5%	5.6%	4.4%	–	–
MCD1	Yes	Relative	5.5%	3.3%	2.2%	4.3%	2.9%	3.0%	2.3%
MCD1	Yes	Absolute	4.1%	1.1%	3.2%	1.5%	1.8%	1.6%	1.3%
MCD1	No	Relative	3.5%	3.7%	3.4%	1.7%	2.3%	–	–
MCD1	No	Absolute	4.7%	1.5%	1.1%	0.7%	1.9%	–	–
MCD2	Yes	Relative	8.6%	12.9%	8.9%	5.1%	3.6%	6.9%	3.3%
MCD2	Yes	Absolute	4.9%	5.6%	2.2%	2.1%	2.5%	1.9%	2.4%
MCD2	No	Relative	13.1%	7.4%	4.3%	5.2%	3.7%	–	–
MCD2	No	Absolute	4.2%	4.0%	1.7%	1.7%	1.1%	–	–
MCD3	Yes	Relative	12.3%	12.2%	6.3%	5.5%	7.0%	2.0%	2.7%
MCD3	Yes	Absolute	3.3%	4.3%	2.8%	3.2%	3.0%	1.8%	1.9%
MCD3	No	Relative	8.1%	7.6%	6.9%	3.2%	1.6%	–	–
MCD3	No	Absolute	3.9%	5.1%	2.0%	3.3%	1.2%	–	–

```
def shift_stack_pointers(z, stack_pointer_offset):
  new_stack_pointer_0 = z["stack_pointer_0"] + stack_pointer_offset
  z["stack_pointer_0"] = new_stack_pointer_0
  z["stack_pointer_1"] = new_stack_pointer_0 - 1
  z["stack_pointer_2"] = new_stack_pointer_0 - 2
  z["stack_pointer_3"] = new_stack_pointer_0 - 3

def reduce(z, matched_rule):
  # Pop RHS elements and add LHS nonterminal to stack.
  if z["position"] == (z["stack_pointer_0"] - rule_len(matched_rule)):
    z["symbol_id"] = rule_lhs_id(matched_rule)
  shift_stack_pointers(z, 1 - rule_len(matched_rule))
  # Add rule to parse tree.
  if z["position"] == z["tree_pointer"]:
    # Use 1-indexing to reserve 0 for no rule.
    z["rule_id"] = rule_id(matched_rule)
  z["tree_pointer"] += 1

def shift(z):
  # Shift the next token to the stack.
  if z["position"] == z["stack_pointer_0"]:
    z["symbol_id"] = get_symbol_id(z["input_pointer_token_id"])
  shift_stack_pointers(z, 1)
  z["input_pointer"] += 1

def ffn_fn(z):
  if not z["done"]:
    # Check if top-3 stack symbols (and 1 lookahead token) match any rule.
    matched_rule = maybe_match_rule(
        z["input_pointer_token_id"],
        z["stack_symbol_1"],
        z["stack_symbol_2"],
        z["stack_symbol_3"],
    )
    if matched_rule is not None:
      reduce(z, matched_rule)
    else:
      # Check if parsing is complete.
      if z["input_pointer_token_id"] == EOS_ID:
        z["done"] = 1
      else:
        shift(z)
```

Figure 16: MLP function for parsing SCAN.

E Extended Discussion

Limitations ALTA has many of the same limitations as RASP and Tracr with respect to the potential differences between compiled models and those learned in practice, as discussed in Lindner et al. (2023).

Opportunities There are many potential opportunities for applying ALTA beyond the applications studied in this paper. For example, we discussed potential extensions to how trace supervision could be employed in §B.2. Additionally, ALTA could potentially help develop test cases for interpretability tools. This was one of the primary motivations for Tracr, which has been applied to help design interpretability benchmarks (Thurnherr & Scheurer, 2024) and more interpretable Transformers (Friedman et al., 2023). Finally, compiled components can potentially be integrated within learned models, such as circuits for arithmetic (Nanda et al., 2023) or induction heads (Akyürek et al., 2024).

```
variables = {
    "token": var(NUM_INPUT_TOKENS, input_init_fn=lambda x: x),
    "position": var(NUM_POSITIONS, position_init_fn=lambda x: x),
    # Whether parsing is complete.
    "done": var(2),
    # Pointer to the next stack position, and then the top 3 elements on
    # the stack.
    "stack_pointer_0": var(NUM_POSITIONS, default=STACK_OFFSET),
    "stack_pointer_1": var(NUM_POSITIONS, default=STACK_OFFSET - 1),
    "stack_pointer_2": var(NUM_POSITIONS, default=STACK_OFFSET - 2),
    "stack_pointer_3": var(NUM_POSITIONS, default=STACK_OFFSET - 3),
    # Pointer to write the next rule to.
    "tree_pointer": var(NUM_POSITIONS, default=TREE_OFFSET),
    # Pointer to the next input token to process.
    "input_pointer": var(NUM_POSITIONS, default=INPUT_OFFSET),
    # Stores index of associated parsing rule.
    "rule_id": var(NUM_RULES),
    # Stores symbol ID associated with stack element.
    "symbol_id": var(NUM_SYMBOLS),
}

heads = {
    # Get token at input pointer.
    "input_pointer_token_id": qkv("input_pointer", "position", "token"),
    # Get top 3 symbols on stack.
    "stack_symbol_1": qkv("stack_pointer_1", "position", "symbol_id"),
    "stack_symbol_2": qkv("stack_pointer_2", "position", "symbol_id"),
    "stack_symbol_3": qkv("stack_pointer_3", "position", "symbol_id"),
}

return program_spec(
    variables=variables, heads=heads, ffn_fn=ffn_fn,
    output_name="rule_id",
    input_range=NUM_INPUT_TOKENS,
    position_range=NUM_POSITIONS,
)
```

Figure 17: Program for parsing SCAN. The MLP function is defined in Figure 16.

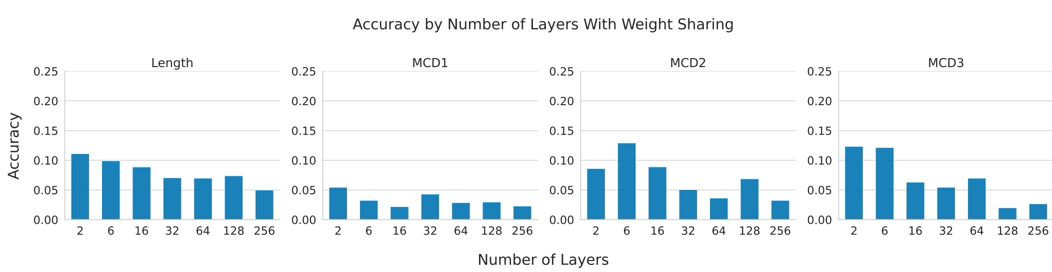

Figure 18: Test accuracy by number of layers on SCAN splits when trained using standard, end-to-end supervision with weight sharing and relative positional encodings. Test accuracy does not increase as the number of layers increases.