
Transformers are Efficient Compilers, Provably

Anonymous Author(s)

Affiliation

Address

email

Abstract

1 Transformer-based large language models (LLMs) have demonstrated surprisingly
2 robust performance across a wide range of language-related tasks, including pro-
3 gramming language understanding and generation. In this paper, we take the first
4 steps towards a formal investigation of using transformers as compilers from an
5 expressive power perspective. To this end, we introduce a representative pro-
6 gramming language, **Mini-Husky**, which encapsulates key features of modern
7 C-like languages. We show that if the input code sequence has a bounded depth
8 in both the Abstract Syntax Tree (AST) and type inference (reasonable assump-
9 tions based on the clean code principle), then the number of parameters required
10 by transformers depends only on the *logarithm of the input sequence length* to
11 handle compilation tasks, such as AST construction, symbol resolution, and type
12 analysis. A significant technical challenge stems from the fact that transform-
13 ers operate at a low level, where each layer processes the input sequence as raw
14 vectors without explicitly associating them with predefined structure or meaning.
15 In contrast, high-level compiler tasks necessitate managing intricate relationships
16 and structured program information. Our primary technical contribution is the
17 development of a domain-specific language, **Cybertron**, which generates formal
18 proofs of the transformer’s expressive power, scaling to address compiler tasks.
19 We further establish that recurrent neural networks (RNNs) require at least a lin-
20 ear number of parameters relative to the input sequence, leading to an exponential
21 separation between transformers and RNNs. Finally, we empirically validate our
22 theoretical results by comparing transformers and RNNs on compiler tasks within
23 **Mini-Husky**.

24 1 Introduction

25 Transformers (Vaswani, 2017) have demonstrated remarkable proficiency across various do-
26 mains, achieving near-expert performance in solving International Mathematical Olympiad prob-
27 lems (Google Deepmind, 2024) and excelling in complex reasoning tasks in science, coding, and
28 mathematics (OpenAI, 2024a). They also handle routine coding tasks with high precision, such
29 as translating Agda code into TypeScript, outperforming the outdated and expensive-to-maintain
30 AgdaJS compiler (Taelin, 2023b), and integrating into code editors to significantly boost program-
31 mers’ productivity (cur, 2024; Taelin, 2023a). Despite these advancements, the full extent of their
32 underlying capabilities remains only partially understood.

33 In this paper, we aim to deepen our understanding of transformers’ abilities to perform compilation
34 tasks. Empirically, transformer-based LLMs have shown rapid progress in code generation and com-
35 pilation. For example, MetaLL (Cummins et al., 2024) enables LLMs to optimize code by interpret-
36 ing compiler intermediate representations (IRs), assembly language, and optimization techniques.
37 Gu (2023) highlights the ability of LLMs to generate high-quality test cases for Golang compil-
38 ers. Surprisingly, Taelin (2023b) demonstrates that models like Sonnet-3.5 can compile legacy code

39 into modern languages like TypeScript, outperforming the now obsolete AgdaJS compiler (Agda
40 Development Team, 2024).

41 To formally study this problem in a controlled setup, we designed a C-like programming language
42 called **mini-husky**, which encapsulates key features of modern C-like languages such as (Flanagan,
43 2011) and Rust (Klabnik & Nichols, 2023). We focus on three representative compilation tasks: ab-
44 stract syntax tree (AST) construction, symbol resolution, and type analysis. The AST is a recursive
45 structure that represents the input as a tree. From the perspective of programming language design,
46 the AST is considered the true representation of the input, with the textual code serving merely as
47 a convenient interface for human users (Alfred et al., 2007). All syntactic and semantic processing
48 can then be interpreted as specific operations on these trees. Symbol resolution involves verifying
49 the validity of references to entities and flagging errors for undefined symbols. Type analysis en-
50 compasses both type inference, which assigns types to variables without explicit annotations, and
51 type checking, which identifies mismatches between actual and expected types.

52 We demonstrate that, under the clean code principle (Martin, 2008), transformers with a number of
53 parameters that scale logarithmically with the input length can efficiently perform AST construction,
54 symbol resolution, and type analysis. To the best of our knowledge, this is the first theoretical
55 demonstration that transformers can function as compilers in a parameter-efficient manner.

56 We further compare transformers and recurrent neural networks (RNNs). By connecting the type
57 analysis task with the associative recall, we show even under the *clean code principle* (Martin,
58 2008), RNNs require a memory size that scales *linearly* with the input sequence length to success-
59 fully perform type analysis. Consequently, for type analysis in compilation, transformers can be
60 *exponentially more efficient* than RNNs. We also empirically validate our theoretical findings by
61 demonstrating the superiority of transformers in the type analysis task.

62 **Technical Challenges and Our Technique.**

63 Proving that transformers can perform compilation tasks presents several challenges:

- 64 • **Transformers operate at too low a level.** Transformers process sequences of floating-point vec-
65 tors, akin to raw bits in computers, and proving their ability to perform specific tasks is similar
66 to writing specialized parallel machine code. Previous work (Yao et al., 2021) often resorts to
67 graphical illustrations for readability, even for basic tasks.
- 68 • **Compilers are exceedingly high-level.** Compilers are one of the most challenging programming
69 projects in our era and early C compilers. Compilers are among the most complex programming
70 endeavors of our time. Compilation involves numerous sophisticated procedures, some of which
71 are undecidable or computationally expensive, such as code optimization (Alfred et al., 2007))
72 and type analysis (Pierce, 2002). For example, type analysis in complex type systems poses
73 significant challenges, often requiring the development of advanced logical frameworks (Dunfield
74 & Krishnaswami, 2019).

75 To overcome these challenges, we design a domain-specific language (DSL) called **Cybertron** to
76 serve as the proof vehicle, i.e., a major part of our proof consists of reasoning about type-correct
77 code in Cybertron that represents a transformer. Without using **Cybertron**, writing an equivalent
78 natural language proof would be too complex and intractable. Using code to prove propositions is
79 not new to computer science; it is, in fact, the norm in interactive theorem proving (ITP) (Har-
80 rison et al., 2014). ITP focuses on generating computer-verifiable proofs through a combination
81 of human-guided instructions and software automation. For instance, the correctness of the Ke-
82 pler conjecture (Hales et al., 2017) is verified by the combination of the ITP theorem provers HOL
83 Light (Harrison, 2009) and Isabelle (Paulson, 1994). To our knowledge, we are the first to apply this
84 approach to understanding neural networks.

85 **Contributions.** We summarize our contributions below:

- 86 • **A testbed for compilation tasks:** We introduce **Mini-Husky**, a simple yet representative C-like
87 programming language, designed to formally assess transformers’ capabilities in programming
88 language processing. We anticipate that **Mini-Husky** will become a standard testbed for this
89 purpose.
- 90 • **Expressive power theory of transformers as compiler:** We provide a formal proof that, when
91 the input code sequence has bounded AST depth and inference depth, the number of parameters
92 in transformers only needs to scale logarithmically with the input sequence length to handle com-

93 pilation tasks such as AST construction, symbol resolution, and type analysis. To the best of our
94 knowledge, this is the first study exploring the power of transformers as compilers.

- 95 • **Transformers vs. RNNs:** Theoretically, we demonstrate a negative result, showing that the num-
96 ber of parameters in RNNs must scale linearly with the input sequence length to perform type
97 analysis correctly. This result establishes an exponential separation between transformers and
98 RNNs. We further empirically confirm the advantage of transformers for the type analysis task.
- 99 • **A Domain-Specific Language for Proofs:** Given the challenges in formal proofs, we design a
100 domain-specific language, **Cybertron**, to serve as a proof vehicle. We believe that **Cybertron**, and
101 the general approach of using DSLs for analysis, can have broader applications in understanding
102 transformers and other architectures.

103 2 Related Work

104 **Expressive Power of Transformers.** A line of work studies the expressive power of attention-based
105 models. One direction focuses on the universal approximation power (Yun et al., 2019; Bhattamishra
106 et al., 2020b,c; Deghani et al., 2018; Pérez et al., 2021). More recent works present fine-grained
107 characterizations of the expressive power for certain functions in different settings, sometimes with
108 statistical analyses (Edelman et al., 2022; Elhage et al., 2021; Likhoshesterov et al., 2021; Akyürek
109 et al., 2022; Zhao et al., 2023; Yao et al., 2021; Anil et al., 2022; Barak et al., 2022; Garg et al.,
110 2022; Von Oswald et al., 2022; Bai et al., 2023; Olsson et al., 2022; Akyürek et al., 2022; Li et al.,
111 2023; Hao et al., 2022; Pérez et al., 2019; Strobl, 2023; Chiang et al., 2023; Wei et al., 2022; Wang
112 et al., 2022; Feng et al., 2023; Li et al., 2024; Reddit User, 2013). The most related one is Yao et al.
113 (2021) where the authors prove constructively that bounded depth Dyck language can be recognized
114 by encoder-only hard attention transformers, which has similarities to our settings of bounded depth
115 programming language recognized encoder-only hard attention transformers. The major difference
116 is that we introduce concepts and tasks from programming language theory Pierce (2002) to study
117 the semantic powers of transformers.

118 **Transformers vs. RNN.** It is important to understand the comparative advantages and disadvantages
119 of transformers against RNNs. Empirically, synthetic experiments have shown an advantage of
120 transformers against RNNs for long range tasks (Bhattamishra et al., 2023; Arora et al., 2023).
121 Theoretically, there has been a rich line of work focusing on comparing transformers and RNNs in
122 terms of recognizing formal languages (Bhattamishra et al., 2020a; Hahn, 2019; Merrill et al., 2021),
123 which show that the lack of recursive structure of transformers prevent them from recognizing some
124 formal languages that RNNs can recognize. However, the gap can be mitigated when we consider
125 the bounded length of input or bounded grammar depth (Liu et al., 2022; Yao et al., 2021), which
126 is quite reasonable in practice and is used in this paper. On the other side, prior work (Jelassi et al.,
127 2024; Wen et al., 2024) proves a representation gap between RNNs and Transformers in repeating
128 a long sequence. In summary, it is somehow intuitive that recursive structures with limited memory
129 perform badly at tasks which requires information retrieval. Our paper shows that semantic analysis
130 for programming languages is such a task.

131 **DSLs for Transformers.** We note that we are not exactly the first one to employ a domain-specific
132 language to understand the expressive powers of transformers. Previously, DSLs with simple typings
133 like RASP (Weiss et al., 2021) were proposed to prove constructively that transformers can do
134 various basic sequence-to-sequence operations. Lindner et al. (2023) writes a compiler that compiles
135 RASP into actual transformers, Friedman et al. (2023) shows that RASP can be learned, and Zhou
136 et al. (2023) uses RASP to prove that simple transformers can perform certain algorithms. The major
137 difference between RASP and our DSL **Cybertron** is that **Cybertron** has a powerful algebraic type
138 system that helps to prove complicated operations beyond simple algorithms.

139 3 Preliminaries

140 The major innovation in the transformer architecture is self-attention, which processes input tokens
141 in a distributed manner. This capability enables the model to handle long-range dependencies, a
142 crucial feature for language tasks. We use hard attention and simplified position encoding to simplify
143 our theoretical reasoning.

144 **Attention.** In practice, attention heads use **soft attention**. Given model dimension d_{model} , num-
 145 ber of heads H , and a finite set of token positions Pos, an attention layer with simplified position
 146 encoding is defined as a function $f_{\text{attn}} : \mathbb{R}^{\text{Pos} \times d_{\text{model}}} \rightarrow \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$ given by

$$\forall p \in \text{Pos}, \quad f_{\text{attn}}(X)_p := W_O \text{Concat} \left(\text{Attn}^{(1)}(X)_p, \dots, \text{Attn}^{(H)}(X)_p \right), \quad (1)$$

147 where the h th attention head is defined using soft attention as: $\text{Attn}^{(h)}(X)_p := \sum_{p' \in \text{Pos}} \alpha_{p,p'}^{(h)} V_{p'}^{(h)}$.

148 The attention weights $\alpha_{p,p'}^{(h)}$ given by: $\alpha_{p,p'}^{(h)} = \frac{\exp(Q_p^{(h)\top} K_{p'}^{(h)} + \lambda^{(h)\top} \Psi_{p'-p})}{\sum_{p'' \in \text{Pos}} \exp(Q_p^{(h)\top} K_{p''}^{(h)} + \lambda^{(h)\top} \Psi_{p''-p})}$, where $W_O \in$

149 $\mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ are trainable parameters, $Q_p^{(h)}, K_p^{(h)}, V_p^{(h)} \in \mathbb{R}^{d_{\text{model}}/H}$ are linear transformations of
 150 $X_p, \lambda^{(h)} \in \mathbb{R}^2$ depends on the head, and $\Psi_q = \begin{pmatrix} q \\ 1_{q>0} \end{pmatrix} \in \mathbb{R}^2$ accounts for relative position.

151 For theoretical convenience, we use hard attention, commonly used in theoretical analysis of trans-
 152 former (Yao et al., 2021; Hahn, 2019). Hard attention can be viewed as the limit of soft attention
 153 when the attention logits become infinitely large. The hard attention head is defined as:

$$\text{Attn}^{(h)}(X)_p := \frac{1}{|S_p|} \sum_{p' \in S_p} V_{p'}^{(h)}, \quad \text{where } S_p = \arg \max_{p' \in \text{Pos}} \left(Q_p^{(h)\top} K_{p'}^{(h)} + \lambda^{(h)\top} \Psi_{p'-p} \right) \quad (2)$$

154 In other words, hard attention selects the positions p' that maximize the attention score for each
 155 position p , and averages the corresponding value vectors $V_{p'}^{(h)}$.

156 **Feed-Forward Layer.** Given model dimension d_{model} , and a finite set of token positions Pos, a
 157 feed-forward layer is a fully connected layer applied independently to each position, defined as a
 158 function $f_{\text{ffn}} : \mathbb{R}^{\text{Pos} \times d_{\text{model}}} \rightarrow \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$ given by

$$\forall p \in \text{Pos}, \quad f_{\text{ffn}}(X)_p = W_2 \sigma_{\text{ReLU}}(W_1 X_p + b_1) + b_2, \quad (3)$$

159 where $W_1 \in \mathbb{R}^{d_{\text{ffn}} \times d_{\text{model}}}$ and $W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ffn}}}$ are trainable weight matrices, $b_1 \in \mathbb{R}^{d_{\text{ffn}}}$ and $b_2 \in$
 160 $\mathbb{R}^{d_{\text{model}}}$ are trainable bias vectors, d_{ffn} is the hidden dimension of the feed-forward layer, chosen to be
 161 $2d_{\text{model}}$, as commonly used in practice, σ_{ReLU} is the ReLU activation function.

162 **Encoder-Only Transformer.** Encoder-only transformers consist solely of the encoder stack, mak-
 163 ing them ideal for tasks like classification, regression, and sequence labeling that do not require
 164 sequence generation. Each encoder layer includes a multi-head self-attention mechanism and a
 165 feed-forward network, allowing the model to capture complex dependencies and contextual infor-
 166 mation.

167 One can define it using the following recursion,

- 168 • The input is given by: $X^{(0)} = X$.
- 169 • For each layer $l = 1, 2, \dots, L$:
 - 170 – Compute attention output: $\hat{X}^{(l)} = X^{(l-1)} + f_{\text{attn}}^{(l)}(X^{(l-1)})$,
 - 171 – Compute feed-forward output: $X^{(l)} = \hat{X}^{(l)} + f_{\text{ffn}}^{(l)}(\hat{X}^{(l)})$.

172 In the above, $f_{\text{attn}}^{(l)}$ are the attention layers, and $f_{\text{ffn}}^{(l)}$ are the feed-forward layers, with the same model
 173 dimension d_{model} , number of heads H , and set of token positions Pos. For simplicity, layer normal-
 174 ization is ignored. See Appendix C for full details of transformers and other architectures.

175 4 Programming Language Processing and The Target C-Like Language: 176 Mini-Husky

177 Recently, transformers have expanded to include code analysis and generation (Nijkamp et al., 2023;
 178 Chen et al., 2021; Anysphere, 2023). Programming languages offer a cleaner foundation for study-
 179 ing language understanding, as their syntactic and semantic tasks are precisely defined. To formally
 180 study the language processing capabilities of transformers, we design **Mini-Husky**, a representative

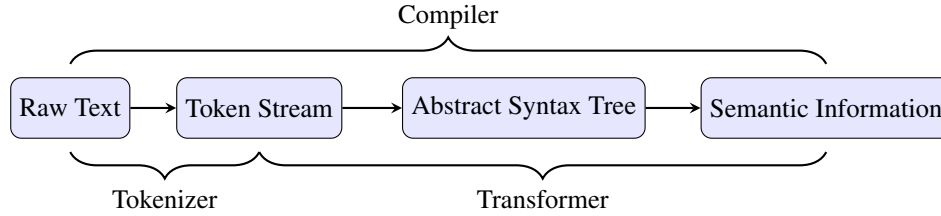


Figure 1: Programming language processing pipeline

181 mix of modern C-like languages with strong typing and typical syntactic features. It supports user-
 182 defined types (e.g., structs, enums) and enforces strict type equality, disallowing implicit conver-
 183 sions. Lexical scoping, including shadowing, ensures proper variable accessibility based on block
 184 structures, type inference, and checking. These features make *the Mini-Husky compiler* a rep-
 185 resentative task to evaluate transformers’ capabilities in syntactic and semantic tasks like symbol
 186 resolution and type checking. See Appendix D for the full details of **Mini-Husky**.

187 The standard pipeline of processing programming languages is shown in Figure 1 (Alfred et al.,
 188 2007). The raw text is firstly segmented into parts like literals, identifiers, punctuations, keywords,
 189 etc, called token stream, then parsed into a tree like structure representation of the generation process
 190 of the input, finally syntactic and semantic analysis is performed on the tree. In this paper, to simplify
 191 the presentation, we assume tokenizer has been provided priori. Below we describe key tasks of
 192 programming language processing.

193 **Abstract Syntax Tree Construction.** Abstract Syntax Tree (AST) is a hierarchical, tree-like repre-
 194 sentation of the syntactic structure of source code in a programming language. Unlike the raw text of
 195 the code, the AST abstracts away from surface syntax details, capturing the essential elements and
 196 their relationships in a structured form. Each node in the AST corresponds to a construct occurring
 197 in the source code, such as expressions, statements, or declarations. This representation is central
 198 to various stages of language processing, enabling efficient syntax checking, semantic analysis, and
 199 code generation. The formal definition of ASTs is standard in the programming language literature
 200 but is lengthy, so we defer to Appendix A.

201 The AST construction task’s final output is the collection all AST nodes. We will show transformers
 202 can construct AST efficiently.

203 **Symbol Resolution.** In programming languages, symbols are functions, types, generics, variables,
 204 macros, etc. They are defined in one place and can be used by referring to the corresponding identi-
 205 fier or path in a certain scope. The scope can be within a certain tree of modules, or within a certain
 206 curly bracketed scope within one module. For simplicity, we only consider curly bracketed scope.

207 In **Mini-Husky**, the following showcases symbol resolution.

```

208 1 pub fn f() {
209 2     fn f1() {}
210 3
211 4     let a = 1;
212 5     let x = a;
213 6     let a = 2;
214 7     {
215 8         let a = 3;
216 9         { let a = 4; }
217 10        let y = a;
218 11    }
219 12    let z = a;
220 13 }
221 14
222 15 fn g() { f() }
  
```

223 The outer function f is accessible everywhere in the body of function g . However, the inner function
 224 $f1$ can only be used inside the body of f as it is defined within the body. For variables with the same
 225 identifier a , the first is accessible from lines 4 and 5, the second is accessible from line 12, the third
 226 is from line 10, and the fourth is not accessible from anywhere. Thus $x = 1, y = 3, z = 2$.

227 The output of the **symbol resolution** task is the collection of symbol resolution results on all appli-
 228 cable tokens. More concretely, the output is a sequence of values of type `Option<SymbolResolution>`
 229 where `Option<SymbolResolution>` is the type `SymbolResolution` with a null value added for non-
 230 applicability and `SymbolResolution` is the type storing the result of the symbol resolution, be-
 231 ing either a success with a resolved symbol of type `Symbol` or a failure with an error of type
 232 `SymbolResolutionError`. We shall prove that transformers can do symbol resolution and that atten-
 233 tion is crucial.

234 **Type Analysis.** In general, type is essential for conveying the intended usage of the written functions
 235 and specifying constraints. As a first exploration of this topic, we try to make the type analysis in
 236 **Mini-Husky** as simple as possible yet able to bring out the essential difficulty. The type system con-
 237 sists of four sequential components: (1) *Type definition*, (2) *Type specification*, (3) *Type inference*,
 238 and (4) *Type checking*. Due to the page limit, here we only introduce (4) *Type checking* because it is
 239 the final step and this is a crucial step which separates transformers and RNNs. See Appendix D.1
 240 for details of (1) *Type definition*, (2) *Type specification*, and (3) *Type inference*

241 (4) *Type checking*. Type checking ensures that the type expressions agree with its expectations.
 242 For simplicity, we do not allow implicit type conversion, so the agreement means exact equality of
 243 types. The arguments of function calls are expected to have types according to the definition of the
 244 function. The operand type of field access must be a struct type with a field of the same name. The
 245 type of the last expression of the function body or the expr in the return statement must be equal to
 246 the return type of the function. For variables defined in the `let` statement, If the types are annotated,
 247 the types of the left-hand side and right-hand side should be in agreement.

```

248 1 // Type Error: the return type is 'i32', yet the last expression is of type 'f32'
249 2 fn f(a: i32) -> i32 { 1.1 }
250 3
251 4 struct A { x: i32 }
252 5
253 6 fn g() {
254 7   // Type Error: 'x' is of type f32 but it's assigned by a value of type 'i32'
255 8   // Type Error: the first argument of 'f' expects be of type 'i32' but gets a float
256   literal instead
257 9   let x: f32 = f(1.1);
258 10  // Type Error: no field named 'y'
259 11  let y = A { x: 1 }.y;
260 12 }
```

261 The above incorporates typical examples of type disagreements that count as type errors. A compiler
 262 should be able to report these errors.

263 The **type analysis** task's final output is the collection of all type errors. More concretely, the output
 264 is a sequence of `Option<TypeError>`, where `Option<TypeError>` denoted the type `TypeError` will a
 265 null value added and `TypeError` is the type storing the information of a type error. The position of
 266 type errors agrees with the source tokens leading to these errors.

267 5 Expressive Power of Transformers as Efficient Compilers

268 In this section we discuss main theoretical results about the expressive power of transformers to
 269 perform compilation tasks: AST construction, symbol resolution, and type analysis. In Section 5.4,
 270 we discuss **Cybertron**, a DSL specifically designed for our proof.

271 5.1 Abstract Syntax Tree Construction

272 We start with a definition that characterizes low-complexity codes.

273 **Definition 1** (Codes with Bounded AST-Depth). *Let MiniHusky_D be the set of token sequences*
 274 *that can be parsed into valid ASTs in **Mini-Husky** with a depth less than D .*

275 D in the above definition is small in practice, and a linear dependency on D is acceptable, but the lin-
 276 ear dependency on L is not. The fundamental reason is that *the clean code principle* (Martin, 2008)
 277 requires one to write code with as little nested layer as possible for greater readability. Readability is of

278 the utmost importance because “Programs are meant to be read by humans and only incidentally for
 279 computers to execute” (Abelson et al., 1996). This assumption of bounded hierarchical depth is not
 280 limited to just programming languages, but is often seen as applicable to natural languages (Frank
 281 et al., 2012; Brennan & Hale, 2019; Ding et al., 2017), motivating Yao et al. (2021) to have a similar
 282 boundedness assumption. Below is the main result for AST construction using transformers.

283 **Theorem 1.** *There exists a transformer encoder of model dimension and number of layers being*
 284 *$O(\log L + D)$ and number of heads being $O(1)$ that represents a function that maps any token*
 285 *sequence of length L in MiniHusky_D to its abstract syntax tree represented as a sequence.*

286 We note $\log L$ is small because a 64-bit computers can only process context length at most 2^{64} and
 287 D is small by assumption. Therefore, there exists a transformer with an almost constant number of
 288 parameters that is able to process comparatively much longer context length.

289 *Proof Sketch.* The idea is to construct ASTs in a bottom-up manner with full parallelism. We shall
 290 recursively produce the final ASTs in at most D steps. We shall maintain two values, called `pre_ast`
 291 and `ast`. `ast` represents ASTs that have already been allocated, although they might not been fully
 292 initialized. `pre_ast` represents tokens that have yet to form ASTs and new ASTs that have not been
 293 fully initialized. For each round, we try to create new ASTs from `pre_ast` and update `ast` and
 294 `pre_ast`. For the n -th round, we provably allocated all ASTs with depth no more than n . Then for
 295 the D -th round, all ASTs are properly constructed and allocated. Each round can be represented by
 296 a transformer of a number of heads $O(1)$, model dimension $O(\log L + D)$, and a number of layers
 297 $O(1)$. Therefore, The end-to-end process is then representable by a transformer of the number of
 298 heads $O(1)$, model dimension $O(\log L + D)$ and the number of layers $O(\log L + D)$. See full details
 299 in in Appendix F. \square

300 5.2 Symbol Resolution

301 Next, we show that transformers can effectively perform symbolic resolution as $\log L$ and D are
 302 almost constant as compared with context length L .

303 **Theorem 2.** *There exists a transformer encoder of model dimension and number of layers being*
 304 *$O(\log L + D)$ and number of heads being $O(1)$ that represents a function that maps any token*
 305 *sequence of length L in MiniHusky_D to its symbol resolution represented as a sequence of values*
 306 *of type `Option<SymbolResolution>`.*

307 *Proof Sketch.* First, we need to define the type for scopes. It is represented by a tiny sequence of
 308 indices of curly brace block AST that enclose the type/function/variable. We assign the scope by
 309 walking through the ASTs in a top-down manner. We not only assign scopes to item definitions,
 310 we also: (1) assign scopes to ASTs representing curly brace blocks, with these scopes equal to the
 311 scope of block itself, and (2) assign scopes to identifiers waiting to be resolved, with these scopes
 312 equal to the maximum possible scope of its resolved definition. The computation process is easily
 313 represented in Cybertron, indicating attention is expressive enough for this calculation and it only
 314 takes $O(D)$ number of layers.

315 After obtaining all the scopes for all items, it takes only one additional layer to obtain the symbolic
 316 resolution through attention. As attention is expressed through the dot product of two linear projec-
 317 tions Q and K , we have to choose the representation of the scope type properly to finish the proof.
 318 The full details are in Appendix G. \square

319 5.3 Type Analysis

320 We need an additional definition to characterize the complexity of codes for type analysis.

321 **Definition 2** (Codes with Bounded AST-Depth and Type-Inference-Depth). *We use*
 322 *$\text{MiniHuskyAnnotated}_{D,H}$ to denote the subset of MiniHusky_D with the depth of type in-*
 323 *ference no more than H . The depth of type inference is the number of rounds of computation needed*
 324 *to infer all the types using the type-inference algorithm (described in Appendix D.1).*

325 In practice, H is significantly smaller than with context length L for reasonably written code be-
 326 cause it is upper bounded by the number of statements in a function body which is required to be

327 small according to clean code principle (Martin, 2008). Below, we present the main result of using
328 transformers for type analysis. See full details in Appendix H.

329 **Theorem 3.** For $L, D, H \in \mathbb{N}$, there exists a transformer encoder of model dimension, and number
330 of layers being $O(\log L + D + H)$ and number of heads being $O(1)$ that represents a function that
331 maps any token sequence of length L in `MiniHuskyAnnotatedD,H` to its type errors represented as
332 a sequence of values of type `Option<TypeError>`.

333 5.4 Proof Vehicle: Cybertron, a Domain-Specific Language

334 Here we highlight our main proof technique. Proving that transformers can express complex algo-
335 rithms and software like compilers is a significant challenge due to the inherent differences between
336 how transformers operate and the nature of high-level tasks they are expected to perform. Trans-
337 formers process input at a low level, where each layer manipulates raw token sequences as vectors
338 without predefined structure or meaning. However, high-level tasks—such as constructing ASTs
339 and performing type and symbol analysis—require handling complex, structured information that
340 depends on long-range relationships and interactions across the input. Bridging the gap between
341 this raw, unstructured processing and the structured, multi-step logic required for these tasks in-
342 troduces significant difficulty. Compilers, for instance, typically rely on rule-based, step-by-step
343 operations that are abstract and sequential, which transformers must simulate through their attention
344 mechanisms and feedforward layers. The challenge is further compounded by the need to formally
345 prove that transformers can handle such tasks efficiently and accurately, despite operating in a fun-
346 damentally different manner. To address these challenges, we propose a domain-specific language
347 (DSL) called **Cybertron**, which allows us to *systematically prove* that transformers are capable of
348 expressing complex algorithms while maintaining sufficient readability.

349 A key feature of **Cybertron** is its expressive type system, which provides strong correctness guar-
350 antees. The type system ensures that every value is strongly typed, making it easier to reason about
351 function composition and ensuring the validity of our proofs. This type system is crucial for man-
352 aging how transformers represent and manipulate both local and global types—where local types
353 correspond to individual tokens and global types refer to sequences of tokens, encapsulating broader
354 program information.

355 What transformers output (possibly in the intermediate layers) is a representation in sequences of
356 vector of sequences of values in these types. As types are mathematically interpreted in this paper a
357 discrete subset of a vector space, **Cybertron** allows us to construct transformers with an automatic
358 value validity guarantees if the **Cybertron** code is type-correct.

359 In **Cybertron**, complex functions are broken down into “atomic” operations through propositions
360 on function compositions and computation graphs (Propositions 11,13,14,2). It is straightforward to
361 prove that these “atomic” operations are representable by transformers, either by feedforward layers
362 or attention layers. For example:

- 363 • **Feedforward layers:** boolean operations like AND (Proposition 6), OR (Proposition 7), or NOT
364 (Proposition 5), or operations over option types like `Option::or` (Proposition 9) being applied to
365 each token in a sequence.
- 366 • **Attention layers:** operations that requires information transmission between tokens such as
367 `nearest_left` and `nearest_right` that collect for each token the nearest left/right non-nil informa-
368 tion (Proposition 15).

369 This approach allows us to break down complex operations into primitive tasks that transformers
370 can simulate. Feedforward layers handle local operations on individual tokens, while attention lay-
371 ers manage long-range dependencies and interactions between tokens, simulating the multi-step
372 reasoning required for higher-level tasks.

373 **Cybertron**’s expressive type system and function composition framework help bridge the gap be-
374 tween the low-level processing transformers perform and the high-level reasoning necessary for Cy-
375 bertron’s type system and function composition. For full details, including the mathematical foundations of **Cy-**
376 **bertron**’s type system and function composition, see Appendix E.

377 6 Comparisons between Transformers and RNN

378 Now we compare transformers and RNNs from both theoretical and empirical perspectives.

379 6.1 A Lower Bound for RNNs for Type Checking

380 Previously, it has shown that RNN is provably less parameter efficient than transformers for associate
381 recall (Wen et al., 2024). Intuitively speaking, type checking step covers associate recall. Based on
382 this observation, we obtain the following lower bound for RNNs.

383 **Theorem 4.** For $L, D, H \in \mathbb{N}$, for any RNN that represents a function that maps any token sequence
384 of length L in $\text{MiniHuskyAnnotated}_{D,H}$ with $D, H = O(1)$ to its type errors represented as a
385 sequence of values of type `Option<TypeError>`, then its state space size is at least $\Omega(L)$.

386 Theorem 3 and Theorem 4 give a clear separation between transformers and RNNs in terms of the
387 compilation capability. Specifically, if the input codes satisfy $D, H \ll L$, which typically the case
388 under the clean code principle (Martin, 2008), then transformers at most need $O((\log L + D + H))$
389 number of parameters, which is significantly smaller what RNNs requires, $\Omega(L)$.

390 6.2 Empirical Comparison between Transformers and RNNs

391 We validate our theoretical results by conducting experiments on synthetic data.

392 **Dataset construction.** The synthetic dataset is parameterized by n (the number of data pieces), f
393 (the number of functions in a data piece), d (the minimum distance between the declaration and
394 the first call of a function, as well as the minimum distance between its consecutive calls), v (the
395 probability of using a variable in a function call), and e (the error rate of using an incorrect type in
396 a function call).

397 The names of the functions are drawn randomly from fx where $x \in \{0, 1, \dots, 99\}$. For each
398 function, there is only one argument whose symbol is randomly drawn from $\{a, b, \dots, z\}$ and
399 whose type is randomly drawn from $\{\text{Int}, \text{Float}, \text{Bool}\}$. There are at most 5 function calls in
400 a function, and those called functions must be declared and not called by at least d functions ahead
401 of the current one. In each function call, with probability v , the argument variable of the enclosing
402 function is used regardless of its type, with probability $(1 - v)(1 - e)$, a literal of the correct type
403 is used, and with probability $(1 - v)e$ an incorrect type literal is used. For integers, the literals are
404 from $\{0, 1, \dots, 99\}$; for floats, the literals are from $\{0.1, 1.1, \dots, 99.1\}$; for booleans, the literals
405 are from $\{\text{true}, \text{false}\}$.

406 Below is a data piece with $f = 10, d = 3, v = 0.2, e = 0.5$:

```
407 1 fn f2 ( w : Float ) { } fn f12 ( f : Float ) { } fn f98 ( l : Bool ) { } fn f74 ( f : Int  
408   ) { f2 ( f ) ; } fn f84 ( b : Float ) { f12 ( false ) ; } fn f59 ( h : Int ) { f98 ( h ) ; }  
409   fn f25 ( n : Int ) { f2 ( n ) ; f74 ( n ) ; } fn f81 ( a : Float ) { f84 ( 52.1 ) ; }  
410   fn f85 ( a : Bool ) { f12 ( false ) ; f98 ( 0 ) ; }
```

411 **Model and training.** We use customized BERT models (Devlin et al., 2019) and bidirectional RNN
412 models (Schuster & Paliwal, 1997) in our experiments. To control the model size (i.e., the number of
413 trainable parameters), we adjust only the hidden sizes while keeping other hyperparameters constant.
414 Detailed model specifications can be found in Table 1. For both transformers and RNNs, we use the
415 hyperparameters listed in Table 2 in Appendix during the training process.

416 **Results.** We experimented with multiple combinations of models (Table 1) and datasets (Table 2).
417 For each combination, we conducted independent runs using a fixed set of $k = 5$ random seeds.
418 When plotting the figures, we took the top $t = 5$ evaluation losses/accuracies from each run and
419 averaged over all the $k \times t$ values. We plotted separate figures for each dataset and separate sub-
420 figures for each metric. In each sub-figure, the x -axis represents the number of trainable parameters,
421 and the y -axis represents the averaged values. Results are shown in Figure 2. They demonstrate
422 that customized BERT models are able to perform better at type checking than bidirectional RNN
423 models when both sizes scale up, corroborating our theories. Other results are in Appendix J.

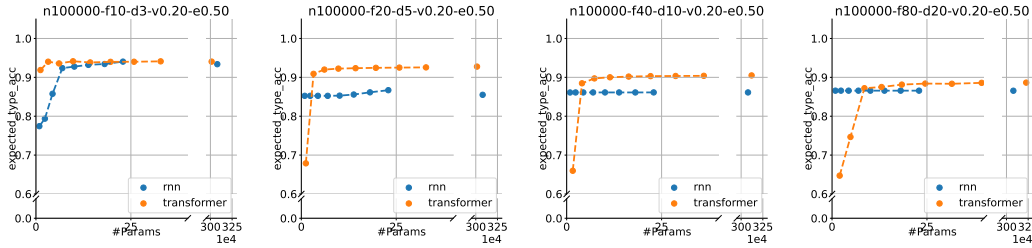


Figure 2: Figures depicting the evaluation accuracy of the expected type (see Section 5.3) across different models, measured by their number of trainable parameters, when trained on various datasets. The first 8 points of each model in each experiment are not aligned with the x -axis because the number of trainable parameters scales with hidden sizes differently for different models.

424 7 Conclusion

425 We demonstrated that transformers can efficiently handle syntactic and semantic analysis in C-like
 426 languages, using Cybertron to prove their capacity for tasks like AST generation, symbol resolution,
 427 and type analysis. We show a theoretical advantage of transformers over RNNs, particularly in
 428 their ability to manage long-range dependencies with logarithmic parameter scaling. In a sense,
 429 transformers have the right inductive bias for language tasks. Our experiments confirmed these
 430 theoretical insights, showing strong performance on synthetic and real datasets, underscoring the
 431 expressiveness and efficiency of transformers in sequence-based learning.

432 References

- 433 Cursor: Ai-powered code editor, 2024. URL <https://www.cursor.com/>. Accessed: Septem-
 434 ber 29, 2024.
- 435 Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. *Structure and Interpretation of*
 436 *Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition, 1996. ISBN
 437 0-262-01153-0.
- 438 Agda Development Team. *Agda compilers manual v2.6.4.2*, 2024. URL [https://agda.readthedocs.io/en/v2.6.4.2/tools/compilers.html#](https://agda.readthedocs.io/en/v2.6.4.2/tools/compilers.html#javascript-backend)
 439 [javascript-backend](https://agda.readthedocs.io/en/v2.6.4.2/tools/compilers.html#javascript-backend).
- 441 Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algo-
 442 rithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*,
 443 2022.
- 444 V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools*.
 445 pearson Education, 2007.
- 446 Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Ramasesh, Am-
 447 brose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization
 448 in large language models. *arXiv preprint arXiv:2207.04901*, 2022.
- 449 Anysphere. Cursor, 2023. URL <https://www.cursor.com/features>.
- 450 Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri
 451 Rudra, and Christopher R’e. Zoology: Measuring and improving recall in efficient language
 452 models. *ArXiv*, abs/2312.04927, 2023. URL [https://api.semanticscholar.org/](https://api.semanticscholar.org/CorpusID:266149332)
 453 [CorpusID:266149332](https://api.semanticscholar.org/CorpusID:266149332).
- 454 Yu Bai, Fan Chen, Huan Wang, Caiming Xiong, and Song Mei. Transformers as statisticians: Prov-
 455 able in-context learning with in-context algorithm selection. *arXiv preprint arXiv:2306.04637*,
 456 2023.

- 457 Boaz Barak, Benjamin Edelman, Surbhi Goel, Sham Kakade, Eran Malach, and Cyril Zhang. Hid-
458 den progress in deep learning: Sgd learns parities near the computational limit. *Advances in*
459 *Neural Information Processing Systems*, 35:21750–21764, 2022.
- 460 S. Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to
461 recognize formal languages. In *Conference on Empirical Methods in Natural Language Process-*
462 *ing*, 2020a. URL <https://api.semanticscholar.org/CorpusID:222225236>.
- 463 S. Bhattamishra, Arkil Patel, Phil Blunsom, and Varun Kanade. Understanding in-context learning
464 in transformers and llms by learning to learn discrete functions. *ArXiv*, abs/2310.03016, 2023.
465 URL <https://api.semanticscholar.org/CorpusID:263620583>.
- 466 S. Bhattamishra, Michael Hahn, Phil Blunsom, and Varun Kanade. Separations in the represen-
467 tational capabilities of transformers and recurrent architectures. *ArXiv*, abs/2406.09347, 2024.
468 URL <https://api.semanticscholar.org/CorpusID:270440803>.
- 469 Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers
470 to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020b.
- 471 Satwik Bhattamishra, Arkil Patel, and Navin Goyal. On the computational power of transformers
472 and its implications in sequence modeling. *arXiv preprint arXiv:2006.09286*, 2020c.
- 473 Jonathan Brennan and John Tracy Hale. Hierarchical structure guides rapid linguistic pre-
474 dictions during naturalistic listening. *PLoS ONE*, 14, 2019. URL <https://api.semanticscholar.org/CorpusID:260538292>.
- 475
- 476 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared
477 Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large
478 language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- 479 David Chiang, Peter A. Cholak, and Anand Pillay. Tighter bounds on the expressivity of transformer
480 encoders. In *International Conference on Machine Learning*, 2023. URL <https://api.semanticscholar.org/CorpusID:256231094>.
- 481
- 482 Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Rozière, Jonas Gehring, Gabriele Syn-
483 naeve, and Hugh Leather. Meta large language model compiler: Foundation models of com-
484 piler optimization. *ArXiv*, abs/2407.02524, 2024. URL <https://api.semanticscholar.org/CorpusID:270924331>.
- 485
- 486 Valentin David. *Language Constructs for C++-like languages*. PhD thesis, University of Bergen,
487 2009.
- 488 Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal
489 transformers. *arXiv preprint arXiv:1807.03819*, 2018.
- 490 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep
491 bidirectional transformers for language understanding, 2019. URL <https://arxiv.org/abs/1810.04805>.
- 492
- 493 Nai Ding, Lucia Melloni, Xing Tian, and David Poeppel. Rule-based and word-level statistics-based
494 processing of language: insights from neuroscience. *Language, Cognition and Neuroscience*, 32:
495 570–575, 2017. URL <https://api.semanticscholar.org/CorpusID:46747073>.
- 496 Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas
497 Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An
498 image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint*
499 *arXiv:2010.11929*, 2020.
- 500 Jana Dunfield and Neelakantan R Krishnaswami. Sound and complete bidirectional typechecking
501 for higher-rank polymorphism with existentials and indexed types. *Proceedings of the ACM on*
502 *Programming Languages*, 3(POPL):1–28, 2019.
- 503 Benjamin L Edelman, Surbhi Goel, Sham Kakade, and Cyril Zhang. Inductive biases and variable
504 creation in self-attention mechanisms. In *International Conference on Machine Learning*, pp.
505 5793–5831. PMLR, 2022.

- 506 N Elhage, N Nanda, C Olsson, T Henighan, N Joseph, B Mann, A Askell, Y Bai, A Chen, T Conerly,
507 et al. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021.
- 508 Guhao Feng, Yuntian Gu, Bohang Zhang, Haotian Ye, Di He, and Liwei Wang. Towards revealing
509 the mystery behind chain of thought: a theoretical perspective. *ArXiv*, abs/2305.15408, 2023.
510 URL <https://api.semanticscholar.org/CorpusID:258865989>.
- 511 David Flanagan. *JavaScript: The definitive guide: Activate your web pages*. " O'Reilly Media,
512 Inc.", 2011.
- 513 Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings*
514 *of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp.
515 111–122, 2004.
- 516 S. Frank, Rens Bod, and Morten H. Christiansen. How hierarchical is language use? *Proceedings*
517 *of the Royal Society B: Biological Sciences*, 279:4522 – 4531, 2012. URL [https://api.](https://api.semanticscholar.org/CorpusID:11969171)
518 [semanticscholar.org/CorpusID:11969171](https://api.semanticscholar.org/CorpusID:11969171).
- 519 Dan Friedman, Alexander Wettig, and Danqi Chen. Learning transformer programs. *ArXiv*,
520 abs/2306.01128, 2023. URL [https://api.semanticscholar.org/CorpusID:](https://api.semanticscholar.org/CorpusID:259064324)
521 [259064324](https://api.semanticscholar.org/CorpusID:259064324).
- 522 Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn
523 in-context? a case study of simple function classes. *Advances in Neural Information Processing*
524 *Systems*, 35:30583–30598, 2022.
- 525 Google Deepmind. Ai achieves silver-medal standard solving international mathematical
526 olympiad problems, July 2024. URL [https://deepmind.google/discover/blog/](https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/)
527 [ai-solves-imo-problems-at-silver-medal-level/](https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/).
- 528 Qiuhan Gu. Llm-based code generation method for golang compiler testing. *Proceedings of the 31st*
529 *ACM Joint European Software Engineering Conference and Symposium on the Foundations of*
530 *Software Engineering*, 2023. URL [https://api.semanticscholar.org/CorpusID:](https://api.semanticscholar.org/CorpusID:265509921)
531 [265509921](https://api.semanticscholar.org/CorpusID:265509921).
- 532 Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions*
533 *of the Association for Computational Linguistics*, 8:156–171, 2019. URL [https://api.](https://api.semanticscholar.org/CorpusID:189928186)
534 [semanticscholar.org/CorpusID:189928186](https://api.semanticscholar.org/CorpusID:189928186).
- 535 Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong,
536 Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof
537 of the kepler conjecture. In *Forum of mathematics, Pi*, volume 5, pp. e2. Cambridge University
538 Press, 2017.
- 539 Sophie Hao, Dana Angluin, and Roberta Frank. Formal language recognition by hard attention
540 transformers: Perspectives from circuit complexity. *Transactions of the Association for Com-*
541 *putational Linguistics*, 10:800–810, 2022. URL [https://api.semanticscholar.org/](https://api.semanticscholar.org/CorpusID:248177889)
542 [CorpusID:248177889](https://api.semanticscholar.org/CorpusID:248177889).
- 543 John Harrison. Hol light: An overview. In *International Conference on Theorem Proving in Higher*
544 *Order Logics*, pp. 60–66. Springer, 2009.
- 545 John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In *Hand-*
546 *book of the History of Logic*, volume 9, pp. 135–214. Elsevier, 2014.
- 547 Samy Jelassi, David Brandfonbrener, Sham M. Kakade, and Eran Malach. Repeat after me: Trans-
548 formers are better than state space models at copying. *ArXiv*, abs/2402.01032, 2024. URL
549 <https://api.semanticscholar.org/CorpusID:267406617>.
- 550 Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.
- 551 Shuai Li, Zhao Song, Yu Xia, Tong Yu, and Tianyi Zhou. The closeness of in-context learning and
552 weight shifting for softmax regression. *arXiv preprint arXiv:2304.13276*, 2023.

- 553 Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. Chain of thought empowers transformers to
554 solve inherently serial problems. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=3EWTEy9MTM>.
555
- 556 Valerii Likhoshesterov, Krzysztof Choromanski, and Adrian Weller. On the expressive power of
557 self-attention matrices. *arXiv preprint arXiv:2106.03764*, 2021.
- 558 David Lindner, János Kramár, Matthew Rahtz, Tom McGrath, and Vladimir Mikulik. Tracr:
559 Compiled transformers as a laboratory for interpretability. *ArXiv*, abs/2301.05062, 2023. URL
560 <https://api.semanticscholar.org/CorpusID:255749093>.
- 561 Bingbin Liu, Jordan T. Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Trans-
562 formers learn shortcuts to automata. *ArXiv*, abs/2210.10749, 2022. URL <https://api.semanticscholar.org/CorpusID:252992725>.
563
- 564 Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR,
565 USA, 1 edition, 2008. ISBN 0132350882.
- 566 William Merrill, Ashish Sabharwal, and Noah A. Smith. Saturated transformers are constant-depth
567 threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856,
568 2021. URL <https://api.semanticscholar.org/CorpusID:248085924>.
- 569 Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2:
570 Lessons for training llms on programming and natural languages. *ICLR*, 2023.
- 571 Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan,
572 Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction
573 heads. *arXiv preprint arXiv:2209.11895*, 2022.
- 574 OpenAI. Openai o1 system card, September 2024a. URL [https://openai.com/index/
575 openai-o1-system-card/](https://openai.com/index/openai-o1-system-card/).
- 576 OpenAI. Sora: Creating video from text, February 2024b. URL [https://openai.com/
577 index/sora/](https://openai.com/index/sora/).
- 578 Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.
- 579 Jorge Pérez, Javier Marinkovic, and Pablo Barceló. On the turing completeness of mod-
580 ern neural network architectures. *ArXiv*, abs/1901.03429, 2019. URL <https://api.semanticscholar.org/CorpusID:57825721>.
581
- 582 Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing complete. *The Journal of*
583 *Machine Learning Research*, 22(1):3463–3497, 2021.
- 584 Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.
- 585 The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics.
586 *arXiv preprint arXiv:1308.0729*, 2013.
- 587 Reddit User. I think the main secret sauce of o1 is the data. [https://www.reddit.com/
588 r/singularity/comments/lfi6yy9/i_think_the_main_secret_sauce_of_
589 o1_is_the_data/](https://www.reddit.com/r/singularity/comments/lfi6yy9/i_think_the_main_secret_sauce_of_o1_is_the_data/), 2013. Accessed: 2024-09-28.
- 590 Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions*
591 *on Signal Processing*, 45(11):2673–2681, 1997.
- 592 Lena Strobl. Average-hard attention transformers are constant-depth uniform threshold cir-
593 cuits. *ArXiv*, abs/2308.03212, 2023. URL [https://api.semanticscholar.org/
594 CorpusID:260680416](https://api.semanticscholar.org/CorpusID:260680416).
- 595 Victor Taelin. Ai and the future of coding. [https://medium.com/jonathans-musings/
596 ai-and-the-future-of-coding-43caad31c3d3](https://medium.com/jonathans-musings/ai-and-the-future-of-coding-43caad31c3d3), 2023a. Accessed: 2024-10-01.
- 597 Victor Taelin. Agda to typescript compilation with sonnet-3.5, 2023b. URL [https://x.com/
598 VictorTaelin/status/1837925011187027994](https://x.com/VictorTaelin/status/1837925011187027994). Accessed: September 29, 2024.

- 599 A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.
- 600 Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordv-
601 intsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient
602 descent. *arXiv preprint arXiv:2212.07677*, 2022.
- 603 Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Huai hsin Chi, and Denny Zhou. Self-
604 consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171,
605 2022. URL <https://api.semanticscholar.org/CorpusID:247595263>.
- 606 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc
607 Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language mod-
608 els. *ArXiv*, abs/2201.11903, 2022. URL <https://api.semanticscholar.org/CorpusID:246411621>.
- 610 Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. *ArXiv*, abs/2106.06981,
611 2021. URL <https://api.semanticscholar.org/CorpusID:235421630>.
- 612 Kaiyue Wen, Xingyu Dang, and Kaifeng Lyu. Rnns are not transformers (yet): The key
613 bottleneck on in-context retrieval. *ArXiv*, abs/2402.18510, 2024. URL <https://api.semanticscholar.org/CorpusID:268041425>.
- 615 Shangda Wu, Xu Tan, Zili Wang, Rui Wang, Xiaobing Li, and Maosong Sun. Beyond language
616 models: Byte models are digital world simulators. *ArXiv*, abs/2402.19155, 2024. URL <https://api.semanticscholar.org/CorpusID:268063492>.
- 618 Shunyu Yao, Binghui Peng, Christos H. Papadimitriou, and Karthik Narasimhan. Self-attention
619 networks can process bounded hierarchical languages. In *Annual Meeting of the Association for Computational Linguistics*, 2021. URL <https://api.semanticscholar.org/CorpusID:235166395>.
- 622 Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar.
623 Are transformers universal approximators of sequence-to-sequence functions? *arXiv preprint*
624 *arXiv:1912.10077*, 2019.
- 625 Haoyu Zhao, Abhishek Panigrahi, Rong Ge, and Sanjeev Arora. Do transformers parse while pre-
626 dicting the masked word? *arXiv preprint arXiv:2303.08117*, 2023.
- 627 Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Ben-
628 gio, and Preetum Nakkiran. What algorithms can transformers learn? a study in length gener-
629 alization. *ArXiv*, abs/2310.16028, 2023. URL <https://api.semanticscholar.org/CorpusID:264439160>.
- 630

631 A Tree

632 To define a syntax tree, one commonly resorts to generation rules, such as context-free grammars
633 (CFG) (Alfred et al., 2007) and parsing expression grammars (PEG) (Ford, 2004). In most cases,
634 just generation rules themselves are not sufficient to define properly a language. Many practical
635 languages like C and C++ cannot be solely described by these rules (David, 2009). Furthermore, se-
636 mantic constraints like type correctness are intrinsically contextual and cannot be expressed through
637 CFG or similar rules. However, CFG or other rules provide a valuable construct, the AST. With
638 an AST, one can refine the language definition by putting restrictions on the syntax tree through
639 tree operations. Effectively, a language can be seen as a subset of trees, not as a subset of strings.
640 Semantic analysis like symbol resolution and type checking can be described effectively based on
641 trees.

642 A.1 What are Trees

643 Trees in data structures have slightly additional meaning as compared to trees in mathematics. In
644 this paper, all trees are trees in data structures. For clarity, we lay down the precise definition of
645 trees in data structure.

646 **Definition 3** (Tree). *A tree T is a set of nodes storing elements such that the nodes have a parent-
647 child relationship that satisfies the following:*

- 648 • *If T is not empty, it has a special node called the **root** that has no parent.*
- 649 • *Each node v of T other than the root has a unique parent node w ; each node with parent
650 w is a child of w .*

651 *We denote the nodes of T as $N(T)$.*

652 **Definition 4** (Recursive Definition of a Tree). *A tree T is either empty or consists of a node r (the
653 root) and a possibly empty set of trees whose roots are the children of r .*

654 However, the above definition is too permissive. We shall define a typed version as follows:

655 **Definition 5** (Typed Tree). *A tree type consists of a set of values V and a set of relationships
656 $C \subseteq V \times \mathbb{N}$, and a typed tree under this type is any tree T such that for each node, a value $v \in V$
657 is assigned such that $(v, n) \in C$ where n is the number of the children of the node.*

658 **All trees in this paper are typed.**

659 **Example 1** (AST as Typed Tree). *Consider an AST for a simple arithmetic expression. Let the set
660 of values V be:*

$$V = \{ \text{num}, \text{add}, \text{sub}, \text{mul}, \text{div} \}$$

661 *and the set of relationships $C \subseteq V \times \mathbb{N}$ specify the allowed number of children for each value:*

$$C = \{ (\text{num}, 0), (\text{add}, 2), (\text{sub}, 2), (\text{mul}, 2), (\text{div}, 2) \}$$

662 *An example AST for the arithmetic expression $(3 + 5) \times 2$ is the following typed tree:*

- 663 • *The root node is labeled **mul** (multiplication), and it has two children.*
 - 664 – *The left child is labeled **add** (addition), and it has two children:*
 - 665 * *The left child of **add** is labeled **num** with the value 3.*
 - 666 * *The right child of **add** is labeled **num** with the value 5.*
 - 667 – *The right child of **mul** is labeled **num** with the value 2.*

668 *This tree conforms to the typing rules because:*

- 669 • ***num** has 0 children,*
- 670 • ***add** has 2 children,*
- 671 • ***mul** has 2 children,*

672 *all of which satisfy the relationships in C .*

673 A.2 Representations of Trees

674 It's also important to talk about tree representations. We are studying transformers, and then it's
675 necessary to represent large trees as a sequence, otherwise the model dimension is not large enough
676 to contain the information locally. Let's first talk about the classical **arena pattern** used in sys-
677 tem programming for representing trees and we shall slightly adapt it to our use case for studying
678 transformers.

679 **Arena Pattern.** To represent trees efficiently in memory, especially when trees are frequently
680 modified (such as insertions or deletions of nodes), an arena pattern is often used. The arena pattern
681 provides a way to manage memory allocation for tree structures, allowing for efficient memory usage
682 and avoiding fragmentation. Here's how the arena pattern works in the context of tree representation:

683 **Definition 6** (Arena Pattern in Tree Representation). *In the arena pattern, a tree is represented by*
684 *an array (or vector) of nodes, called an **arena**. Each node in the arena contains:*

- 685 • *An element or value stored in the node.*
- 686 • *References (often indices or pointers) to the node's children and possibly to its parent.*

687 *The key characteristics of the arena pattern are:*

- 688 • *Memory Contiguity: All nodes are stored contiguously in memory within the arena, which*
689 *allows for efficient traversal and modification operations.*
- 690 • *Fixed Capacity: The arena has a fixed or dynamically resizable capacity, and nodes are*
691 *added sequentially. This avoids the overhead of allocating individual nodes on the heap.*
- 692 • *Index-based References: Instead of using pointers, the nodes reference each other using*
693 *indices within the array, which simplifies memory management and can lead to cache-*
694 *friendly operations.*
- 695 • *Efficient Allocation and Deallocation: Nodes can be efficiently allocated and deallocated*
696 *within the arena without requiring complex memory management techniques like garbage*
697 *collection or reference counting.*

698 The arena pattern is particularly useful in scenarios where the structure of the tree is highly dynamic
699 or when performance is critical. It allows for a simple and efficient way to manage and traverse trees
700 without the typical overhead associated with more traditional pointer-based tree representations.

701 **Adaptations for Transformers** For transformers, inputs, intermediate values and outputs are all
702 sequences. So the trees are represented as sequences of nodes with node reference representable by
703 token position encoding.

704 B Context Free Grammar

705 In this section, we lay down the well-known definitions of context free grammar, derivations, and
706 parse trees.

707 A context-free grammar (CFG) is defined as a 4-tuple $G = (V, \Sigma, R, S)$, where:

- 708 • V is a finite set of variables (non-terminal symbols).
- 709 • Σ is a finite set of terminal symbols, disjoint from V . Sequences of Σ , i.e., elements of Σ^*
710 are called strings.
- 711 • $R \subset V \times (V \cup \Sigma)^*$ is a finite set of production rules, where each rule is of the form $A \rightarrow \alpha$,
712 with $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.
- 713 • $S \in V$ is the start symbol.

714 Given a context-free grammar $G = (V, \Sigma, R, S)$, we define derivation as follows:

- 715 • A **derivation** is a sequence of steps where, starting from the start symbol S , each step
716 replaces a non-terminal with the right-hand side of a production rule.
- 717 • Formally, we write $u \Rightarrow v$ if $u = \alpha A \beta$ and $v = \alpha \gamma \beta$ for some production $A \rightarrow \gamma$ in R ,
718 where $\alpha, \beta \in (V \cup \Sigma)^*$ and $A \in V$.
- 719 • A **leftmost derivation** is a derivation in which, at each step, the leftmost non-terminal is
720 replaced.
- 721 • A **rightmost derivation** is a derivation in which, at each step, the rightmost non-terminal
722 is replaced.
- 723 • We denote a **derivation sequence** as $S \Rightarrow^* w$, where $w \in \Sigma^*$ is a string derived from S in
724 zero or more steps.

725 A **parse tree** (or **syntax tree**) for a context-free grammar $G = (V, \Sigma, R, S)$ is a tree that satisfies
726 the following conditions:

- 727 • The root of the tree is labeled with the start symbol S .
- 728 • Each leaf of the tree is labeled with a terminal symbol from Σ or the empty string ϵ .
- 729 • Each internal node of the tree is labeled with a non-terminal symbol from V .
- 730 • If an internal node is labeled with a non-terminal A and has children labeled with
731 X_1, X_2, \dots, X_n , then there is a production rule $A \rightarrow X_1 X_2 \dots X_n$ in R .
- 732 • The yield of the parse tree, which is the concatenation of the labels of the leaves (in left-to-
733 right order), forms a string in Σ^* that is derived from the start symbol S .

734 C Neural Architectures

735 In this section, we lay down the precise mathematical definitions of neural architectures we are going
736 to use in our proof.

737 **Definition 7** (Single-Layer Fully Connected Network with $4 \times$ Intermediate Space).

738 *Given model dimension d_{model} , a single-layer feed-forward network with an intermediate space ex-
739 panded to 4 times the input dimension is a function from $\mathbb{R}^{d_{model}}$ to $\mathbb{R}^{d_{model}}$, denoted by f_{fcn} and defined
740 as follows:*

741 *given $X \in \mathbb{R}^{d_{model}}$, weights $W_1 \in \mathbb{R}^{4d_{model} \times d_{model}}$, $W_2 \in \mathbb{R}^{d_{model} \times 4d_{model}}$, and biases $B_1 \in \mathbb{R}^{4d_{model}}$,
742 $B_2 \in \mathbb{R}^{d_{model}}$, the output $f_{fcn}(X)$ is computed as:*

$$f_{fcn}(X) = W_2 \sigma_{\text{ReLU}}(W_1 X + B_1) + B_2,$$

743 *where $\sigma_{\text{ReLU}} : \mathbb{R}^{4d_{model}} \rightarrow \mathbb{R}^{4d_{model}}$ is the Rectified Linear Unit activation function applied element-
744 wise, defined by:*

$$\sigma_{\text{ReLU}}(z) = (\max(z_1, 0), \max(z_2, 0), \dots, \max(z_{4d_{model}}, 0))^{\top},$$

745 *for $z = (z_1, z_2, \dots, z_{4d_{model}})^{\top} \in \mathbb{R}^{4d_{model}}$.*

746 The choice of a $4 \times$ intermediate space is common in practice, often used in Transformer architec-
747 tures. Interestingly, this empirical choice turns out to have a useful theoretical property: it allows
748 the network to express any affine transformation, as we'll see in the following proposition.

749 **Proposition 1.** *A single-layer fully connected network with a $4 \times$ intermediate space, as defined
750 previously, can express any affine map from $\mathbb{R}^{d_{model}}$ to $\mathbb{R}^{d_{model}}$.*

751 *Proof.* Let $f : \mathbb{R}^{d_{model}} \rightarrow \mathbb{R}^{d_{model}}$ be any affine map given by $f(X) = AX + b$, where $A \in \mathbb{R}^{d_{model} \times d_{model}}$
752 and $b \in \mathbb{R}^{d_{model}}$. We will construct weights $W_1 \in \mathbb{R}^{4d_{model} \times d_{model}}$, $W_2 \in \mathbb{R}^{d_{model} \times 4d_{model}}$ and biases
753 $B_1 \in \mathbb{R}^{4d_{model}}$, $B_2 \in \mathbb{R}^{d_{model}}$ such that $f_{fcn}(X) = f(X)$ for all $X \in \mathbb{R}^{d_{model}}$.

754 Define:

$$W_1 = \begin{pmatrix} I_{d_{\text{model}}} \\ -I_{d_{\text{model}}} \\ 0 \\ 0 \end{pmatrix}, \quad B_1 = 0 \in \mathbb{R}^{4d_{\text{model}}},$$

755 where $I_{d_{\text{model}}}$ is the $d_{\text{model}} \times d_{\text{model}}$ identity matrix, and 0 represents zero matrices of appropriate
756 dimensions. Set:

$$W_2 = (A \quad -A \quad 0 \quad 0), \quad B_2 = b.$$

757 For any $X \in \mathbb{R}^{d_{\text{model}}}$, compute:

$$\begin{aligned} f_{\text{fcn}}(X) &= W_2 \sigma_{\text{ReLU}}(W_1 X + B_1) + B_2 \\ &= (A \quad -A \quad 0 \quad 0) \sigma_{\text{ReLU}} \left(\begin{pmatrix} X \\ -X \\ 0 \\ 0 \end{pmatrix} \right) + b \\ &= (A \quad -A \quad 0 \quad 0) \begin{pmatrix} \sigma_{\text{ReLU}}(X) \\ \sigma_{\text{ReLU}}(-X) \\ 0 \\ 0 \end{pmatrix} + b \\ &= A \sigma_{\text{ReLU}}(X) - A \sigma_{\text{ReLU}}(-X) + b. \end{aligned}$$

758 Noting that $\sigma_{\text{ReLU}}(X) - \sigma_{\text{ReLU}}(-X) = X$ (applied element-wise), we have:

$$f_{\text{fcn}}(X) = AX + b = f(X).$$

759 Therefore, the network can represent any affine map from $\mathbb{R}^{d_{\text{model}}}$ to $\mathbb{R}^{d_{\text{model}}}$. \square

760 **Definition 8** (Single-Layer Feed Forward Network with $4 \times$ Intermediate Space). Given model
761 dimension d_{model} and position set Pos, the Transformer Feed Forward Network is a function
762 $f_{\text{ffn}} : \mathbb{R}^{\text{Pos} \times d_{\text{model}}} \rightarrow \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$ defined as follows:

763 For an input $X \in \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$, the output $f_{\text{ffn}}(X)$ is computed by applying the single-layer feed-
764 forward network f_{fcn} (as defined previously) independently to each position:

$$f_{\text{ffn}}(X)_p = f_{\text{fcn}}(X_p) \quad \forall p \in \text{Pos}$$

765 where $X_p \in \mathbb{R}^{d_{\text{model}}}$ is the p -th row of X , corresponding to the p -th position in the input sequence.

766 Next, we define the attention mechanism, which is a key component of the Transformer architecture.
767 This definition presents a hard attention layer with a simplified position encoding. We use hard
768 attention here for theoretical simplicity, as it represents a discrete limit of the more commonly used
769 soft attention mechanism. Hard attention forces the model to make a clear choice about which inputs
770 to focus on, which can simplify analysis and provide clearer insights into the model's behavior. It
771 can be viewed as the limiting case of soft attention as the temperature approaches zero, where the
772 softmax operation becomes increasingly peaked and eventually converges to a one-hot vector.

773 **Definition 9** (Hard Attention Layer with Simplified Position Encoding). Given model dimension
774 d_{model} , number of heads H , and number of layers L , a transformer with simplified position encoding
775 and hard attention is defined to be a function $f_{\text{attn}} : \mathbb{R}^{\text{Pos} \times d_{\text{model}}} \rightarrow \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$ defined by

$$\forall p \in \text{Pos}, f_{\text{attn}}(X)_p := W_O \text{Concat} \left(\text{Attn}^{(1)}(X)_p, \dots, \text{Attn}^{(H)}(X)_p \right), \quad (4)$$

776 where the h th attention head uses hard attention, defined as:

$$\text{Attn}^{(h)}(X)_p := \frac{1}{|S_p|} \sum_{p' \in S_p} V_{p'}^{(h)}, \quad (5)$$

777 where

778 • $W_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ are trainable parameters;

- 779 • $S_p = \arg \max_{p' \in \text{Pos}} \left(Q_p^{(h)\top} K_{p'}^{(h)} + \lambda^{(h)\top} \Psi_{p'-p} \right)$ with $Q_p^{(h)}, K_p^{(h)}, V_p^{(h)}, \lambda^{(h)}, \Psi_q$ de-
780 fined by
- 781 – $Q_p^{(h)} = W_Q^{(h)} X_p, K_p^{(h)} = W_K^{(h)} X_p$ are vectors of dimension d_{model}/H , with trainable
782 parameters $W_Q^{(h)}, W_K^{(h)} \in \mathbb{R}^{(d_{\text{model}}/H) \times d_{\text{model}}}$;
- 783 – $V_p^{(h)} = W_V^{(h)} X_p$ are vectors of dimension d_{model}/H , linear transformations of X_p
784 with trainable parameters $W_V^{(h)} \in \mathbb{R}^{(d_{\text{model}}/H) \times d_{\text{model}}}$;
- 785 – $\lambda^{(h)} \in \mathbb{R}^2$ are constants depending only on head count h ;
- 786 – $\Psi_q \in \mathbb{R}^2$ are 2-dimensional vectors depending on relative position q but not on head
787 count h . It is explicitly defined as

$$\Psi_q = \begin{pmatrix} q \\ \mathbf{1}_{q>0} \end{pmatrix}. \quad (6)$$

788 This formulation allows for both past and future masking.

789 Having defined the basic components, we can now proceed to describe the full Transformer archi-
790 tecture. This definition builds upon the previously introduced concepts, incorporating them into a
791 complete model structure.

792 **Definition 10** (Transformer). A **Transformer** is a function $f_{\text{tf}}: \mathbb{R}^{\text{Pos} \times d_{\text{model}}} \rightarrow \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$ that maps
793 an input sequence to an output sequence through a series of layers, each consisting of a multi-head
794 attention mechanism and a position-wise feed-forward network (MLP).

795 Given:

- 796 • Input sequence $X \in \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$, where Pos is the set of positions and d_{model} is the model
797 dimension.
- 798 • Number of layers L .
- 799 • Number of attention heads H .

800 The Transformer computes the output $Y = X^{(L)}$ through recursive application of attention and
801 feed-forward layers:

- 802 • Initialization is given by:

$$X^{(0)} = X.$$

- 803 • For each layer $l = 1, 2, \dots, L$:

- 804 – Compute attention output:

$$\hat{X}^{(l)} = X^{(l-1)} + f_{\text{attn}}^{(l)} \left(X^{(l-1)} \right)$$

- 805 – Compute feed-forward output:

$$X^{(l)} = \hat{X}^{(l)} + f_{\text{ffn}}^{(l)} \left(\hat{X}^{(l)} \right)$$

806 Here:

- 807 • $f_{\text{attn}}^{(l)}$ are **hard attention layers with simplified position encoding** as previously defined. It
808 operates on $X^{(l-1)}$ and produces an output in $\mathbb{R}^{\text{Pos} \times d_{\text{model}}}$.
- 809 • $f_{\text{ffn}}^{(l)}$ are **feed-forward networks with $4 \times$ intermediate space** as previously defined. It oper-
810 ates position-wise on $\hat{X}^{(l)}$ and produces an output in $\mathbb{R}^{\text{Pos} \times d_{\text{model}}}$.

811 *Remark 1.* For simplicity, we have omitted the Layer Normalization component typically present
812 in Transformer architectures. This simplification allows us to focus on the core attention and feed-
813 forward mechanisms while maintaining the essential structure of the Transformer.

814 We use $\text{Trf}_{H,L}^{d_{\text{model}}}$ to denote the set of transformers of model size d_{model} , number of heads H and number
815 of layers L as functions from $\mathbb{R}^{d_{\text{model}}}$ to $\mathbb{R}^{d_{\text{model}}}$.

816 For purpose of proof, we shall also need residual multi-layer perceptron. Functions over local types
817 are first represented by multi-layer perceptron, then by Proposition 2 applications of these func-
818 tions over sequences can be representable by transformers. Residual multi-layer perceptron can be
819 assembled through composition or computer graph, as we shall see.

820 Here's the definition of a residual MLP Network.

821 **Definition 11** (Residual Multi-Layer Perceptron). A Residual Multi-Layer Perceptron (*ResMLP*) is
822 a function $f_{\text{resmlp}} : \mathbb{R}^{d_{\text{model}}} \rightarrow \mathbb{R}^{d_{\text{model}}}$ defined recursively by

$$X^{(0)} = X, \quad X^{(l)} = X^{(l-1)} + f_{\text{fcn}} \left(X^{(l-1)} \right), \quad l = 1, 2, \dots, L, \quad f_{\text{resmlp}}(X) = X^{(L)}$$

823 where $X \in \mathbb{R}^{d_{\text{model}}}$ is the input vector, L is the total number of layers, and $f_{\text{fcn}} : \mathbb{R}^{d_{\text{model}}} \rightarrow \mathbb{R}^{d_{\text{model}}}$
824 is the Single-Layer Fully Connected Network with $4 \times$ Intermediate Space as previously defined in
825 Definition 1.

826 We use $\text{ResMLP}_L^{d_{\text{model}}} \subset \mathbb{R}^{d_{\text{model}}} \mathbb{R}^{d_{\text{model}}}$ to represent the set of residual MLPs with dimension d_{model} and
827 L layers, as defined in Definition 11.

828 The following proposition is quite basic. It demonstrates that any function representable by a
829 ResMLP can be applied position-wise by a Transformer.

830 **Proposition 2** (Position-wise ResMLP Application is Representable by Transformers). Let $f : \mathbb{R}^{d_{\text{model}}} \rightarrow \mathbb{R}^{d_{\text{model}}}$
831 be a function representable by a Residual Multi-Layer Perceptron (*ResMLP*) as
832 defined in Definition 11. Then the function $F : \mathbb{R}^{\text{Pos} \times d_{\text{model}}} \rightarrow \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$, defined by applying f
833 position-wise,

$$F(X)_p = f(X_p), \quad \forall p \in \text{Pos},$$

834 is representable by a Transformer as defined in Definition 10.

835 *Proof.* Since f is representable by a ResMLP with L layers, it is defined recursively by

$$X^{(0)} = X, \quad X^{(l)} = X^{(l-1)} + f_{\text{fcn}} \left(X^{(l-1)} \right) \text{ for } l = 1, \dots, L,$$

836 and

$$f(X) = X^{(L)},$$

837 where $f_{\text{fcn}} : \mathbb{R}^{d_{\text{model}}} \rightarrow \mathbb{R}^{d_{\text{model}}}$ is the Single-Layer Fully Connected Network with $4 \times$ intermediate
838 space (Definition 1).

839 We construct a Transformer with L layers such that, for any input sequence $X \in \mathbb{R}^{\text{Pos} \times d_{\text{model}}}$, the
840 output $Y = f_{\text{tf}}(X)$ satisfies $Y_p = f(X_p)$ for all $p \in \text{Pos}$.

841 To achieve this, we configure the Transformer so that the attention mechanism outputs zero at each
842 layer. This can be done by setting the attention weights to zero, ensuring $f_{\text{attn}}(X^{(l-1)}) = 0$. Conse-
843 quently, the update equations simplify to

$$\hat{X}^{(l)} = X^{(l-1)}.$$

844 We then set the feed-forward network f_{ffn} in the Transformer to have the same weights and biases
845 as f_{fcn} in the ResMLP. The Transformer layer update becomes

$$X^{(l)} = \hat{X}^{(l)} + f_{\text{ffn}} \left(\hat{X}^{(l)} \right) = X^{(l-1)} + \left(f_{\text{fcn}} \left(X_p^{(l-1)} \right) \right)_{p \in \text{Pos}}.$$

846 This recursion matches that of the ResMLP applied position-wise to X . Therefore, after L layers,
847 the Transformer output satisfies $f_{\text{tf}}(X)_p = f(X_p)$ for all $p \in \text{Pos}$.

848 □

849 D Mini-Husky Details

850 Here's the BNF of the Mini-Husky language:

```
⟨ast⟩ ::= ⟨literal⟩
        | ⟨ident⟩
        | ⟨prefix⟩
        | ⟨binary⟩
        | ⟨suffix⟩
        | ⟨delimited⟩
        | ⟨separated_item⟩
        | ⟨call⟩
        | ⟨let_init⟩
        | ⟨if_stmt⟩
        | ⟨else_stmt⟩
        | ⟨defn⟩

⟨literal⟩ ::= ...
⟨ident⟩ ::= ...
⟨prefix⟩ ::= ⟨prefix_opr⟩ ⟨ast⟩
⟨binary⟩ ::= ⟨ast⟩ ⟨binary_opr⟩ ⟨ast⟩
⟨suffix⟩ ::= ⟨ast⟩ ⟨suffix_opr⟩
⟨delimited⟩ ::= ⟨left_delimiter⟩ ⟨separated_item⟩* ⟨right_delimiter⟩
⟨separated_item⟩ ::= [⟨ast⟩] ⟨separator⟩
⟨call⟩ ::= ⟨ast⟩ ⟨left_delimiter⟩ ⟨ast⟩* ⟨right_delimiter⟩
⟨let_init⟩ ::= let ⟨ast⟩
⟨if_stmt⟩ ::= if ⟨ast⟩ ⟨delimited⟩
⟨else_stmt⟩ ::= ⟨if_stmt⟩ else (⟨delimited⟩ | ⟨else_stmt⟩)
⟨defn⟩ ::= ⟨defn_keyword⟩ ⟨ident⟩ ⟨ast⟩
⟨prefix_opr⟩ ::= +|-|!|...
⟨binary_opr⟩ ::= +|-|*|/|...
⟨suffix_opr⟩ ::= ++|--|...
⟨left_delimiter⟩ ::= '('|'|{|
⟨right_delimiter⟩ ::= ')'|'|}|
⟨separator⟩ ::= ,|;
⟨defn_keyword⟩ ::= def|fn|...
```

851

852 Below is a sample piece of codes:

```
853 1 struct Dog { weight: f32, .. }
854 2
855 3 fn see_vet(dog: Dog) -> f32 {
856 4     assert dog.weight < 100;
857 5     let mut fee = dog.weight * 10.0;
858 6     fee +=100.0;
859 7     return fee
860 8 }
```

861 It should be noted that the above is not the full story. There are additional constraints put on the
862 ASTs. However, these can be easily implemented as tree functions that are easy for transformers to
863 express. As we are focusing on higher level language processing capabilities, we ignore the details
864 here.

865 Additionally, we need to require that for semantic correctness, we must have proper symbol resolu-
866 tion and type correctness.

867 D.1 Additional Details about Compiler Tasks.

868 The outputs of the tasks are defined using Cybertron as follows:

- 869 • The construction of AST task's final output is the collection all AST nodes. More concretely,
870 the output is a sequence of `Option<Ast>` with length equal to the input token sequence's length,
871 where `Option<Ast>` denoted the type `Ast` will a null value added and `Ast` is the type storing the
872 information of a node, including its parent, and its data of type `AstData`. In Cybertron, we define
873 `Ast` and `AstData` explicitly as follows:

```
874 1 /// Represents a node in an Abstract Syntax Tree (AST).  
875 2 ///  
876 3 /// Each 'Ast' node has a reference to its parent node (if any) and holds  
877 4 /// the associated syntax data (such as expressions, statements, or other  
878 5 /// constructs defined in the 'AstData' enum).  
879 6 pub struct Ast {  
880 7     /// The index of the parent node in the AST, if it exists.  
881 8     ///  
882 9     /// - 'Some(Idx)': The node has a parent, and 'Idx' represents its position.  
883 10    /// - 'None': The node is the root or does not have a parent.  
884 11    pub parent: Option<Idx>,  
885 12    /// The data associated with this AST node.  
886 13    pub data: AstData,  
887 14 }  
888 15  
889 16 /// Enumeration representing different types of Abstract Syntax Tree (AST) nodes  
890 17 pub enum AstData {  
891 18     /// Represents a literal value (e.g., integer, string)  
892 19     Literal(Literal),  
893 20     /// Represents an identifier (e.g., variable name)  
894 21     Ident(Ident),  
895 22     /// Represents a binary expression (e.g., 'x + y', 'a * b')  
896 23     Binary {  
897 24         /// Index of the left operand  
898 25         l opd: Idx,  
899 26         /// Operator in the binary expression (e.g., '+', '*')  
900 27         opr: BinaryOpr,  
901 28         /// Index of the right operand  
902 29         ropd: Idx,  
903 30     },  
904 31     ... // other variants  
905 32 }
```

- 906 • The output of the **symbol resolution** task is the collection of symbol resolution results
907 on all applicable tokens. More concretely, the output is a sequence of values of type
908 `Option<SymbolResolution>` where `Option<SymbolResolution>` is the type `SymbolResolution` with
909 a null value added for non-applicability and `SymbolResolution` is the type storing the result of the
910 symbol resolution, being either a success with a resolved symbol of type `Symbol` or a failure with
911 an error of type `SymbolResolutionError`. In Cybertron, we define `SymbolResolution` explicitly as
912 follows:

```
913 1 // an enum type definition, basically a tagged union type  
914 2 pub enum SymbolResolution {  
915 3     Ok(Symbol), // enum type variant for success with a resolved symbol  
916 4     Err(SymbolResolutionError), // enum type variant for failure with an error  
917 5 }
```

- 918 • The **type analysis** task's final output is the collection of all type errors. More concretely, the output
919 is a sequence of `Option<TypeError>`, where `Option<TypeError>` denoted the type `TypeError` will
920 a null value added and `TypeError` is the type storing the information of a type error. The position
921 of type errors agrees with the source tokens leading to these errors. In Cybertron, we define
922 `TypeError` explicitly as follows:

```
923 1 // This enum represents various kinds of type errors  
924 2 pub enum TypeError {  
925 3     /// This variant indicates a type mismatch  
926 4     /// 'expected' is the type that was anticipated  
927 5     /// 'actual' is the type that was encountered  
928 6     TypeMismatch { expected: Type, actual: Type },  
929 7 }
```

930 One can expand the definition to include other kinds of type errors.

931 (1) *Type definition*. Types are either identified uniquely by a single identifier like `<identifier>`, or
932 builtin generic types `Option<<identifier>>` or `Vec<<identifier>>`. Users can define custom types
933 without generics like the following (f32 means float32 and i32 means int32 below):

```
934 1 struct Dog { weight: f32 }
935 2
936 3 enum Animal {
937 4     Dog,
938 5     Cat,
939 6 }
```

940 This part is actually a part of the AST task and type definition is a variant of the `AstData` type:

```
941 1 /// Enumeration representing different types of Abstract Syntax Tree (AST) nodes
942 2 pub enum AstData {
943 3     ...
944 4     /// Represents a function or variable definition
945 5     ///
946 6     /// # defn
947 7     ///
948 8     Defn {
949 9         /// The keyword in the definition (e.g., 'fn', 'enum')
950 10        keyword: DefnKeyword,
951 11        /// Index of the identifier in the definition
952 12        ident_idx: Idx,
953 13        /// The identifier being defined (e.g., function name, variable name)
954 14        ident: Ident,
955 15        /// Index of the content or body of the definition
956 16        content: Idx,
957 17    },
958 18 }
```

959 (2) *Type specification*. Each appeared variable has a unique type, either by specification or specu-
960 lation. All parameters of a function must be specified explicitly by users. Variables defined by let
961 statements might or might not be specified, as follows:

```
962 1 fn f(a: i32) { // type of 'a' must be specified
963 2     let x: i32 = a; // type of 'x' specified
964 3     let y = a; // type of 'y' unspecified
965 4 }
```

966 The return type of functions must be specified. The field type of structs and enum variants must be
967 specified. the type of expressions of function calls and field access will be determined correspond-
968 ingly.

969 The output of the task is the collection of all type signatures, represented as a sequence of values of
970 type `Option<TypeSignature>` where `TypeSignature` is the type holding the essential information of
971 type specifications. In Cybertron, `TypeSignature` is defined as,

```
972 1 pub struct TypeSignature {
973 2     pub key: TypeSignatureKey,
974 3     pub ty: Type,
975 4 }
976 5
977 6 pub enum TypeSignatureKey {
978 7     FnParameter { fn_ident: Ident, rank: Rank },
979 8     FnOutput { fn_ident: Ident },
980 9     StructField { ty_ident: Ident, field_ident: Ident },
981 10 }
```

982 (3) *Type inference*. As discussed above, not all variables have their types specified.

```
983 1 fn f() {
984 2     let x: i32 = 1;
985 3     let y = x;
986 4     let z = y;
987 5 }
```

988 In the above code, 1 is an ambiguous literal that can be of type `i32`, `i64`, `u32`, `u64`, etc, and
989 the types of `y` and `z` is not specified. However, one easily sees that there exists one and only one

990 choice of the types of `x`, `y`, and `z` such that the whole code is type correct. Utilizing this property,
991 the user can opt out of a significant portion of type specification, achieving static guarantees.

992 **A Type Inference Algorithm:** For simplicity, we shall prove transformers can implement a simple
993 type inference algorithm: we maintain a table of type assignments for variables. We update the
994 entries of the table by means of reduction, i.e., assuming the whole code is correctly typed and infer
995 more and more unspecified types until we encounter errors or all types are inferred. The process is
996 largely parallel, and we call the number of rounds needed the depth of type inference.

997 In the above code, the first round, we determine that the type of both `x` and the type of `y` are equal
998 to the type of `z` which is `i32`. But we have no way to determine the type of `z` because the type
999 of `y` is unknown at the first round. In the second round, `z` can be determined to be of type `i32`
1000 because the type of `y` is already inferred.

1001 The output of the task is the collection all types inferred, represented as a sequence of values of type
1002 `Option<TypeInference>` where `TypeInference` is the type holding the inferred type. In Cybertron,
1003 `TypeSignature` is defined as,

```
1004 1 pub struct TypeInference {  
1005 2     pub ty: Type,  
1006 3 }
```

1007 E Cybertron

1008 E.1 Introduction

1009 It's often difficult to directly prove that transformers or in general other low level forms of computa-
1010 tion can express complicated algorithms and even complex software. There are way too many details
1011 as compared with typical mathematical proofs in machine learning theory. Hence, we propose the
1012 domain specific language Cybertron, where we can systematically prove transformers can express
1013 complicated algorithms and complex software with sufficient readability.

1014 (Note: Cybertron is fundamentally different from Mini-Husky! Mini-Husky is the target language
1015 that we want transformers to analyze yet Cybertron is the domain specific language we use to prove
1016 that transformers can do that.)

1017 RASP (Weiss et al., 2021) is quite close to Cybertron in terms of its design purpose. However,
1018 Cybertron is more powerful with advanced algebraic type system, global and local function con-
1019 structions, etc. Thus, using Cybertron one can argue more complicated operations can be simulated
1020 by transformers than simple algorithms.

1021 In the broader perspective of computer science, it's not uncommon to use code to prove things. In
1022 fact, in the formal verification community, mathematical proofs are viewed as a special case of a
1023 much larger universe of possible proof systems.

1024 Essentially, Cybertron works as follows:

- 1025 1. in Cybertron, one builds complicated functions from the composition of smaller functions.
1026 We have lemmas that prove that the composed functions are representable by certain archi-
1027 tecture given that smaller functions are representable.

1028 For example:

```
1029 1 fn f(a: f32, b: f32) -> f32 {  
1030 2     a + b  
1031 3 }
```

- 1032 2. in Cybertron, there is an algebraic type system and every value is strongly typed and im-
1033 mutable, making it highly readable and easy to reason about;

1034 For example:

```
1035 1 fn f(a: f32, b: f32) -> f32 {  
1036 2     a + b  
1037 3 }
```


1038 3. in Cybertron, there is a distinction between global and local types/functions. Local types
1039 are those information hovered over a single token, and global types are sequences of local
1040 types, i.e., the collection of information over the whole token stream. One can define a
1041 global function by mapping a local function.

1042 For example:

```
1043 1 fn f(a: f32, b: f32) -> f32 {  
1044 2     a + b  
1045 3 }
```

1046 4. in Cybertron, there are many functions that is defined externally, requiring external expla-
1047 nation that they can be represented by transformers.

1048 For example:

```
1049 1 fn f(a: f32, b: f32) -> f32 {  
1050 2     a + b  
1051 3 }
```

1052 It's implemented as a subset of the Rust programming language that can be understood as computa-
1053 tion graphs over sequences. It can be executed for testing purposes and we've tested our implemen-
1054 tation for a range of inputs and validated its correctness.

1055 E.2 Philosophy: Sequential Representation of Everything

1056 Before going through the full details, let's first talk about the fundamental philosophy behind trans-
1057 former and Cybertron.

1058 One of the fundamental reasons transformers can be easily adapted across multiple modalities, in-
1059 cluding NLP and CV, is their sequence-to-sequence operation. Everything can be represented as an
1060 arbitrary-length sequence. Texts are sequences of words, images are sequences of image patches,
1061 videos are sequences of spacetime patches (OpenAI, 2024b), and even graphs with sparse spatial
1062 structures can be represented as sequences of indexed nodes with additional information like parent
1063 node indices. Since inputs of various modalities can be cast into vector sequences, transformers can
1064 be applied to different domains without modifications to their architecture (Dosovitskiy et al., 2020).

1065 Interestingly, this sequence-based thinking is not new. **We've actually been representing every-**
1066 **thing as sequences since the very early days of computer science.** This has been the foundation
1067 of how data is stored and processed in computers. However, sequence representations were tradi-
1068 tionally viewed as low-level and sometimes inefficient for practical use, prompting the development
1069 of higher-level abstractions for programming. The rise of transformers, with their scalable learning
1070 capabilities, encourages us to reconsider the significance of sequence-based representations.

1071 From a systems perspective, viewing everything as a sequence is the foundational approach in com-
1072 puter science. Data in a computer is stored as a continuous stream of bits. Whether it's text, images,
1073 videos, or graphs, this data is represented in computer memory as an ordered sequence of bits. This
1074 aligns with how transformers handle different types of input by transforming them into sequences
1075 of vectors. Thus, the sequence-based operation of transformers mirrors the sequence-based repre-
1076 sentation of data in computer memory.

1077 In essence, **if a data structure can be represented in computer memory using N bits, it can**
1078 **be processed as a sequence of bits of length N .** This natural sequence representation in memory
1079 is consistent with how transformers process data, which makes them particularly flexible across
1080 different modalities. For example, recent state-of-the-art approaches Wu et al. (2024) show that
1081 transformers can even be trained directly on raw bits of data, further emphasizing this connection.

1082 Moreover, this sequence-based viewpoint offers fresh insights when applied to the domain of pro-
1083 gramming, particularly in areas such as code generation and analysis. With tools like ChatGPT and
1084 Copilot being widely used by developers, the impact of transformers on programming workflows
1085 is growing. Understanding the complexity of algorithms and programs expressed in sequence form
1086 becomes an interesting area of study, as it reveals new possibilities for how we approach computa-
1087 tion.

1088 In comparison to traditional systems like CPUs and human cognition, transformers are highly paral-
1089 lel but shallow in their operation. A transformer processes data in a fixed number of layers, while a

1090 CPU executes 10^9 cycles per second, and humans may take days to process information like reading
1091 a book. Transformers, therefore, represent a fundamentally different computational model that is
1092 worth studying further in the context of sequence-based operations.

1093 **Example 2. Image to Sequence:** *In computer memory, an image is typically stored as a continuous*
1094 *block of pixel values, often in row-major order, where each pixel’s value is encoded as bits in a*
1095 *sequence. When a transformer processes an image, it divides the image into patches (e.g., $16 \times$*
1096 *16 pixels), and each patch is flattened into a vector of pixel values. This creates a sequence of*
1097 *patches, where each patch corresponds to a vector. The way transformers represent these patches*
1098 *as a sequence closely aligns with how the image data is sequentially stored in computer memory.*

1099 **Example 3. Video to Sequence:** *A video is stored in computer memory as a sequence of frames,*
1100 *where each frame is essentially an image. In a similar manner to images, these frames are stored as*
1101 *continuous pixel values. Transformers process videos by dividing the frames into spacetime patches,*
1102 *where each patch captures a small region of space over a short segment of time. These spacetime*
1103 *patches are flattened and arranged into a sequence for the transformer to process. The sequential*
1104 *ordering of these patches matches how video frames and pixel data are stored in computer memory.*

1105 **Example 4. Graph to Sequence:** *In computer memory, a graph is typically stored using an adjac-*
1106 *ency list or adjacency matrix, where nodes and their connections (edges) are stored sequentially*
1107 *in a data structure. Transformers process graphs by representing each node and its features as a*
1108 *vector, and then creating a sequence of these vectors. The sequence may also encode additional*
1109 *information, such as the parent-child relationships between nodes. This sequence-based represen-*
1110 *tation of graphs is consistent with how graph data is stored in memory, where nodes and edges are*
1111 *arranged in a structured order.*

1112 **Example 5. Text to Sequence:** *Text is naturally stored in computer memory as a sequence of char-*
1113 *acters or words, where each character is encoded as a sequence of bits (such as ASCII or Unicode*
1114 *values). When transformers process text, they convert each word into a word embedding, which is a*
1115 *vector of real numbers. The sequence of word embeddings corresponds to the sequence of charac-*
1116 *ters or words stored in memory. This natural sequential representation of text in both memory and*
1117 *transformers ensures efficient handling of linguistic data.*

1118 **Example 6. AST (Abstract Syntax Tree) to Sequence:** *In computer memory, an abstract syntax*
1119 *tree (AST) is typically stored as a tree-like structure, where each node represents a component of*
1120 *the program (e.g., operators, variables, or statements). However, this tree can be linearized into a*
1121 *sequence by traversing it in a specific order (e.g., pre-order traversal). When transformers process*
1122 *an AST, they convert it into a sequence of tokens, where each token corresponds to a node in the tree.*
1123 *This sequential representation of the tree in transformers mirrors how the tree is stored as nodes and*
1124 *edges in memory, and how it can be flattened into a linear sequence.*

1125 In conclusion, the sequence-based representation in transformers is not just a novel approach for
1126 deep learning but is deeply rooted in how data has been stored and processed in computer memory
1127 since the early days of computing. This consistency between how data is stored in memory and how
1128 transformers process data as sequences is a key factor in their adaptability across different domains.

1129 E.3 Local and Global Types

1130 Now we define the type foundation of Cybertron.

1131 Types are fundamental objects for programming language theory. Here we use types to facilitate our
1132 proofs. Type signatures contain rich information that help guarantee correctness of the program.
1133 Here, we choose a mathematical definition of types that is most convenient for the discussion in
1134 this paper. We introduce the notion of “local type”. Roughly speaking, they are types without heap
1135 allocation and intended to be represented with $\mathbb{R}^{d_{\text{model}}}$ over a single token. For more complicated
1136 heap-allocated data structures like trees, graphs, etc., we shall represent them by sequences of these
1137 “local type”s, which translates directly to vector sequences for transformers.

1138 **Definition 12 (Local Type).** *Given a base space B with at least two elements and a countably*
1139 *infinite identification space Ψ , a local type \mathcal{T} over B is a finite set S together with an embedding ϕ*
1140 *from S to B^d and some fixed $d \in \mathbb{N}$ and an identification $\psi \in \Psi$.*

1141 *For convenience, define $\text{Set}(\mathcal{T}) = S$, $d_{\mathcal{T}} = d$ and $\phi_{\mathcal{T}} = \phi$ and $\psi_{\mathcal{T}} = \psi$. And let 0_B , and 1_B be two*
1142 *different elements of B . And $B^0 := \{0_B\}$ so that $|B^i| = |B|^i$ holds for all $i \in \mathbb{N}$.*

1143 *Remark 2.* We need B to be at least size 2, so that B^d can be as large as we want for d large
 1144 enough. For typical computer representation, we can take B to be $\mathbf{2} = \{0, 1\}$. For transformers
 1145 or neural networks in general, we can take B to be \mathbb{R} if we ignore precision. If we don't ignore
 1146 precision, B should be some finite set of floating point numbers. Thus, we shall keep the generality
 1147 of B throughout our discussion as all of these settings are important.

1148 *Remark 3.* The role of identification $\psi_{\mathcal{T}} \in \Psi$ is to make two types mathematically different even if
 1149 they have the same underlying set, encoding dimension, and encoding. Basically we are establishing
 1150 a specialized type of theory tailored towards the expressive power of transformers upon a foundation
 1151 of intuitive set theory.

1152 **Example 7 (Finite Set).** *In mathematics, we have the finite set denoted by $[n] = \{0, 1, \dots, n-1\}$.
 1153 Here we use a slightly different notation for a type with underlying set $\llbracket n \rrbracket$ and some encoding.*

1154 **Example 8 (Position Encoding).** *Position encoding can be viewed as the encoding of a type denoted
 1155 by $\text{Pos}(n)$ with the underlying set being $[n]$ where n is the context length. Although it has the same
 1156 underlying set as type $\llbracket n \rrbracket$, it is a different type for a different purpose and might have different
 1157 encoding.*

1158 *If B is \mathbb{R} , then the position encoding can be understood as the encoding of type $\llbracket L \rrbracket$ where L is the
 1159 context length. More explicitly, we have*

$$\phi(x) = (e^{iL^{-i/d}x})_{i \in [d/2]}, \quad (7)$$

1160 *viewed as a d dimensional \mathbb{R} -vector through the natural conversion of \mathbb{C} to \mathbb{R}^2 , since d is even.*

1161 It's too cumbersome to manually give the underlying set and the encoding. Here we introduce a
 1162 classical concept from programming language theory [Program \(2013\)](#) that makes it super easy to
 1163 construct new types and make things fairly readable.

1164 **Definition 13 (Finite Algebraic Data Type, Mathematical Forms).** *We define two ways of creating
 1165 new types by combining existing types:*

1166 1. *Sum type. Given types $\mathcal{T}_i = (S_i, \phi_i, d_i)$ over base space B for $i = 1, \dots, n$, we define the
 1167 sum type of \mathcal{T}_i , denoted by $\sum_{i=1}^n \mathcal{T}_i$, as follows,*

- 1168 • let $S = (\{1\} \times S_1) \sqcup \dots \sqcup (\{n\} \times S_n)$;
- 1169 • let $d = d_{\llbracket n \rrbracket} + \max_{i=1}^n d_i$;
- 1170 • let $\phi : S \rightarrow B^d$ be such that

$$\forall i \in \llbracket n \rrbracket, s \in S_i, \phi((i, s)) = \phi_{\llbracket n \rrbracket}(i) \oplus \phi_i(s) \in B^{d_{\llbracket n \rrbracket} + d_i} \subseteq B^d. \quad (8)$$

1171 *Note that $|S| = \sum_{i=1}^n |S_i|$, thus the name sum type.*

1172 2. *Product type. Given Local Types $\mathcal{T}_i = (S_i, \phi_i, d_i)$ over base space B for $i = 1, \dots, n$, we
 1173 define the product type of \mathcal{T}_i , denoted by $\prod_{i=1}^n \mathcal{T}_i$, as follows,*

- 1174 • let $S = S_1 \times \dots \times S_n$;
- 1175 • let $d = \sum_{i=1}^n d_i$;
- 1176 • let $\phi : S \rightarrow B^d$ be such that

$$\forall s = (s_1, \dots, s_n) \in S, \phi(s) = \phi_1(s_1) \oplus \dots \oplus \phi_n(s_n) \in B^d. \quad (9)$$

1177 *Note that $|S| = \prod_{i=1}^n |S_i|$, thus the name product type.*

1178 Although we can define things and refer to things in terms of mathematical equations, it's sometimes
 1179 cumbersome to do so. So we shall frequently refer to types using a programming language form,
 1180 like `CybertronForm` or more complicated things like `Option<T>` a builtin generic type.

1181 **Definition 14 (Unit Type).** *The unit type is a type with $S = \{0\}$ and $\phi : S \rightarrow B^0, 0 \mapsto 0_B$. In
 1182 `Cybertron`, it's denoted as `()`.*

1183 **Definition 15 (Array Type).** *Given a type \mathcal{T} , the array type of \mathcal{T} with length $\ell \in \mathbb{N}$ is the type with
 1184 $S = S(\mathcal{T})^\ell$, $d = \ell d_{\mathcal{T}}$ and $\phi : S \rightarrow B^{\ell d_{\mathcal{T}}}, (s_1, \dots, s_\ell) \mapsto \phi_{\mathcal{T}}(s_1) \oplus \dots \oplus \phi_{\mathcal{T}}(s_\ell)$. It's denoted by
 1185 \mathcal{T}^ℓ . In `Cybertron`, it's denoted as `[T;N]`.*

1186 **Definition 16** (Vector Type of Finite Capacity). *Given a type \mathcal{T} , the vector type of finite capacity of*
 1187 *\mathcal{T} with maximal length $\ell \in \mathbb{N}$ is the type with $S = \bigsqcup_{i=1}^{\ell} \text{Set}(\mathcal{T})^i$, $d = d_{[\ell]} + \ell d_{\mathcal{T}}$ and $\phi : S \rightarrow$*
 1188 *$B^{d_{[\ell]} + \ell d_{\mathcal{T}}}$, $(s_1, \dots, s_i) \mapsto \phi_{[\ell]}(i) \oplus \phi_{\mathcal{T}}(s_1) \oplus \dots \oplus \phi_{\mathcal{T}}(s_i) \oplus 0_B \oplus \dots \oplus 0_B$ with just enough*
 1189 *number of copies of 0_B such that the dimensionality matches. It's denoted by $\mathcal{T}^{\leq \ell}$. In cybertron, it's*
 1190 *denoted as `BoundedVec<T,N>`.*

1191 However, it's cumbersome and obtuse to define and operate in mathematical forms only. So we shall
 1192 give a definition closer to actual programming that is more convenient and easy to read.

1193 **Definition 17** (Finite Algebraic Data Type, the Code Forms). *We define two ways to create new*
 1194 *types:*

1195 1. *Enum type. An enum type is the sum type of a finite set of variant types. Each variant type*
 1196 *is associated with a different identifier and can be*

- 1197 • *unit like, a unit type;*
- 1198 • *struct like, a product of several types, each called a field of the variant, and associated*
 1199 *with an identifier;*
- 1200 • *tuple like, a product of several types, each called a field of the variant, but not associ-*
 1201 *ated with an identifier.*

1202 *Syntactically, an enum type is specified as follows,*

```
1203 1 enum <type-name> {
1204 2   <identifier> { // 1st variant, struct like
1205 3     <identifier>: <type>, // 1st named field of 1st variant
1206 4     <identifier>: <type>, // 2nd named field of 1st variant
1207 5     ...
1208 6   },
1209 7   <identifier> { // 2nd variant, struct like
1210 8     <identifier>: <type>, // 1st field of 2nd variant
1211 9     ...
1212 10  },
1213 11  <identifier> ( // 3rd variant, tuple like
1214 12    <type>, // 1st tuple field of 3rd variant
1215 13    <type>, // 2nd tuple field of 3rd variant
1216 14    ...
1217 15  ),
1218 16  <identifier>, // 4th variant, unit like
1219 17  ...
1220 18 }
```

1221 *For example,*

```
1222 1 enum Expr {
1223 2   Variable(IdentToken), // 1st variant, tuple like
1224 3   Binary { // 2nd variant, struct like
1225 4     lopd: ExprId,
1226 5     opr: BinaryOprToken,
1227 6     ropd: ExprId,
1228 7   },
1229 8   Prefix { // 3rd variant, struct like
1230 9     opr: PrefixOprToken,
1231 10    opd: ExprId,
1232 11  },
1233 12  Suffix { // 4th variant, struct like
1234 13    opd: ExprId,
1235 14    opr: SuffixOprToken,
1236 15  },
1237 16  Panic, // 5th variant, unit like
1238 17 }
```

1239 2. *Struct type. A struct type is just the product type of*

```
1240 1 struct <type-name> {
1241 2   <identifier>: <type>,
1242 3   <identifier>: <type>,
1243 4   ...
1244 5 }
```

```
1245 1 struct A {
1246 2   a: i32
1247 3 }
```

1248 To show how convenient this is, we can define the very useful option type as follows,

1249 **Definition 18** (Option type). *For a local type T , we can define the local as*

```
1250 1  enum Option<T> {  
1251 2     Some(T),  
1252 3     None  
1253 4 }
```

1254 **Definition 19** (Global Types). *Global types are defined to be sequences of local types.*

1255 **Example 9** (Representation of Graphs). *Graphs can be represented as sequences of its nodes. We*
1256 *can use position index to use as node references.*

1257 E.4 Computation Graph

1258 For convenience, we shall use computation graph as a vehicle to describe complicated computa-
1259 tion processes. Computation graph is close to actual computation process and one can derive an
1260 understanding of the computation difficulty from the graph's mathematic properties (width, depth,
1261 etc.)

1262 **Definition 20** (Directed Simple Graph). *A directed simple graph G is a pair (V, E) where V is a*
1263 *finite set, and $E \subseteq V \times V$ is called edges.*

1264 In the following, we shall simplify the "directed simple graph" to just graph.

1265 **Definition 21** (Computation Graph). *A computation graph is an acyclic directed graph $G = (V, E)$*
1266 *with additional structures:*

- 1267 1. *for each vertex $v \in V$, there is an associated type, denoted by T_v ;*
- 1268 2. *for each vertex $v \in V$ with a positive number of incoming edges, let v_1, \dots, v_n be the other*
1269 *vertices for the incoming edges, then there is an associated function f_v from $T_{v_1} \times \dots \times T_{v_n}$*
1270 *to T_v .*

1271 A computation graph naturally generates a function from **source vertices** to **sink vertices**. Let
1272 $v_1^{\text{in}}, \dots, v_n^{\text{in}}$ be the set of vertices with no incoming edges, and let $v_1^{\text{out}}, \dots, v_m^{\text{out}}$ be the set of vertices
1273 with no outgoing edges. Then we can construct a function from $T_{v_1^{\text{in}}} \times \dots \times T_{v_n^{\text{in}}}$ to $T_{v_1^{\text{out}}} \times \dots \times T_{v_m^{\text{out}}}$
1274 in the following obvious manner:

- 1275 1. let $(x_1, \dots, x_n) \in T_{v_1^{\text{in}}} \times \dots \times T_{v_n^{\text{in}}}$ be an input;
- 1276 2. for each v_i^{in} , assign it with value x_i ;
- 1277 3. for each vertex $v \in V$ with all its incoming vertices v_1, \dots, v_l assigned with a value, assign
1278 it with the value $f_v(x_{v_1}, \dots, x_{v_l})$ where x_{v_i} denotes the value assigned to v_i ;
- 1279 4. repeat the process until all vertices are assigned a value, then take $(x_{v_1^{\text{out}}}, \dots, x_{v_m^{\text{out}}})$ as the
1280 output.

1281 Our goal is to make a hypothesis class using the above graph. To control the statistical and compu-
1282 tational complexity, we put restrictions on the choice of T_v and f_v , as follows:

1283 **Definition 22** (Restricted Computation Graph). *Let \mathcal{U} be a set of types, and for any $A, B \in \mathcal{U}$, there*
1284 *is a set of functions $\text{Mor}(A, B)$ from A to B . We require that $T_v, T_v^{\text{in}} \in \mathcal{U}$ and $f_v \in \text{Mor}(T_v^{\text{in}}, T_v)$*
1285 *where $T_v^{\text{in}} := \prod_{v'v \in E} T_{v'}$. We also require that the underlying graph G satisfies certain conditions*
1286 *(width, depth, etc.)*

1287 **Definition 23** (Restricted Computation Graph Of Sequences). *Let \mathcal{U} be a universe such that for a set*
1288 *of types \mathcal{U}_0 all types in \mathcal{U} are of the form A^* for some type $A \in \mathcal{U}_0$, and $\text{Mor}(A^*, B^*)$ are functions*
1289 *that preserve sequence lengths.*

1290 Given a restriction, the class of functions generated by restricted computation graphs is the central
1291 object to study in this paper. We shall use an even more restricted computation graph of sequences.
1292 We shall argue about the class of functions formed that

- 1293 1. it's rich enough to contain many interesting operations including SQL, compiler (type in-
1294 ference, static analysis)

- 1295 2. it’s computationally reasonable, and can be represented by transformers with pragmatic
1296 bounds
1297 3. it has a reasonable statistical complexity

1298 As a corollary, our theories suggest that transformers can possibly learn to do many interesting things
1299 with reasonable computational and statistical complexity.

1300 To our knowledge, this is the first theoretical paper that gives pragmatic optimistic bounds for the
1301 power of transformers in a wide range of meaningful language tasks.

1302 Now we introduce graph-theoretical measures that will play key roles in our new complexity theory.

1303 The most basic one is the following:

1304 **Definition 24** (Depth of Graph). *The depth of a computation graph is defined to the length of the*
1305 *longest path, denoted by $\text{Depth}(G)$.*

1306 For convenience, we define the following vertex-wise depth.

1307 **Definition 25** (Depth of Graph Vertex). *The depth of a vertex v of a computation graph is defined*
1308 *as the length of the longest path with end v , denoted by $\text{Depth}(v)$.*

1309 The smaller d_G is, the more parallel the computation is.

1310 However, we shall discuss a more nuanced measure, containment, as follows:

1311 E.5 Functions over Local Types

1312 **Definition 26** (Functions over Local Types). *Given Local Types \mathcal{T}, \mathcal{R} , the functions from \mathcal{T} to \mathcal{R}*
1313 *are defined to be just the functions from $\text{Set}(\mathcal{T})$ to $\text{Set}(\mathcal{R})$.*

1314 *Remark 4.* The domains and codomains are all finite sets, so there aren’t many constraints we want
1315 to enforce here. Basically, these are “discrete” functions.

1316 **Definition 27** (Functions over Algebraic Data Types). *Let $\mathcal{T}, \mathcal{S}_1, \dots, \mathcal{S}_m, \mathcal{R}$ be Local Types, and*
1317 *suppose that \mathcal{T} is an algebraic data type, then we can construct functions from $\mathcal{T} \times \mathcal{S}_1 \times \dots \times \mathcal{S}_m$*
1318 *to \mathcal{R} as follows,*

- 1319 1. *suppose that \mathcal{T} is the sum type of $\mathcal{T}_1, \dots, \mathcal{T}_n$. Then given functions $f_i : \mathcal{T}_i \times \mathcal{S}_1 \times \dots \times \mathcal{S}_m$*
1320 *for $i = 1, \dots, n$, we can construct a function f , by letting*

$$f((i, t), s_1, \dots, s_m) = f_i(t, s_1, \dots, s_m), \quad (10)$$

1321 *for each $t \in \text{Set}(\mathcal{T}_i), s_1 \in \text{Set}(\mathcal{S}_1), \dots, s_m \in \text{Set}(\mathcal{S}_m)$.*

1322 *(Note that we use the pair (i, t) because the underlying set of \mathcal{T} is $\bigsqcup_{i=1}^n \{i\} \times \text{Set}(\mathcal{T}_i)$.)*

- 1323 2. *suppose that \mathcal{T} is the product type of $\mathcal{T}_1, \dots, \mathcal{T}_n$. Then given a function $f_* : \mathcal{T}_1 \times \dots \times$*
1324 *$\mathcal{T}_n \times \mathcal{S}_1 \times \dots \times \mathcal{S}_m$ for $i = 1, \dots, n$, we can construct a function f , by letting*

$$f((t_1, \dots, t_n), s_1, \dots, s_m) = f_*(t_1, \dots, t_n, s_1, \dots, s_m), \quad (11)$$

1325 *for each $t \in \text{Set}(\mathcal{T}_i), s_1 \in \text{Set}(\mathcal{S}_1), \dots, s_m \in \text{Set}(\mathcal{S}_m)$.*

1326 It is not enough to just mathematically construct. We should also discuss how neural networks can
1327 represent these functions. We define the representation of functions over Local Types formally as
1328 follows:

1329 **Definition 28** (Representation of Functions over Local Types Using Multi-Layer Perceptions). *Let*
1330 *\mathcal{T}, \mathcal{R} be Local Types. Given a function f from \mathcal{T} to \mathcal{R} , we say it is representable by MLP of*
1331 *dimension $d \geq \max\{d_{\mathcal{T}}, d_{\mathcal{R}}\}$ and number of layers L , if there exists $\hat{f} \in \text{ResMlp}_L^d$ such that*

$$\iota_1 \circ \phi_{\mathcal{R}} \circ f = \tilde{f} \circ \iota_2 \circ \phi_{\mathcal{T}}, \quad (12)$$

1332 *where $\iota_1 : \mathbb{R}^{d_{\mathcal{R}}} \rightarrow \mathbb{R}^d$ and $\iota_2 : \mathbb{R}^{d_{\mathcal{T}}} \rightarrow \mathbb{R}^d$ are the canonical embeddings by putting zeros to fit the*
1333 *dimensionalities.*

1334 Here are some trivially true facts:

1335 **Proposition 3.** [Identities are Representable] For any Local Type \mathcal{T} , the identity map $\text{Id}_{\mathcal{T}}$ is repre-
 1336 sentable in $\text{ResMlp}_1^{d_{\mathcal{T}}}$.

1337 *Proof.* Just take $W_0^{(1)} = I_d, W_1^{(1)} = W_2^{(2)} = 0, B_1^{(1)} = B_2^{(2)} = 0$. □

1338 **Proposition 4.** [Equality is Representable] The equality function for any local type \mathcal{T} is repre-
 1339 sentable in ResMlp_2^{2d} , where d is the encoding dimension of \mathcal{T} .

1340 *Proof.* Let $x, y \in \mathcal{T}$ be the inputs. We encode them as $\phi_{\mathcal{T}}(x), \phi_{\mathcal{T}}(y) \in \mathbb{R}^d$. The equality function
 1341 can be represented as:

$$f_{\text{eq}}(x, y) = \min \left(1, A \sum_{i=1}^d |\phi_{\mathcal{T}}(x)_i - \phi_{\mathcal{T}}(y)_i| \right),$$

1342 where A is a large enough positive constant such that the RHS is either 1 or 0.

1343 This can be implemented in two-layer ResMLP with dimension $2d$. □

1344 **Proposition 5.** [Boolean NOT is Representable] The Boolean NOT function is representable in
 1345 ResMlp_1^1 .

1346 *Proof.* It's affine. □

1347 **Proposition 6.** [Boolean AND is Representable] The Boolean AND function is representable in
 1348 ResMlp_1^2 .

1349 *Proof.* Represent each Boolean value as a binary flag within a 1-dimensional vector. Then AND is
 1350 just taking the minimum. By $\min(a, b) = b - \sigma_{\text{ReLU}}(b - a)$, we're done. □

1351 **Proposition 7.** [Boolean OR is Representable] The Boolean OR function is representable in
 1352 ResMlp_1^2 .

1353 *Proof.* Represent each Boolean value as a binary flag within a 1-dimensional vector. Then OR is
 1354 just taking the maximum. By $\max(a, b) = a + \sigma_{\text{ReLU}}(b - a)$, we're done. □

1355 **Proposition 8.** [THEN_SOME is Representable] The function `Bool::then_some : Bool × T →`
 1356 `Option T` returns `Some t` if the boolean is true and `None` otherwise. This function is representable
 1357 in ResMlp_1^{d+1} .

1358 *Proof.* Encode the boolean as a binary flag in a $(d + 1)$ -dimensional vector, where the first compo-
 1359 nent indicates the boolean value and the remaining d components hold the value of type `T`. The
 1360 residual MLP f_{resmlp} constructs the output `Option T` by assembling the flag and the value split into
 1361 positive and negative parts influenced by the flag:

$$f_{\text{resmlp}}(X) = \left(\sigma_{\text{ReLU}}(x_{2:d+1} - Ax_1) \begin{matrix} x_1 \\ \end{matrix} - \sigma_{\text{ReLU}}(-x_{2:d+1} - Ax_1) \right).$$

1362 Here, A is a vector of dimension d with all entries positive and large enough to ensure proper
 1363 thresholding. Specifically, each entry of A should be larger than the maximum absolute value that
 1364 can be represented in the corresponding dimension of type `T`. This ensures that when $x_1 = 1$, the
 1365 subtraction $x_{2:d+1} - A$ will always be negative, and when $x_1 = 0$, it will not affect the value.

1366 When the flag is true ($x_1 = 1$), $\sigma_{\text{ReLU}}(x_{2:d+1} - A) = 0$ and $\sigma_{\text{ReLU}}(-x_{2:d+1} - A)$ retains the
 1367 negated value, resulting in `Some t`. When the flag is false ($x_1 = 0$), both ReLU terms preserve the
 1368 value, yielding `None`. Thus, f_{resmlp} effectively implements `Bool::then_some` within a single layer
 1369 of the MLP. □

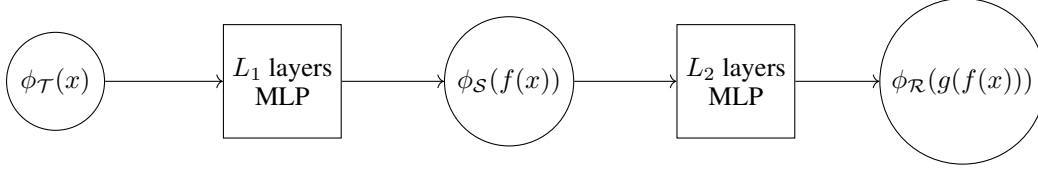


Figure 3: Transformation from $\phi_{\mathcal{T}}(x)$ to $\phi_{\mathcal{S}}(f(x))$ to $\phi_{\mathcal{R}}(g(f(x)))$ with MLP layers.

1370 **Proposition 9.** [Option Or is Representable] Let T be a local type, let `Option::or` be the function
 1371 that maps two values a, b of type `Option T` to a value c of type `Option T` such that c is equal to a
 1372 when a is not none, and equal to b otherwise. Then `Option::or` is representable in $\text{ResMlp}_1^{2(d+1)}$.

1373 *Proof.* Each `Option T` is represented as a $(d + 1)$ -dimensional vector, where the first component is
 1374 a binary flag indicating the presence (1 for `Some`, 0 for `None`), and the remaining d components
 1375 encode the value. Given inputs $a, b \in \text{Option T}$, the residual MLP f_{resmlp} processes the concatenated
 1376 vector

$$X = \begin{pmatrix} a_{\text{flag}} \\ a_{\text{val}} \\ b_{\text{flag}} \\ b_{\text{val}} \end{pmatrix}.$$

1377 The MLP is designed to separate b_{val} into positive and negative parts (b_+, b_- respectively) influenced
 1378 by a_{flag} . Specifically, it computes:

$$\begin{aligned} f_{\text{resmlp}}(X) &= a_{\text{val}} + \sigma_{\text{ReLU}}(b_+ - Aa_{\text{flag}}) - \sigma_{\text{ReLU}}(b_- - Aa_{\text{flag}}) \\ &= a_{\text{val}} + \sigma_{\text{ReLU}}(b_{\text{val}} - Aa_{\text{flag}}) - \sigma_{\text{ReLU}}(-b_{\text{val}} - Aa_{\text{flag}}), \end{aligned} \quad (13)$$

1379 where A is a vector with large positive entries that ensures the ReLU activation zeroes out the non-
 1380 selected parts based on the flag. When $a_{\text{flag}} = 1$, the terms involving b are suppressed, resulting in
 1381 $c = a$. Conversely, when $a_{\text{flag}} = 0$, the positive part of b remains, effectively selecting b . Thus,
 1382 f_{resmlp} accurately implements the `Option::or` function, demonstrating that it is representable within
 1383 $\text{ResMlp}_1^{2(d+1)}$. □

1384 **Proposition 10** (Field Access Is Representable in ResMlp). For algebraic data type, either struct
 1385 field access, enum discriminator, and variant field access can be represented in ResMlp_1^d where d is
 1386 the encoding dimension.

1387 *Proof.* Obvious because these operations are linear. □

1388 **Proposition 11.** [Composition of Functions Representable in ResMlp] For local types $\mathcal{T}, \mathcal{S}, \mathcal{R}$, with
 1389 maps $f : \mathcal{T} \rightarrow \mathcal{S}$ and map $g : \mathcal{S} \rightarrow \mathcal{R}$ representable in $\text{ResMlp}_{L_1}^{d_1}$ and $\text{ResMlp}_{L_2}^{d_2}$ respectively.
 1390 Then $g \circ f$ is representable in $\text{ResMlp}_{L_1+L_2}^{\max\{d_1, d_2\}}$.

1391 *Proof.* Obvious by using the first L_1 layers to map from \mathcal{T} to \mathcal{S} and using the rest L_2 layers to map
 1392 from \mathcal{S} to \mathcal{R} . The process can be visualized as in Figure 3. □

1394 **Proposition 12.** [Computation Graph of Functions Representable in ResMlp] Let \mathcal{G} be a com-
 1395 putation graph, with each vertex v being of some local type \mathcal{T}_v , and the construction functions are
 1396 representable in $\text{ResMlp}_{L_v}^{d_v}$. For convenience, if v is a source vertex, d_v is defined to be the encoding
 1397 dimension of \mathcal{T}_v and $L_v = 0$. Then the function induced by the computation graph is representable
 1398 in $\text{ResMlp}_{\text{Depth}(\mathcal{G})(\max_{v \in \mathcal{G}} L_v + 1) + 1}^{\sum_{v \in \mathcal{G}} d_v}$.

1399 *Proof.* We construct a global residual multi-layer perceptron (ResMLP) that simulates the com-
 1400 putation graph \mathcal{G} by aggregating and updating the states of all vertices simultaneously. Let

1401 $D = \sum_{v \in \mathcal{G}} d_v$ be the total dimension, where d_v is the dimension associated with vertex v . The
 1402 global ResMLP will have a depth of $\text{Depth}(\mathcal{G})(\max_v L_v + 1)$.

1403 Consider the concatenated state vector $X^{(t)} \in \mathbb{R}^D$, which is a concatenation of the states of all
 1404 vertices:

$$X^{(t)} = \left(X_v^{(t)} \right)_{v \in \mathcal{G}},$$

1405 where $X_v^{(t)} \in \mathbb{R}^{d_v}$ is the state of vertex v at layer t .

1406 Initialization occurs at depth zero, corresponding to the source vertices of the computation graph.
 1407 The state vector $X^{(0)}$ is set by assigning the input vectors to the source vertices and initializing all
 1408 other vertices to zero. Formally, if V_0 denotes the set of source vertices, then:

$$X_v^{(0)} = \begin{cases} x_v & \text{if } v \in V_0, \\ 0 & \text{otherwise,} \end{cases}$$

1409 where $x_v \in \mathbb{R}^{d_v}$ is the input to source vertex v . Because $X_v^{(0)}$ is of dimensionality d_v equal to the
 1410 encoding dimension, this agrees with our convention for representing functions over local types.

1411 We proceed inductively over the depth levels of the computation graph. For each depth level $k =$
 1412 $1, 2, \dots, \text{Depth}(\mathcal{G})$, we perform the following steps in the global ResMLP.

1413 1. **Input Aggregation Layer.** We apply a linear transformation to gather the outputs from the
 1414 predecessor vertices of each vertex at depth k and feed them as inputs to these vertices.
 1415 Specifically, we define a linear mapping $W_{\text{agg}}^{(k)} \in \mathbb{R}^{D \times D}$ such that:

$$\tilde{X}^{(t_k)} = W_{\text{agg}}^{(k)} X^{(t_{k-1})},$$

1416 where t_{k-1} is the layer after processing depth $k - 1$, and $\tilde{X}^{(t_k)}$ is the aggregated input
 1417 for the vertices at depth k . The matrix $W_{\text{agg}}^{(k)}$ rearranges and combines the outputs from
 1418 predecessor vertices to provide the correct inputs to each vertex at depth k . Specifically,
 1419 for each vertex v at depth k , and for each predecessor u of v in the computation graph, the
 1420 matrix $W_{\text{agg}}^{(k)}$ contains entries that copy the output of u into the input positions of v . All
 1421 other entries in $W_{\text{agg}}^{(k)}$ are set to zero or identity as appropriate.

1422 2. **Local Computation Layers.** For each vertex v at depth k , we simulate its local ResMLP of
 1423 depth L_v . Since the depths L_v may vary, we pad the local ResMLPs to have a uniform
 1424 depth $L = \max_v L_v$ by adding identity mappings where necessary. The updates for vertex
 1425 v are computed as:

$$\begin{aligned} X_v^{(t_k+1)} &= \tilde{X}_v^{(t_k)} + f_{\text{fcn}_v} \left(\tilde{X}_v^{(t_k)} \right), \\ X_v^{(t_k+k')} &= X_v^{(t_k+k'-1)} + f_{\text{fcn}_v} \left(X_v^{(t_k+k'-1)} \right), \quad \text{for } k' = 2, \dots, L_v, \\ X_v^{(t_k+k')} &= X_v^{(t_k+k'-1)}, \quad \text{for } k' = L_v + 1, \dots, L. \end{aligned}$$

1426 Here, f_{fcn_v} denotes the single-layer fully connected network (as per Definition 1) for vertex
 1427 v .

1428 3. **State Update.** After completing the local computations for depth k , we update the global
 1429 state vector $X^{(t_k+L)}$ by concatenating the updated states of all vertices:

$$X^{(t_k+L)} = \left(X_v^{(t_k+L)} \right)_{v \in \mathcal{G}}.$$

1430 The total number of layers added for depth k is $L + 1$, accounting for the input aggregation layer
 1431 and the L layers simulating the local ResMLPs.

1432 By repeating this process for each depth level $k = 1, 2, \dots, \text{Depth}(\mathcal{G})$, we simulate the entire
 1433 computation graph within a global ResMLP of depth $\text{Depth}(\mathcal{G})(\max_v L_v + 1)$.

1434 Lastly, we use the final layer to perform a linear mapping so that the output is in the correct linear
 1435 representation, clearing out the intermediate values.

1436 Therefore, the function computed by the global ResMLP is equivalent to the function induced by
 1437 the computation graph \mathcal{G} , and it is representable in $\text{ResMlp}_{\text{Depth}(\mathcal{G})(\max_v L_v + 1)}^D$. \square

1438 *Remark 5.* We only prove things around MLPs here. Later, we shall show that this will imply that
 1439 the induced map operation over sequences can be represented by transformers.

1440 E.6 Functions over Global Types

1441 The task we want transformers to express is too complicated to be cleanly described in one shot. So
 1442 we introduce the following lemma to significantly simplify things. The lemma shall be useful for
 1443 our future papers on this topic.

1444 **Proposition 13.** *[Composition of Functions Representable in Transformers] For local types $\mathcal{T}, \mathcal{S},$
 1445 \mathcal{R} , with maps $f : \mathcal{T}^* \rightarrow \mathcal{S}^*$ and $g : \mathcal{S}^* \rightarrow \mathcal{R}^*$ representable in $\text{Tf}_{H_1, L_1}^{d_1}$ and $\text{Tf}_{H_2, L_2}^{d_2}$ respectively.*

1446 *Then the composition $g \circ f$ is representable in $\text{Tf}_{\max\{H_1, H_2\}, L_1 + L_2}^{\max\{d_1, d_2\}}$.*

1447 *Proof.* This is basically the same as the proof of Proposition 11. \square

1448 **Proposition 14.** *[Computation Graphs of Functions Representable in Transformers] Suppose we
 1449 have a computation graph $G = (V, E)$ with types $\mathcal{T}_v = T_v^*$ together with encoding map $\psi_v : T_v \rightarrow$
 1450 \mathbb{R}^{d_v} and decoding map $\phi_v : \mathbb{R}^{d_v} \rightarrow T_v$, satisfying $\phi_v \circ \psi_v \equiv \text{id}_{T_v}$, and there exists some positive
 1451 integer d_0 such that for each $v \in V$, f_v can be represented in*

$$\text{Tf}_{H_v, L_v}^d$$

1452 *Let f be the function generated by the computation graph. Then f can be represented in $\text{Tf}_{H, L}^d$ if
 1453 $d \geq \sum_v d_v + Hd_0$, $L \geq \frac{|G|}{H} + d_G$ where d_G is the depth of the graph.*

1454 *Remark 6.* This doesn't really cover the above. The bound in Proposition 14 isn't always tight for
 1455 model dimension when the computation graph is deep and Proposition 13 complements it.

1456 *Proof.* WLOG, assume that $d = \sum_{v \in V} d_v + Hd_0$. Then

$$\mathbb{R}^d = \underbrace{\left(\bigoplus_{v \in V} \mathbb{R}^{d_v} \right)}_C \oplus \underbrace{\left(\bigoplus_{h \in [H]} \mathbb{R}^{d_0} \right)}_A. \quad (14)$$

1457 Here C stands for "cache" used for storing computed values, and A stands for "active" used for
 1458 storing intermediate computation results.

1459 Make an order of all the nodes in the graph, say $V = \{v_1, \dots, v_{|G|}\}$ such that $\text{Depth}(v_i) \leq$
 1460 $\text{Depth}(v_j)$ if $i \leq j$.

1461 We now imagine the transformer computation process as gradually evaluating the value of each
 1462 vertex, starting from v_1 to $v_{|G|}$. Every $\max_v L_v$ layers form a layer group, and after each layer
 1463 group, at most H vertices are assigned values. The equation 14 implies that we have enough memory
 1464 to cache the computed values and intermediate values in small transformers.

1465 Now let this process continue until we compute all the values. It must be finite because after each
 1466 layer group, at least one of the vertices is computed. But this bound is too loose. We claim the
 1467 following:

1468 **Claim:** the number of layer groups where less than H vertices are assigned values is smaller than
 1469 $\text{Depth}(G)$.

1470 **Sketch of Proof of Claim:** for any layer group where less than H vertices are assigned, all the
 1471 vertices that aren't assigned after this layer group must have larger depth than any vertices that are
 1472 assigned values before this layer group, otherwise such a vertex can be evaluated in this layer group.
 1473 Define the depth of any layer group to be the smallest depth of vertices evaluated in this layer group.
 1474 Then for any unsatiated layer group, it must have a larger depth than the previous layer group. But
 1475 depth can only increase $\text{Depth}(G)$ times, thus there are at most $\text{Depth}(G)$ unsatiated layer groups.

1476 **Proof of Claim:** let V_1, \dots, V_l be the vertices evaluated at each layer group. Note that l is a different
 1477 symbol than L and means that the number of layer groups rather than the number of layers.

1478 For convenience, let D_i be the minimum of the depths of vertices in V_i .

1479 Suppose that the i th layer group is unsatiated, then $i < l$. We want to show that $D_i < D_{i+1}$.
 1480 Suppose otherwise, i.e., $D_i = D_{i+1}$. Because the i th layer group is unsatiated, for any $v \in V_{i+1}$,
 1481 v must have dependencies that haven't been evaluated before the i th layer group. Choose $v_0 \in$
 1482 $V_i, v_1 \in V_{i+1}$ such that $\text{Depth}(v_0) = \text{Depth}(v_1) = D_i = D_{i+1}$. Note that any dependency of v_1
 1483 must have smaller depths than v_0 , then must have already be evaluated before the i th layer group.
 1484 Contradiction!

1485 Now given the claim, we have that for all but at most $\text{Depth}(G)$ choices of $i = 1, \dots, l$, we have
 1486 $|V_i| = H$, then we have

$$|G| = \sum_{i=1}^l |V_i| \geq (l - \text{Depth}(G))H \quad (15)$$

1487 Then $l \leq \frac{|G|}{H} + \text{Depth}(G)$.

1488 Then $L \leq l \cdot \max_{v \in G} L_v = \left(\frac{|G|}{H} + \text{Depth}(G) \right) \max_{v \in G} L_v$.

1489

□

1490 **Proposition 15.** [Nearest Left/Right] For any local type T , consider the function that maps a
 1491 sequence of type $\text{Option}\langle T \rangle$ to nearest left/right neighbors that are not none. It's representable in
 1492 $\text{Tf}_{1,1}^{d+1}$

1493 *Proof.* There is only one layer and one head needed, so we can omit the layer and head index.

1494 WLOG, we consider the nearest left case.

1495 We just need to make the attention exponential look like this:

$$Q_p^\top K_{p'} + \lambda \Psi_{p'-p} = a_{\text{flag}, p'} - 1_{p'-p > 0}, \quad (16)$$

1496 where $a_{\text{flag}, p'} \in \{0, 1\}$ indicates whether the value at position p' is some or none.

1497 We set $V_{p'}$ to represent the value of type $\text{Option}\langle T \rangle$.

1498 For the starter token p_0 , we make it such that

$$Q_p^\top K_{p_0} + \lambda \Psi_{p_0-p} = 1, \quad (17)$$

1499 and

$$V_{p_0} = \mathbf{0}, \quad (18)$$

1500 so that when there are no some to the left, it will give us none. □

1501 **Proposition 16.** [Nearest Two Left/Right] For any local type T , consider the function that maps a
 1502 sequence of type $\text{Option}\langle T \rangle$ to nearest two left/right neighbors that are not none. It's representable
 1503 in $\text{Tf}_{O(1), O(1)}^{O(d)}$ where d is the encoding dimension of T .

1504 *Proof.* We can utilize Proposition 15 and 14.

1505 The nearest two left or right is equivalent to first computing the nearest left/right, and then packing
 1506 them together into one and compute its nearest left/right. The process is represented by a small
 1507 constant computation graph, then we're done. □

1508 E.7 Syntax and Semantics of Cybertron

1509 Having laid the necessary mathematical foundation behind **Cybertron**, we now turn to explaining
 1510 its surface—its **syntax** and **semantics**. Cybertron serves as a syntax sugar for expressing local
 1511 and global computation graphs, which are the vehicles used to demonstrate the expressive power of
 1512 transformers. In Cybertron, computations are divided into two layers: the **local world** and the **global**
 1513 **world**. These layers play distinct but complementary roles in constructing computation graphs.

1514 E.7.1 Local World

1515 The **local world** in Cybertron corresponds to the feed-forward layers of a transformer, focusing on
1516 computations over **local types**. Local types represent individual tokens or data points, and compu-
1517 tations in this world handle operations on tokens independently of their surrounding context.

1518 **Data Types.** Local types in Cybertron include basic types such as `Bool`, `Idx`, `Pos`, `Fin<n>`,
1519 `BoundedVec<T, N>`, etc. These types are essential for building local computation graphs that operate
1520 over individual tokens. Compound types, like **structs** and **enums**, can also be defined for more
1521 complex token representations. These types serve as the building blocks for the local computation
1522 graphs that transform data at the token level.

```
1523 1 struct Node {  
1524 2   id: Idx,  
1525 3   position: Pos,  
1526 4 }  
1527 5  
1528 6 enum Operation {  
1529 7   Add {  
1530 8     lhs: Pos,  
1531 9     rhs: Pos,  
1532 10  },  
1533 11   Multiply {  
1534 12     factor: Pos,  
1535 13   },  
1536 14 }
```

1537 **Functions.** Functions in the local world define operations upon information over individual tokens.
1538 These operations form nodes in the local computation graphs. For instance, operations like binary
1539 or unary expressions, conditionals, and matches on token types are transformed into computation
1540 graphs by handling each individual token's data.

```
1541 1 fn process_ast(ast: AstData) -> Option<Role> {  
1542 2   match ast {  
1543 3     AstData::LetInit { pattern, initial_value, .. } => {  
1544 4       Some(Role::LetStmt { pattern, initial_value })  
1545 5     }  
1546 6     AstData::Defn { keyword, ident, .. } => {  
1547 7       Some(match keyword {  
1548 8         DefnKeyword::Struct => Role::StructDefn(ident),  
1549 9         DefnKeyword::Enum => Role::EnumDefn(ident),  
1550 10        DefnKeyword::Fn => Role::FnDefn(ident),  
1551 11        })  
1552 12     }  
1553 13     _ => None,  
1554 14   }  
1555 15 }
```

1556 **Control Flow.** In the local world, control flow structures such as **if** and **match** are transformed
1557 into computation graphs by treating each branch or arm as an expression that returns an `Option`
1558 based on conditions. These `Option` values are then combined using the `Option::or` function. Ac-
1559 cording to **Proposition 9**, `Option::or` maps two `Option<T>` values and returns the first non- `None`
1560 value, or the second one otherwise. This allows conditional branches to be represented in com-
1561 putation graphs as sequential option evaluations, where the first matching condition provides the
1562 result.

1563 E.7.2 Global World

1564 The **global world** extends beyond individual tokens to sequences of tokens, represented as **global**
1565 **types**. These global types are denoted as `Seq<T>`, where `T` is a local type. The global world
1566 represents the full transformer, focusing on operations involving sequences of tokens, including
1567 variable definitions, expressions involving variable references, and function calls.

1568 **Variable Definitions.** Variables in the global world are defined using global types, which represent
1569 sequences of local tokens. These definitions correspond to nodes in the global computation graph.

1570 **Expressions.** Expressions in the global world consist of references to variables or function calls.
1571 Since the global world operates over sequences of tokens, these expressions are translated into
1572 sequence-level operations in the computation graph.

1573 **Function Calls.** Function calls are key elements of the global world. They are represented by
1574 applying global functions to sequences of tokens. Cybertron provides **map functions** to elevate
1575 local functions to global functions by mapping them across sequences. Additionally, **attention**
1576 **methods** like `nearest_left` and `nearest_right` handle dependencies between tokens in the sequence
1577 by identifying relationships based on their positions.

```
1578 1 let result = seq_of_values.nearest_left();
```

1579 In the global world, computation graphs are built by composing map functions and attention meth-
1580 ods. These graphs, unlike those in the local world, do not include control flow mechanisms.

1581 E.8 Dyck Language

1582 This section demonstrates how the **local world** in Cybertron operates over token-level computations
1583 and how the **global world** handles sequence-level operations. We use a Dyck language example to
1584 explain the interactions between these two worlds. The example processes a sequence of delimiters
1585 (like parentheses) and checks for matching pairs.

1586 **Local World.** In Cybertron, the **local world** operates on individual tokens. Here, the local types
1587 are simple, such as `Delimiter` and `PreAst`, which represent information associated with individual
1588 tokens. These types allow for token-level operations like comparisons and transformations.

1589 We define a **struct** to represent a delimiter and an **enum** to classify delimiters as either left or right.
1590 These definitions reflect local types, as they hold information over a single token.

```
1591 1 // Define a struct 'Delimiter' that wraps a 'u8' value.  
1592 2 #[derive(Debug, Clone, Copy, PartialEq, Eq)]  
1593 3 pub struct Delimiter(u8);  
1594 4  
1595 5 // Define an enum 'PreAst' which represents a left or right delimiter.  
1596 6 #[derive(Debug, Clone, Copy, PartialEq, Eq)]  
1597 7 pub enum PreAst {  
1598 8     LeftDelimiter(Delimiter),  
1599 9     RightDelimiter(Delimiter),  
1600 10 }
```

1601 Here, the local types `Delimiter` and `PreAst` define operations upon individual tokens, representing
1602 fundamental units of the computation graph at the local level. The local world is responsible for
1603 handling these small, token-level computations independently of the global sequence.

1604 **Global World.** In the **global world**, Cybertron operates on sequences of tokens, treating the col-
1605 lection of local types as a single unit of computation. The global world introduces global types such
1606 as `Seq<Option<PreAst>>`, which represents a sequence of optional delimiters. The global world han-
1607 dles sequence-level operations by applying functions like `nearest_left` and `nearest_right` to capture
1608 the relationships between tokens in the sequence.

1609 The following function operates on a sequence of `PreAst`, reducing matched pre-asts. The recursive
1610 application of `step` gives us the classifier for Dyck language.

```
1611 1 fn step(pre_asts: Seq<Option<PreAst>>) -> Seq<Option<PreAst>> {  
1612 2     let pre_asts_nearest_left = pre_asts.nearest_left();  
1613 3     let pre_asts_nearest_right = pre_asts.nearest_right();  
1614 4     step_aux.apply(pre_asts_nearest_left, pre_asts, pre_asts_nearest_right)  
1615 5 }
```

1616 **Local Worlds.** The `step_aux` function matches tokens based on their nearest neighbors within the
1617 sequence, eliminating pre-asts if a match is found.

```
1618 1 fn step_aux(  
1619 2     pre_ast_nearest_left: Option<(Idx, PreAst)>,
```

```

1620 3     pre_ast: Option<PreAst>,
1621 4     pre_ast_nearest_right: Option<Idx, PreAst>
1622 5 ) -> Option<PreAst> {
1623 6     match pre_ast? {
1624 7         PreAst::LeftDelimiter(delimiter) => match pre_ast_nearest_right {
1625 8             Some(⟦, PreAst::RightDelimiter(delimiter1)) if delimiter1 == delimiter =>
1626 9             None,
1627 10            _ => pre_ast,
1628 11            },
1629 12         PreAst::RightDelimiter(delimiter) => match pre_ast_nearest_left {
1630 13             Some(⟦, PreAst::LeftDelimiter(delimiter1)) if delimiter1 == delimiter =>
1631 14             None,
1632 15            _ => pre_ast,
1633 16            },
1634 17         }
1635 18     }

```

1636 In this example, the global function `step` uses `nearest_left` and `nearest_right` to capture sequence-
1637 level dependencies, while the local function `step_aux` uses conditional logic to check for matching
1638 pairs of delimiters. The local world handles token-level logic, while the global world coordinates
1639 operations across the entire sequence. This separation reflects how Cybertron handles computations
1640 at different levels of granularity.

1641 Thus, this example illustrates how Cybertron leverages both the local and global worlds to build
1642 comprehensive computation graphs in a convenient, comprehensive yet rigorous manner. The local
1643 world performs individual tokenwise operations, and the global world captures relationships be-
1644 tween tokens in a sequence, demonstrating how Cybertron enables transformers to express complex
1645 computations.

1646 F Transformer AST Proof

1647 F.1 High Level Overview

1648 Here we give the full details of the proof of transformers being able to parse ASTs.

1649 On a high level, we are going to see the parsing of ASTs as an assembly process. First, we im-
1650 mediately get the atomic ones, like identifiers, literals, etc. Then we assembly all composite ASTs
1651 with enough precedence until all tokens are consumed. We can prove that at the n th round, all ASTs
1652 with depth no more than n are already constructed. In the process, we must keep track of the un-
1653 consumed tokens and newly constructed ASTs (to be consumed as children for new ASTs in the
1654 next round, as we are going bottom up). We use `pre_ast`s to denote all the unconsumed tokens and
1655 newly constructed ASTs and use `ast`s to denote all the constructed(allocated) ASTs. For correctness
1656 guarantees, we give detailed type specifications for tokens, ASTs, and PreASTs as follows.

1657 We define the Token type as follows:

```

1658 1 // The 'Token' enum represents the various types of tokens that can be
1659 2 // identified during the lexical analysis phase of a compiler. Each variant
1660 3 // corresponds to a specific category of token that can be encountered
1661 4 // in the source code.
1662 5 pub enum Token {
1663 6     // A literal value, which can be a number, string, or other primitive type.
1664 7     Literal(Literal),
1665 8     // A reserved keyword in the language, such as 'if', 'else', 'while', etc.
1666 9     Keyword(Keyword),
1667 10    // An identifier, typically representing variable names, function names,
1668 11    // or other user-defined symbols.
1669 12    Ident(Ident),
1670 13    // An operator, such as '+', '-', '*', '==', etc., representing mathematical
1671 14    // or logical operations.
1672 15    Opr(Opr),
1673 16    // A left delimiter, such as '(', '{', '[', used to denote the beginning of
1674 17    // a block, list, or expression.
1675 18    LeftDelimiter(LeftDelimiter),
1676 19    // A right delimiter, such as ')', '}', ']', used to denote the end of a
1677 20    // block, list, or expression.
1678 21    RightDelimiter(RightDelimiter),
1679 22    // A separator, such as ',' or ';', used to separate elements in a list or
1680 23    // statements in a block.
1681 24    Separator(Separator),
1682 25 }

```

1683 The type has an encoding dimension $d_{\text{Token}} = \Theta(\log L)$, which is large enough to faithfully represent
1684 its information.

1685 More specifically, the types `Literal`, `Keyword`, `Ident`, `Opr`, `LeftDelimiter`, `RightDelimiter`,
1686 `Separator` are local types assumed to have encoding dimension less than d_{Token} . `Keyword`, `Opr`,
1687 `LeftDelimiter`, `RightDelimiter`, `Separator` are small, so they can be encoded in a straight-forward
1688 manner entirely using d_{Token} . However, `Literal` and `Ident` are larger than representable by a lim-
1689 ited number of bits because potentially a `Literal` can be a string literal of arbitrary length and an
1690 `Ident` can also be of arbitrary length. This can be solved through methods like interning, which
1691 gives all literals and identifiers that actually appear in the input distinct encodings. As the context
1692 length is L , the number of different literals/identifiers are bounded by context length and interning
1693 needs $O(d_{\text{Token}}) = O(\log L)$ to work. As far as our theories are concerned, it's totally reasonable to
1694 assume that all these types are assumed to have encoding dimension less than $d_{\text{Token}} = O(\log L)$.

1695 We define AST type as follows:

```
1696 1 /// Represents a node in an Abstract Syntax Tree (AST).  
1697 2 ///  
1698 3 /// Each 'Ast' node has a reference to its parent node (if any) and holds  
1699 4 /// the associated syntax data (such as expressions, statements, or other  
1700 5 /// constructs defined in the 'AstData' enum).  
1701 6 pub struct Ast {  
1702 7     /// The index of the parent node in the AST, if it exists.  
1703 8     ///  
1704 9     /// - 'Some(Idx)': The node has a parent, and 'Idx' represents its position.  
1705 10    /// - 'None': The node is the root or does not have a parent.  
1706 11    pub parent: Option<Idx>,  
1707 12    /// The data associated with this AST node.  
1708 13    ///  
1709 14    /// This field holds the actual syntax information, which is typically  
1710 15    /// defined by the 'AstData' enum. This could represent literals, expressions,  
1711 16    /// statements, and other constructs in the source language.  
1712 17    pub data: AstData,  
1713 18 }
```

1714 Note that we intentionally structure the tree by always storing the parent but not necessarily storing
1715 all children information. In our assumptions, we only control the depth of ASTs but don't control
1716 the number of children. More specifically, a function can have as many statements as possible. To
1717 avoid overflowing, we don't store all children information. As we shall see, parent information alone
1718 is enough for transformers to perform tree operations.

1719 The `AstData` is the most complicated we define in this paper, as follows:

```
1720 1 /// Enumeration representing different types of Abstract Syntax Tree (AST) nodes  
1721 2 pub enum AstData {  
1722 3     /// Represents a literal value (e.g., integer, string)  
1723 4     Literal(Literal),  
1724 5     /// Represents an identifier (e.g., variable name)  
1725 6     Ident(Ident),  
1726 7     /// Represents a prefix expression (e.g., '!x', '-x')  
1727 8     ///  
1728 9     /// # exprs  
1729 10    ///  
1730 11    Prefix {  
1731 12        /// Operator in the prefix expression (e.g., '!', '-')  
1732 13        opr: PrefixOpr,  
1733 14        /// Operand index of the expression  
1734 15        opd: Idx,  
1735 16    },  
1736 17    /// Represents a binary expression (e.g., 'x + y', 'a * b')  
1737 18    Binary {  
1738 19        /// Index of the left operand  
1739 20        lopr: Idx,  
1740 21        /// Operator in the binary expression (e.g., '+', '*')  
1741 22        opr: BinaryOpr,  
1742 23        /// Index of the right operand  
1743 24        ropd: Idx,  
1744 25    },  
1745 26    /// Represents a suffix expression (e.g., 'x++', 'y--')  
1746 27    Suffix {  
1747 28        /// Index of the operand  
1748 29        opd: Idx,  
1749 30        /// Operator in the suffix expression (e.g., '++', '--')
```

```

1750 31     opr: SuffixOpr,
1751 32 },
1752 33 // Represents a delimited expression (e.g., '(x + y)', '{a, b, c}')
1753 34 Delimited {
1754 35     // Index of the left delimiter in the expression
1755 36     left_delimiter_idx: Idx,
1756 37     // The left delimiter (e.g., '(', '{')
1757 38     left_delimiter: LeftDelimiter,
1758 39     // The right delimiter (e.g., ')', '}')
1759 40     right_delimiter: RightDelimiter,
1760 41 },
1761 42 // Represents an item separated by a separator (e.g., elements in an array or list)
1762 43 SeparatedItem {
1763 44     // Index of the content, if any
1764 45     content: Option<Idx>,
1765 46     // The separator (e.g., ',', ';')
1766 47     separator: Separator,
1767 48 },
1768 49 // Represents a function call or array access (e.g., 'f(...)', 'a[...]')
1769 50 //
1770 51 // things like 'f(...)' or 'a[...]
1771 52 Call {
1772 53     // Index of the caller (e.g., function or array)
1773 54     caller: Idx,
1774 55     // The left delimiter of the call (e.g., '(', '[')
1775 56     left_delimiter: LeftDelimiter,
1776 57     // The right delimiter of the call (e.g., ')', ']')
1777 58     right_delimiter: RightDelimiter,
1778 59     // Index of the delimited arguments in the call
1779 60     delimited_arguments: Idx,
1780 61 },
1781 62 // Represents a 'let' statement with an initialization (e.g., 'let x = 5;')
1782 63 //
1783 64 // # stmts
1784 65 //
1785 66 LetInit {
1786 67     // Index of the expression in the initialization
1787 68     expr: Idx,
1788 69     // Index of the pattern being initialized
1789 70     pattern: Idx,
1790 71     // Optional index of the initial value
1791 72     initial_value: Option<Idx>,
1792 73 },
1793 74 // Represents an 'if' statement
1794 75 If {
1795 76     // Index of the condition in the 'if' statement
1796 77     condition: Idx,
1797 78     // Index of the body of the 'if' statement
1798 79     body: Idx,
1799 80 },
1800 81 // Represents an 'else' statement
1801 82 Else {
1802 83     // Index of the associated 'if' statement
1803 84     if_stmt: Idx,
1804 85     // Index of the body of the 'else' statement
1805 86     body: Idx,
1806 87 },
1807 88 // Represents a function or variable definition
1808 89 //
1809 90 // # defn
1810 91 //
1811 92 Defn {
1812 93     // The keyword in the definition (e.g., 'fn', 'enum')
1813 94     keyword: DefnKeyword,
1814 95     // Index of the identifier in the definition
1815 96     ident_idx: Idx,
1816 97     // The identifier being defined (e.g., function name, variable name)
1817 98     ident: Ident,
1818 99     // Index of the content or body of the definition
1819 100    content: Idx,
1820 101 },
1821 102 }

1822 1 // The 'PreAst' enum represents the intermediate forms of tokens and ASTs that are
1823 2 // encountered during the parsing phase, before the final AST is constructed.
1824 3 // Each variant corresponds to a specific type of token or partial
1825 4 // AST node that contributes to the construction of the final AST.
1826 5 #[derive(Clone, Copy, PartialEq, Eq)]
1827 6 pub enum PreAst {
1828 7     // A reserved keyword in the language, such as 'if', 'else', 'while', etc.
1829 8     Keyword(Keyword),

```



```

1830 9    /// An operator, such as '+', '-', '*', '=', etc., representing mathematical
1831 10    /// or logical operations.
1832 11    Opr(Opr),
1833 12    /// A left delimiter, such as '(', '{', '[', used to denote the beginning of
1834 13    /// a block, list, or expression.
1835 14    LeftDelimiter(LeftDelimiter),
1836 15    /// A right delimiter, such as ')', '}', ']', used to denote the end of a
1837 16    /// block, list, or expression.
1838 17    RightDelimiter(RightDelimiter),
1839 18    /// A partially constructed AST node, representing a more complex structure
1840 19    /// that will be further processed to build the final AST.
1841 20    Ast(AstData),
1842 21    /// A separator, such as ', ' or ';', used to separate elements in a list or
1843 22    /// statements in a block.
1844 23    Separator(Separator),
1845 24 }

1846 1    /// this is beyond the scope of Cybertron
1847 2    ///
1848 3    /// rather a general Rust function to integrate for testing
1849 4    pub fn calc_ast_from_input(input: &str, n: usize) -> (Seq<Option<PreAst>>,
1850 5    Seq<Option<Ast>>) {
1851 6    let tokens = tokenize(input);
1852 7    let pre_ast = calc_pre_ast_initial_seq(tokens);
1853 8    let allocated_ast: Seq<Option<Ast>> = tokens.map(|token| token.into());
1854 9    reduce_n_times(pre_ast, allocated_ast, n)
1855 10 }

```

1856 The `reduce` function in Cybertron is designed to progressively refine sequences of pre-abstract
1857 syntax trees (pre-ASTs) and allocated abstract syntax trees (ASTs). The function takes two input
1858 sequences: `pre_ast`, which is a sequence of optional pre-ASTs, and `allocated_ast`, which is a
1859 sequence of optional ASTs. It returns a tuple containing the reduced sequences of pre-ASTs and
1860 allocated ASTs.

1861 The reduction process is carried out in multiple stages, each focusing on different syntactic con-
1862 structs:

- 1863 1. `reduce_by_opr` : This step handles reduction by dealing with operators and their precedence.
1864 It simplifies expressions involving operations to form more compact ASTs.
- 1865 2. `reduce_by_delimited` : This step reduces constructs that are delimited, such as those involv-
1866 ing parentheses, braces, or other grouping symbols. It ensures that delimited blocks are
1867 properly nested and combined in the AST.
- 1868 3. `reduce_by_call` : In this stage, function or method calls are reduced. This involves iden-
1869 tifying and structuring calls within the AST, ensuring correct representation of function
1870 invocations.
- 1871 4. `reduce_by_stmt` : This reduction step addresses statements, ensuring that individual state-
1872 ments are correctly parsed and represented within the AST, such as assignment statements,
1873 loops, and conditionals.
- 1874 5. `reduce_by_defn` : Finally, reduction by definition handles the parsing of definitions, such
1875 as variable or function declarations. This step ensures that all definitions are correctly
1876 represented within the AST.

1877 By sequentially applying these reduction steps, the `reduce` function progressively transforms the
1878 initial sequences into their most refined forms, ready for further syntactic or semantic analysis.

```

1879 1    pub fn reduce(
1880 2    pre_ast: Seq<Option<PreAst>>,
1881 3    allocated_ast: Seq<Option<Ast>>,
1882 4    ) -> (Seq<Option<PreAst>>, Seq<Option<Ast>>) {
1883 5    // Reduce ASTs by handling operators and precedence
1884 6    let (pre_ast, allocated_ast) = reduce_by_opr(pre_ast, allocated_ast);
1885 7
1886 8    // Reduce ASTs by handling delimited constructs like parentheses or braces
1887 9    let (pre_ast, allocated_ast) = reduce_by_delimited(pre_ast, allocated_ast);
1888 10
1889 11    // Reduce ASTs by handling function or method calls
1890 12    let (pre_ast, allocated_ast) = reduce_by_call(pre_ast, allocated_ast);

```

```

1891 13
1892 14 // Reduce ASTs by handling statements, ensuring proper syntax structure
1893 15 let (pre_ast, allocated_ast) = reduce_by_stmt(pre_ast, allocated_ast);
1894 16
1895 17 // Reduce ASTs by handling definitions, like variables or functions
1896 18 let (pre_ast, allocated_ast) = reduce_by_defn(pre_ast, allocated_ast);
1897 19
1898 20 // Return the final reduced sequences of pre-ASTs and allocated ASTs
1899 21 (pre_ast, allocated_ast)
1900 22 }

```

```

1901 1 pub fn reduce_n_times(
1902 2     mut pre_ast: Seq<Option<PreAst>>,
1903 3     mut allocated_ast: Seq<Option<Ast>>,
1904 4     n: usize,
1905 5 ) -> (Seq<Option<PreAst>>, Seq<Option<Ast>>) {
1906 6     for _ in 0..n {
1907 7         let (pre_ast1, allocated_ast1) = reduce(pre_ast, allocated_ast);
1908 8         pre_ast = pre_ast1;
1909 9         allocated_ast = allocated_ast1;
1910 10    }
1911 11    (pre_ast, allocated_ast)
1912 12 }

```

1913 In the above definition, we actually used Rust’s mutable variable semantics. However, it’s straight-
1914 forward to see that it translates to a computation graph that is a sequential composition of subgraphs
1915 with sequential length n . Because the AST’s depth is bounded by D , we can just take n to be D .
1916 Each subgraph is generated from the `reduce` function, then they are all constant graphs constructed
1917 by global and local functions, then by Proposition 13,11 and 2 they translate to transformers with
1918 $O(\log L + D)$ depth, model dimension, and number of heads, where $\log L$ comes from the encoding
1919 of types like `Token`.

1920 Below we give full details of the various reduction functions.

1921 As these are implemented as Rust functions, they have been tested against a number of inputs. We
1922 don’t guarantee an industry level of correctness, but the key point is well illustrated.

1923 F.2 Operators

1924 In this section, we lay down the definition of `reduce_by_opr`.

```

1925 1 pub(super) fn reduce_by_opr(
1926 2     pre_ast: Seq<Option<PreAst>>,
1927 3     allocated_ast: Seq<Option<Ast>>,
1928 4 ) -> (Seq<Option<PreAst>>, Seq<Option<Ast>>) {
1929 5     let pre_ast_nearest_left2 = pre_ast.nearest_left2();
1930 6     let pre_ast_nearest_right2 = pre_ast.nearest_right2();
1931 7     let new_opr_ast = new_opr_ast.apply(pre_ast_nearest_left2, pre_ast,
1932 8         pre_ast_nearest_right2);
1933 8     let (pre_ast_reduced, new_parents) = reduce_pre_ast_by_opr(pre_ast, new_opr_ast);
1934 9     let pre_ast = update_pre_ast_by_new_ast(pre_ast_reduced, new_opr_ast);
1935 10    let allocated_ast =
1936 11        allocate_ast_and_update_parents(allocated_ast, new_opr_ast, new_parents);
1937 12    (pre_ast, allocated_ast)
1938 13 }

```

```

1939 1 /// a finite function
1940 2 pub(crate) fn new_opr_ast(
1941 3     nearest_left2: Option2<Idx, PreAst>,
1942 4     current: Option<PreAst>,
1943 5     nearest_right2: Option2<Idx, PreAst>,
1944 6 ) -> Option<AstData> {
1945 7     let Some(PreAst::Opr(opr)) = current else {
1946 8         return None;
1947 9     };
1948 10    match opr {
1949 11        Opr::Prefix(opr) => {
1950 12            let Some((opd, PreAst::Ast(_)) = nearest_right2.first() else {
1951 13                return None;
1952 14            };
1953 15            if let Some( (_, ast) ) = nearest_right2.second() {
1954 16                match ast {
1955 17                    PreAst::Keyword(_) => (),
1956 18                    PreAst::Opr(right_opr) => match right_opr {

```

```

1957 19         Opr::Prefix(_) => (),
1958 20         Opr::Binary(right_opr) => {
1959 21             // every binary opr in our small language is left associative,
1960 22         so '<' instead of '<='
1961 23             if right_opr.precedence() > opr.precedence() {
1962 24                 return None;
1963 25             }
1964 26         }
1965 27         Opr::Suffix(right_opr) => {
1966 28             if right_opr.precedence() > opr.precedence() {
1967 29                 return None;
1968 30             }
1969 31         }
1970 32     },
1971 33     PreAst::Ast(_) => (),
1972 34     // function call or index takes higher precedence
1973 35     PreAst::LeftDelimiter(_) => return None,
1974 36     PreAst::RightDelimiter(_) => (),
1975 37     PreAst::Separator(_) => (),
1976 38     }
1977 39     };
1978 40     Some(AstData::Prefix { opr, opd })
1979 41     }
1980 42     Opr::Binary(opr) => {
1981 43         let Some((lopd, PreAst::Ast(_))) = nearest_left2.first() else {
1982 44             return None;
1983 45         };
1984 46         let Some((ropd, PreAst::Ast(_))) = nearest_right2.first() else {
1985 47             return None;
1986 48         };
1987 49         if let Some( (_, ast) ) = nearest_left2.second() {
1988 50             match ast {
1989 51                 PreAst::Keyword(kw) => (),
1990 52                 PreAst::Opr(left_opr) => match left_opr {
1991 53                     Opr::Prefix(left_opr) => {
1992 54                         if left_opr.precedence() >= opr.precedence() {
1993 55                             return None;
1994 56                         }
1995 57                     }
1996 58                     Opr::Binary(left_opr) => {
1997 59                         // every binary opr in our small language is left
1998 60                         associative, so '>=' instead of '>'
1999 61                         if left_opr.precedence() >= opr.precedence() {
2000 62                             return None;
2001 63                         }
2002 64                     }
2003 65                     Opr::Suffix(_) => (), // actually this will be a syntax error
2004 66                 },
2005 67                 PreAst::Ast(_) => {
2006 68                     if opr != BinaryOpr::LightArrow {
2007 69                         return None;
2008 70                     }
2009 71                 }
2010 72                 PreAst::LeftDelimiter(_) => (),
2011 73                 PreAst::RightDelimiter(_) => return None,
2012 74                 PreAst::Separator(_) => (),
2013 75             }
2014 76         };
2015 77         if let Some( (_, ast) ) = nearest_right2.second() {
2016 78             match ast {
2017 79                 PreAst::Keyword(kw) => match kw {
2018 80                     Keyword::ELSE => return None,
2019 81                     _ => (),
2020 82                 },
2021 83                 PreAst::Opr(right_opr) => match right_opr {
2022 84                     Opr::Prefix(_) => (), // actually this will be a syntax error
2023 85                     Opr::Binary(right_opr) => {
2024 86                         // every binary opr in our small language is left
2025 87                         associative, so '<' instead of '<='
2026 88                         if right_opr.precedence() > opr.precedence() {
2027 89                             return None;
2028 90                         }
2029 91                     }
2030 92                     Opr::Suffix(right_opr) => {
2031 93                         if right_opr.precedence() >= opr.precedence() {
2032 94                             return None;
2033 95                         }
2034 96                     }
2035 97                 },
2036 98                 // function call or index takes higher precedence
2037 99                 PreAst::LeftDelimiter(_) => return None,

```

```

2038 97         PreAst::RightDelimiter(_) => (),
2039 98         PreAst::Ast(_) => (),
2040 99         PreAst::Separator(_) => (),
2041 100     }
2042 101     };
2043 102     Some(AstData::Binary { lopd, opr, ropd })
2044 103 }
2045 104 Opr::Suffix(opr) => {
2046 105     let Some((opd, PreAst::Ast(_)) = nearest_left2.first() else {
2047 106         return None;
2048 107     };
2049 108     if let Some( (_, ast) ) = nearest_left2.second() {
2050 109         match ast {
2051 110             PreAst::Keyword(_) => (),
2052 111             PreAst::Opr(right_opr) => match right_opr {
2053 112                 Opr::Prefix(right_opr) => {
2054 113                     if right_opr.precedence() > opr.precedence() {
2055 114                         return None;
2056 115                     }
2057 116                 }
2058 117                 Opr::Binary(right_opr) => {
2059 118                     // every binary opr in our small language is left
2060 119                     // associative, so '<' instead of '<='
2061 120                     if right_opr.precedence() > opr.precedence() {
2062 121                         return None;
2063 122                     }
2064 123                 }
2065 124                 Opr::Suffix(_) => (),
2066 125             },
2067 126             PreAst::LeftDelimiter(_) => (),
2068 127             PreAst::RightDelimiter(_) => return None,
2069 128             PreAst::Ast(_) => return None,
2070 129             PreAst::Separator(_) => (),
2071 130         }
2072 131     };
2073 132     Some(AstData::Suffix { opr, opd })
2074 133 }
2075 134 }

```

```

2077 1 // returns sequence of remaining PreAsts and new parent idxs
2078 2 pub(crate) fn reduce_pre_ast_by_opr(
2079 3     pre_ast: Seq<Option<PreAst>>,
2080 4     new_ast: Seq<Option<AstData>>,
2081 5 ) -> (Seq<Option<PreAst>>, Seq<Option<Idx>>) {
2082 6     let new_ast_nearest_left = new_ast.nearest_left();
2083 7     let pre_ast = reduce_pre_ast_by_new_ast.apply(pre_ast, new_ast);
2084 8     let (pre_ast, new_parents) = reduce_pre_ast_by_opr_left
2085 9         .apply_enumerated(new_ast_nearest_left, pre_ast)
2086 10        .decouple();
2087 11     let new_ast_nearest_right = new_ast.nearest_right();
2088 12     reduce_pre_ast_by_opr_right
2089 13         .apply_enumerated(new_ast_nearest_right, pre_ast, new_parents)
2090 14         .decouple()
2091 15 }

```

```

2092 1 fn reduce_pre_ast_by_new_ast(pre_ast: Option<PreAst>, new_ast: Option<AstData>) ->
2093 2     Option<PreAst> {
2094 3     if new_ast.is_some() {
2095 4         None
2096 5     } else {
2097 6         pre_ast
2098 7     }
2099 8 }

```

```

2100 1 fn reduce_pre_ast_by_opr_left(
2101 2     idx: Idx,
2102 3     new_ast_nearest_left: Option<(Idx, AstData)>,
2103 4     pre_ast: Option<PreAst>,
2104 5 ) -> (Option<PreAst>, Option<Idx>) {
2105 6     let Some(pre_ast) = pre_ast else {
2106 7         return (None, None);
2107 8     };
2108 9     let Some((new_ast_idx, new_ast_data)) = new_ast_nearest_left else {
2109 10        return (Some(pre_ast), None);
2110 11    };
2111 12    match new_ast_data {
2112 13        AstData::Binary { ropd: opd, .. } | AstData::Prefix { opd, .. } if opd == idx => {
2113 14            (None, Some(new_ast_idx))
2114 15        }

```

```

2115 16     _ => (Some(pre_ast), None),
2116 17     }
2117 18 }

2118 1 fn reduce_pre_ast_by_opr_right(
2119 2     idx: Idx,
2120 3     new_ast_nearest_right: Option<(Idx, AstData)>,
2121 4     pre_ast: Option<PreAst>,
2122 5     new_parent: Option<Idx>,
2123 6 ) -> (Option<PreAst>, Option<Idx>) {
2124 7     let Some(pre_ast) = pre_ast else {
2125 8         return (None, new_parent);
2126 9     };
2127 10    if let Some(new_parent) = new_parent {
2128 11        return (None, Some(new_parent));
2129 12    }
2130 13    let Some((new_ast_idx, new_ast_data)) = new_ast_nearest_right else {
2131 14        return (Some(pre_ast), None);
2132 15    };
2133 16    match new_ast_data {
2134 17        AstData::Binary { lopd: opd, .. } | AstData::Suffix { opd, .. } if opd == idx => {
2135 18            (None, Some(new_ast_idx))
2136 19        }
2137 20        _ => (Some(pre_ast), None),
2138 21    }
2139 22 }

```

2140 E.3 Statements

2141 In this section, we lay down the definition of `reduce_by_stmt`.

```

2142 1 pub(super) fn reduce_by_stmt(
2143 2     pre_ast: Seq<Option<PreAst>>,
2144 3     allocated_ast: Seq<Option<Ast>>,
2145 4 ) -> (Seq<Option<PreAst>>, Seq<Option<Ast>>) {
2146 5     let pre_ast_nearest_left2 = pre_ast.nearest_left2();
2147 6     let pre_ast_nearest_right2 = pre_ast.nearest_right2();
2148 7     let new_stmt_ast =
2149 8         new_stmt_ast.apply(pre_ast_nearest_left2, pre_ast, pre_ast_nearest_right2);
2150 9     let (pre_ast, new_parents) = reduce_pre_ast_by_stmt(pre_ast, new_stmt_ast);
2151 10    let allocated_ast =
2152 11        allocate_ast_and_update_parents(allocated_ast, new_stmt_ast, new_parents);
2153 12    let pre_ast = update_pre_ast_by_new_ast(pre_ast, new_stmt_ast);
2154 13    (pre_ast, allocated_ast)
2155 14 }

```

```

2156 1 fn new_stmt_ast(
2157 2     pre_ast_nearest_left2: Option2<(Idx, PreAst)>,
2158 3     pre_ast: Option<PreAst>,
2159 4     pre_ast_nearest_right2: Option2<(Idx, PreAst)>,
2160 5 ) -> Option<AstData> {
2161 6     let PreAst::Keyword(Keyword::Stmt(kw)) = pre_ast? else {
2162 7         return None;
2163 8     };
2164 9     match kw {
2165 10        StmtKeyword::Let => {
2166 11            let Some((idx1, PreAst::Ast(ast))) = pre_ast_nearest_right2.first() else {
2167 12                return None;
2168 13            };
2169 14            if let Some((), pre_ast) = pre_ast_nearest_right2.second() {
2170 15                match pre_ast {
2171 16                    PreAst::Keyword(_) => (),
2172 17                    PreAst::Opr(_) | PreAst::LeftDelimiter(_) => return None,
2173 18                    PreAst::RightDelimiter(_) => (),
2174 19                    PreAst::Ast(_) => return None,
2175 20                    PreAst::Separator(separator) => match separator {
2176 21                        Separator::Comma => return None,
2177 22                        Separator::Semicolon => (),
2178 23                    },
2179 24                }
2180 25            }
2181 26            let (pattern, initial_value) = match ast {
2182 27                AstData::Binary {
2183 28                    lopd,
2184 29                    opr: BinaryOpr::Assign,
2185 30                    ropd,
2186 31                } => (lopd, Some(ropd)),

```

```

2187 32     AstData::Ident(_)
2188 33     | AstData::Prefix { .. }
2189 34     | AstData::Binary { .. }
2190 35     | AstData::Delimited { .. }
2191 36     | AstData::Call { .. } => (idx1, None),
2192 37     _ => return None,
2193 38     };
2194 39     Some(AstData::LetInit {
2195 40         expr: idx1,
2196 41         pattern,
2197 42         initial_value,
2198 43     })
2199 44     }
2200 45     StmtKeyword::If => {
2201 46         let Some((condition, PreAst::Ast(ast1))) = pre_ast_nearest_right2.first() else
2202 47     {
2203 48         return None;
2204 49     };
2205 49     let Some((
2206 50         body,
2207 51         PreAst::Ast(AstData::Delimited {
2208 52             left_delimiter: LCURL,
2209 53             right_delimiter: RCURL,
2210 54             ..
2211 55         })),
2212 56     ) = pre_ast_nearest_right2.second()
2213 57     else {
2214 58         return None;
2215 59     };
2216 60     Some(AstData::If { condition, body })
2217 61     }
2218 62     StmtKeyword::Else => {
2219 63         let Some((if_stmt, PreAst::Ast(AstData::If { .. }))) =
2220 64     pre_ast_nearest_left2.first()
2221 64     else {
2222 65         return None;
2223 66     };
2224 67     let Some((
2225 68         body,
2226 69         PreAst::Ast(
2227 70             AstData::Delimited {
2228 71                 left_delimiter: LCURL,
2229 72                 right_delimiter: RCURL,
2230 73                 ..
2231 74             }
2232 75             | AstData::If { .. }
2233 76             | AstData::Else { .. },
2234 77         ),
2235 78     ) = pre_ast_nearest_right2.first()
2236 79     else {
2237 80         return None;
2238 81     };
2239 82     if let Some( (_, PreAst::Keyword(Keyword::ELSE))) =
2240 83     pre_ast_nearest_right2.second() {
2241 83         return None;
2242 84     }
2243 85     Some(AstData::Else { if_stmt, body })
2244 86     }
2245 87     }
2246 88     }

```

```

2247 1 fn reduce_pre_ast_by_stmt (
2248 2     pre_ast: Seq<Option<PreAst>>,
2249 3     new_ast: Seq<Option<AstData>>,
2250 4 ) -> (Seq<Option<PreAst>>, Seq<Option<Idx>>) {
2251 5     let new_ast_nearest_left = new_ast.nearest_left();
2252 6     let new_ast_nearest_right = new_ast.nearest_right();
2253 7     reduce_pre_ast_by_stmt
2254 8         .apply_enumerated(new_ast_nearest_left, new_ast_nearest_right, pre_ast)
2255 9         .decouple()
2256 10 }

```

```

2257 1 fn reduce_pre_ast_by_stmt (
2258 2     idx: Idx,
2259 3     new_ast_nearest_left: Option<(Idx, AstData)>,
2260 4     new_ast_nearest_right: Option<(Idx, AstData)>,
2261 5     pre_ast: Option<PreAst>,
2262 6 ) -> (Option<PreAst>, Option<Idx>) {
2263 7     if let Some((idx1, ast)) = new_ast_nearest_left {
2264 8         match ast {

```

```

2265 9         AstData::LetInit { expr, .. } if expr == idx => (None, Some(idx1)),
2266 10         AstData::If {
2267 11             condition, body, ..
2268 12         } if condition == idx || body == idx => (None, Some(idx1)),
2269 13         AstData::Else { body, .. } if body == idx => (None, Some(idx1)),
2270 14         _ => (pre_ast, None),
2271 15     }
2272 16 } else if let Some((idx1, AstData::Else { if_stmt, .. })) = new_ast_nearest_right
2273 17     && if_stmt == idx
2274 18 {
2275 19     (None, Some(idx1))
2276 20 } else {
2277 21     (pre_ast, None)
2278 22 }
2279 23 }

```

2280 F.4 Generalized Call Forms

2281 In this section, we lay down the definition of `reduce_by_call`.

```

2282 1 pub(super) fn reduce_by_call(
2283 2     pre_ast: Seq<Option<PreAst>>,
2284 3     allocated_ast: Seq<Option<Ast>>,
2285 4 ) -> (Seq<Option<PreAst>>, Seq<Option<Ast>>) {
2286 5     let pre_ast_nearest_left2 = pre_ast.nearest_left2();
2287 6     let pre_ast_nearest_right = pre_ast.nearest_right();
2288 7     let new_call_ast =
2289 8         new_call_ast.apply_enumerated(pre_ast_nearest_left2, pre_ast_nearest_right);
2290 9     let (pre_ast, new_parents) = reduce_pre_ast_by_call(pre_ast, new_call_ast);
2291 10    let allocated_ast =
2292 11        allocate_ast_and_update_parents(allocated_ast, new_call_ast, new_parents);
2293 12    let pre_ast = update_pre_ast_by_new_ast(pre_ast, new_call_ast);
2294 13    (pre_ast, allocated_ast)
2295 14 }

```

```

2296 1 fn new_call_ast(
2297 2     idx: Idx,
2298 3     pre_ast_nearest_left2: Option2<(Idx, PreAst)>,
2299 4     pre_ast_nearest_right: Option<(Idx, PreAst)>,
2300 5 ) -> Option<AstData> {
2301 6     let (caller, PreAst::Ast(caller_ast)) = pre_ast_nearest_left2.first()? else {
2302 7         return None;
2303 8     };
2304 9     let (
2305 10         delimited_arguments,
2306 11         PreAst::Ast(AstData::Delimited {
2307 12             left_delimiter_idx,
2308 13             left_delimiter,
2309 14             right_delimiter,
2310 15         }),
2311 16     ) = pre_ast_nearest_right?
2312 17     else {
2313 18         return None;
2314 19     };
2315 20     if let Some((_, snd)) = pre_ast_nearest_left2.second() {
2316 21         match snd {
2317 22             PreAst::Keyword(kw) => match kw {
2318 23                 Keyword::Defn(kw) => match kw {
2319 24                     DefnKeyword::Struct | DefnKeyword::Enum => return None,
2320 25                     DefnKeyword::Fn => match left_delimiter.delimiter() {
2321 26                         Delimiter::Parenthesis | Delimiter::Box => return None,
2322 27                         Delimiter::Curly => (),
2323 28                     },
2324 29                 },
2325 30             Keyword::Stmt(kw) => match kw {
2326 31                 StmtKeyword::Let => (),
2327 32                 StmtKeyword::If => match left_delimiter.delimiter() {
2328 33                     Delimiter::Parenthesis | Delimiter::Box => (),
2329 34                     Delimiter::Curly => return None,
2330 35                 },
2331 36                 StmtKeyword::Else => return None,
2332 37             },
2333 38         },
2334 39         PreAst::Opr(opr) => match opr {
2335 40             Opr::Prefix(_) | Opr::Binary(_) => match left_delimiter.delimiter() {
2336 41                 Delimiter::Parenthesis | Delimiter::Box => (),
2337 42                 Delimiter::Curly => return None,
2338 43             },

```

```

2339 44     Opr::Suffix(_) => return None,
2340 45     },
2341 46     PreAst::LeftDelimiter(_) => (),
2342 47     PreAst::RightDelimiter(_) => return None,
2343 48     PreAst::Ast(snd_ast) => {
2344 49         if let AstData::Ident(_) = snd_ast
2345 50             && left_delimiter == LCURL
2346 51         {
2347 52             match caller_ast {
2348 53                 AstData::Binary {
2349 54                     opr: BinaryOpr::LightArrow,
2350 55                     ..
2351 56                 }
2352 57             | AstData::Delimited {
2353 58                 left_delimiter: LPAR,
2354 59                 right_delimiter: RPAR,
2355 60                 ..
2356 61             } => (),
2357 62             _ => return None,
2358 63         }
2359 64     } else {
2360 65         return None;
2361 66     }
2362 67 }
2363 68 PreAst::Separator(_) => (),
2364 69 }
2365 70 }
2366 71 if left_delimiter_idx != idx {
2367 72     return None;
2368 73 }
2369 74 Some(AstData::Call {
2370 75     caller,
2371 76     delimited_arguments,
2372 77     left_delimiter,
2373 78     right_delimiter,
2374 79 })
2375 80 }

```

```

2376 1 fn reduce_pre_ast_by_call(
2377 2     pre_ast: Seq<Option<PreAst>>,
2378 3     new_ast: Seq<Option<AstData>>,
2379 4 ) -> (Seq<Option<PreAst>>, Seq<Option<Idx>>) {
2380 5     let new_ast_nearest_left = new_ast.nearest_left();
2381 6     let new_ast_nearest_right = new_ast.nearest_right();
2382 7     reduce_pre_ast_by_call
2383 8         .apply_enumerated(new_ast_nearest_left, new_ast_nearest_right, pre_ast)
2384 9     .decouple()
2385 10 }

```

```

2386 1 fn reduce_pre_ast_by_call(
2387 2     idx: Idx,
2388 3     new_ast_nearest_left: Option<(Idx, AstData)>,
2389 4     new_ast_nearest_right: Option<(Idx, AstData)>,
2390 5     pre_ast: Option<PreAst>,
2391 6 ) -> (Option<PreAst>, Option<Idx>) {
2392 7     if let Some((
2393 8         idx1,
2394 9         AstData::Call {
2395 10             delimited_arguments,
2396 11             ..
2397 12         },
2398 13     )) = new_ast_nearest_left
2399 14         && delimited_arguments == idx
2400 15     {
2401 16         (None, Some(idx1))
2402 17     } else if let Some((idx1, AstData::Call { caller, .. })) = new_ast_nearest_right
2403 18         && caller == idx
2404 19     {
2405 20         (None, Some(idx1))
2406 21     } else {
2407 22         (pre_ast, None)
2408 23     }
2409 24 }

```

2410 F.5 Definitions

2411 In this section, we lay down the definition of `reduce_by_defn`.


```

2412 1 pub(super) fn reduce_by_defn(
2413 2     pre_ast: Seq<Option<PreAst>>,
2414 3     allocated_ast: Seq<Option<Ast>>,
2415 4 ) -> (Seq<Option<PreAst>>, Seq<Option<Ast>>) {
2416 5     let pre_ast_nearest_left2 = pre_ast.nearest_left2();
2417 6     let pre_ast_nearest_right2 = pre_ast.nearest_right2();
2418 7     let new_defn_ast =
2419 8         new_defn_ast.apply(pre_ast_nearest_left2, pre_ast, pre_ast_nearest_right2);
2420 9     let (pre_ast, new_parents) = reduce_pre_ast_by_defn(pre_ast, new_defn_ast);
2421 10    let allocated_ast =
2422 11        allocate_ast_and_update_parents(allocated_ast, new_defn_ast, new_parents);
2423 12    let pre_ast = update_pre_ast_by_new_ast(pre_ast, new_defn_ast);
2424 13    (pre_ast, allocated_ast)
2425 14 }

```

```

2426 1 fn new_defn_ast(
2427 2     pre_ast_nearest_left2: Option2<(Idx, PreAst)>,
2428 3     pre_ast: Option<PreAst>,
2429 4     pre_ast_nearest_right2: Option2<(Idx, PreAst)>,
2430 5 ) -> Option<AstData> {
2431 6     let PreAst::Keyword(Keyword::Defn(keyword)) = pre_ast? else {
2432 7         return None;
2433 8     };
2434 9     {
2435 10        let Some((ident_idx, PreAst::Ast(AstData::Ident(ident)))) =
2436 11            pre_ast_nearest_right2.first()
2437 12        else {
2438 13            return None;
2439 14        };
2440 15        let Some((content, PreAst::Ast(content_ast))) = pre_ast_nearest_right2.second()
2441 16        else {
2442 17            return None;
2443 18        };
2444 19        match keyword {
2445 20            DefnKeyword::Struct => match content_ast {
2446 21                AstData::Delimited { .. } => (),
2447 22                _ => return None,
2448 23            },
2449 24            DefnKeyword::Enum => match content_ast {
2450 25                AstData::Delimited { .. } => (),
2451 26                _ => return None,
2452 27            },
2453 28            DefnKeyword::Fn => match content_ast {
2454 29                AstData::Call { .. } => (),
2455 30                _ => return None,
2456 31            },
2457 32        }
2458 33        Some(AstData::Defn {
2459 34            keyword,
2460 35            ident_idx,
2461 36            ident,
2462 37            content,
2463 38        })
2464 39    }
2465 40 }

```

```

2466 1 fn reduce_pre_ast_by_defn(
2467 2     pre_ast: Seq<Option<PreAst>>,
2468 3     new_ast: Seq<Option<AstData>>,
2469 4 ) -> (Seq<Option<PreAst>>, Seq<Option<Idx>>) {
2470 5     let new_ast_nearest_left = new_ast.nearest_left();
2471 6     let new_ast_nearest_right = new_ast.nearest_right();
2472 7     reduce_pre_ast_by_defn
2473 8         .apply_enumerated(new_ast_nearest_left, new_ast_nearest_right, pre_ast)
2474 9     .decouple()
2475 10 }

```

```

2476 1 fn reduce_pre_ast_by_defn(
2477 2     idx: Idx,
2478 3     new_ast_nearest_left: Option<(Idx, AstData)>,
2479 4     new_ast_nearest_right: Option<(Idx, AstData)>,
2480 5     pre_ast: Option<PreAst>,
2481 6 ) -> (Option<PreAst>, Option<Idx>) {
2482 7     if let Some((idx1, ast)) = new_ast_nearest_left {
2483 8         match ast {
2484 9             AstData::Defn {
2485 10                 keyword,
2486 11                 ident_idx,
2487 12                 ident,
2488 13                 content,

```

```

2489 14      ..
2490 15      } if ident_idx == idx || content == idx => (None, Some(idx1)),
2491 16      _ => (pre_ast, None),
2492 17    }
2493 18  } else if let Some((idx1, AstData::Defn { .. })) = new_ast_nearest_right
2494 19    && false
2495 20  {
2496 21    (None, Some(idx1))
2497 22  } else {
2498 23    (pre_ast, None)
2499 24  }
2500 25 }

```

2501 G Transformer Symbol Resolution Proof

2502 Here we lay down the code for symbol resolution. The actual process involves many details such as
2503 computing ranks (the exact position of an AST node among its siblings), scopes, and roles (a more
2504 precise version of AST, computed from its parent recursively), definitions and resolutions.

2505 G.1 Ranks

```

2506 1  #[derive(Debug, Default, PartialEq, Eq, Clone, Copy)]
2507 2  pub struct Rank(u8);
2508 3
2509 4  impl Rank {
2510 5    fn next(self) -> Self {
2511 6      Self(self.0 + 1)
2512 7    }
2513 8  }
2514 9
2515 10 pub fn calc_ranks(asts: Seq<Option<Ast>>) -> Seq<Option<Rank>> {
2516 11   let counts = asts.count_past_by_attention(asts, |ast, ast1| {
2517 12     let Some(ast) = ast else { return false };
2518 13     let Some(ast1) = ast1 else { return false };
2519 14     ast.parent == ast1.parent
2520 15   });
2521 16   (|c: usize, ast| {
2522 17     ast?;
2523 18     Some(Rank(c.try_into().unwrap()))
2524 19   })
2525 20   .apply(counts, asts)
2526 21 }
2527 22
2528 23 pub fn calc_ranks1(asts: Seq<Option<Ast>>, n: usize) -> Seq<Option<Rank>> {
2529 24   let mut ranks: Seq<Option<Rank>> = asts.map(|_| None);
2530 25   for _ in 0..n {
2531 26     ranks = calc_sibling_indicies_step(asts, ranks);
2532 27   }
2533 28   ranks
2534 29 }
2535 30
2536 31 fn calc_sibling_indicies_step(
2537 32   asts: Seq<Option<Ast>>,
2538 33   ranks: Seq<Option<Rank>>,
2539 34 ) -> Seq<Option<Rank>> {
2540 35   let previous_ranks = ranks.nearest_left_filtered_by_attention(asts, asts, |ast, ast1| {
2541 36     let Some(ast) = ast else { return false };
2542 37     let Some(ast1) = ast1 else { return false };
2543 38     ast.parent == ast1.parent
2544 39   });
2545 40   let ranks = (|ast, rank, previous_rank: Option<Option<Rank>>| {
2546 41     let _ = ast?;
2547 42     if let Some(rank) = rank {
2548 43       return Some(rank);
2549 44     }
2550 45     let Some(previous_rank) = previous_rank else {
2551 46       return Some(Default::default());
2552 47     };
2553 48     Some(previous_rank?.next())
2554 49   })
2555 50   .apply(asts, ranks, previous_ranks);
2556 51   ranks
2557 52 }

```

2558 In the above, `count_past_by_attention` that count is representable by transformers by utilizing directly
 2559 hard attention and the starter token. If the count is c , we shall get $c/(c+1)$ from the attention directly.

2560 G.2 Scopes

```

2561 1 const D: usize = 8usize;
2562 2
2563 3 pub struct Scope {
2564 4     enclosing_blocks: BoundedVec<Idx, D>,
2565 5 }
2566 6
2567 7 impl Scope {
2568 8     pub fn from_ast(idx: Idx, ast: AstData, parent_scope: Scope) -> Self {
2569 9         match ast {
2570 10             AstData::Delimited {
2571 11                 left_delimiter_idx,
2572 12                 left_delimiter: LCURL,
2573 13                 right_delimiter: RCURL,
2574 14             } => Self {
2575 15                 enclosing_blocks: parent_scope.enclosing_blocks.append(idx),
2576 16             },
2577 17             _ => parent_scope,
2578 18         }
2579 19     }
2580 20
2581 21     pub fn new(idx: Idx) -> Self {
2582 22         Self {
2583 23             enclosing_blocks: todo!(),
2584 24         }
2585 25     }
2586 26
2587 27     pub fn append(self, idx: Idx) -> Self {
2588 28         Self {
2589 29             enclosing_blocks: self.enclosing_blocks.append(idx),
2590 30         }
2591 31     }
2592 32 }
2593 33
2594 34 impl Scope {
2595 35     pub fn contains(self, other: Self) -> bool {
2596 36         let len = self.enclosing_blocks.len();
2597 37         if len > other.enclosing_blocks.len() {
2598 38             return false;
2599 39         }
2600 40         for i in 0..len {
2601 41             if self.enclosing_blocks[i] != other.enclosing_blocks[i] {
2602 42                 return false;
2603 43             }
2604 44         }
2605 45         true
2606 46     }
2607 47 }
2608 48
2609 49 pub fn infer_scopes(asts: Seq<Option<Ast>>, n: usize) -> Seq<Option<Scope>> {
2610 50     let mut scopes = initial_scope.apply_enumerated(asts);
2611 51     for _ in 0..n {
2612 52         let parent_scopes = parent_queries(asts, scopes);
2613 53         scopes = infer_scopes_step(asts, parent_scopes, scopes);
2614 54     }
2615 55     scopes
2616 56 }
2617 57
2618 58 fn initial_scope(idx: Idx, ast: Option<Ast>) -> Option<Scope> {
2619 59     let ast = ast?;
2620 60     if ast.parent.is_some() {
2621 61         return None;
2622 62     }
2623 63     let scope = Scope::default();
2624 64     Some(Scope::from_ast(idx, ast.data, scope))
2625 65 }
2626 66
2627 67 fn infer_scopes_step(
2628 68     asts: Seq<Option<Ast>>,
2629 69     parent_scopes: Seq<Option<Scope>>,
2630 70     scopes: Seq<Option<Scope>>,
2631 71 ) -> Seq<Option<Scope>> {
2632 72     infer_scope_step.apply_enumerated(asts, parent_scopes, scopes)
2633 73 }
2634 74

```

```

2635 75 fn infer_scope_step(
2636 76   idx: Idx,
2637 77   ast: Option<Ast>,
2638 78   parent_scope: Option<Scope>,
2639 79   scope: Option<Scope>,
2640 80 ) -> Option<Scope> {
2641 81   if let Some(scope) = scope {
2642 82     return Some(scope);
2643 83   }
2644 84   Some(Scope::from_ast(idx, ast?.data, parent_scope?))
2645 85 }

```

2646 G.3 Roles

```

2647 1 #[derive(Debug, Clone, Copy, PartialEq, Eq)]
2648 2 pub enum Role {
2649 3   LetStmt {
2650 4     pattern: Idx,
2651 5     initial_value: Option<Idx>,
2652 6   },
2653 7   LetStmtInner {
2654 8     pattern: Idx,
2655 9     initial_value: Idx,
2656 10  },
2657 11   LetStmtIdent,
2658 12   LetStmtTypedVariables {
2659 13     variables: Idx,
2660 14     ty: Idx,
2661 15  },
2662 16   StructDefn(Ident),
2663 17   EnumDefn(Ident),
2664 18   FnDefn(Ident),
2665 19   FnDefnCallForm {
2666 20     fn_ident: Ident,
2667 21     scope: Scope,
2668 22  },
2669 23   FnParameters {
2670 24     fn_ident: Ident,
2671 25     has_return_ty: bool,
2672 26     scope: Scope,
2673 27  },
2674 28   FnParametersAndReturnType {
2675 29     fn_ident: Ident,
2676 30     parameters: Idx,
2677 31     scope: Scope,
2678 32     return_ty: Idx,
2679 33  },
2680 34   FnBody(Ident),
2681 35   StructFields(Ident),
2682 36   FnParameter {
2683 37     fn_ident: Ident,
2684 38     rank: Rank,
2685 39     ty: Idx,
2686 40     fn_ident_idx: Idx,
2687 41     scope: Scope,
2688 42  },
2689 43   FnParameterIdent {
2690 44     scope: Scope,
2691 45  },
2692 46   FnParameterSeparated {
2693 47     fn_ident: Ident,
2694 48     rank: Rank,
2695 49     scope: Scope,
2696 50  },
2697 51   FnParameterType {
2698 52     fn_ident: Ident,
2699 53     rank: Rank,
2700 54  },
2701 55   FnOutputType {
2702 56     fn_ident: Ident,
2703 57  },
2704 58   StructField {
2705 59     ty_ident: Ident,
2706 60     field_ident: Ident,
2707 61     ty_idx: Idx,
2708 62  },
2709 63   StructFieldType {
2710 64     ty_ident: Ident,
2711 65     field_ident: Ident,

```

```

2712 66     },
2713 67     TypeArgument,
2714 68     TypeArguments,
2715 69     StructFieldSeparated(Ident),
2716 70     LetStmtVariablesType,
2717 71     LetStmtVariables,
2718 72 }

```

```

2719 1  impl Ast {
2720 2     fn role(self) -> Option<Role> {
2721 3         match self.data {
2722 4             AstData::LetInit {
2723 5                 expr,
2724 6                 pattern,
2725 7                 initial_value,
2726 8             } => Some(Role::LetStmt {
2727 9                 pattern,
2728 10                initial_value,
2729 11            }),
2730 12            AstData::Defn {
2731 13                keyword,
2732 14                ident_idx,
2733 15                ident,
2734 16                content,
2735 17            } => Some(match keyword {
2736 18                DefnKeyword::Struct => Role::StructDefn(ident),
2737 19                DefnKeyword::Enum => Role::EnumDefn(ident),
2738 20                DefnKeyword::Fn => Role::FnDefn(ident),
2739 21            }),
2740 22            _ => None,
2741 23        }
2742 24    }
2743 25 }

```

```

2744 1  pub fn calc_roles(
2745 2     asts: Seq<Option<Ast>>,
2746 3     scopes: Seq<Option<Scope>>,
2747 4     n: usize,
2748 5 ) -> Seq<Option<Role>> {
2749 6     let mut roles: Seq<Option<Role>> = asts.map(|ast| ast?.role());
2750 7     let ranks = calc_ranks(ast);
2751 8     for _ in 0..n {
2752 9         let parent_roles = parent_queries(ast, roles);
2753 10        roles = calc_roles_step(ast, parent_roles, roles, ranks, scopes);
2754 11    }
2755 12    roles
2756 13 }

```

```

2757 1  fn calc_roles_step(
2758 2     asts: Seq<Option<Ast>>,
2759 3     parent_roles: Seq<Option<Role>>,
2760 4     roles: Seq<Option<Role>>,
2761 5     ranks: Seq<Option<Rank>>,
2762 6     scopes: Seq<Option<Scope>>,
2763 7 ) -> Seq<Option<Role>> {
2764 8     calc_role_step.apply_enumerated(ast, parent_roles, roles, ranks, scopes)
2765 9 }

```

```

2766 1  fn calc_role_step(
2767 2     idx: Idx,
2768 3     ast: Option<Ast>,
2769 4     parent_role: Option<Role>,
2770 5     role: Option<Role>,
2771 6     rank: Option<Rank>,
2772 7     scope: Option<Scope>,
2773 8 ) -> Option<Role> {
2774 9     if let Some(role) = role {
2775 10        return Some(role);
2776 11    }
2777 12    let ast = ast?;
2778 13    if let Some(role) = ast.role() {
2779 14        return Some(role);
2780 15    }
2781 16    match parent_role? {
2782 17        Role::LetStmt {
2783 18            pattern,
2784 19            initial_value,
2785 20        } => match ast.data {
2786 21            AstData::Ident(ident) if idx == pattern => Some(Role::LetStmtIdent),

```

```

2787 22     AstData::Binary {
2788 23         lopd,
2789 24         opr: BinaryOpr::Assign,
2790 25         ropd,
2791 26         lopd_ident,
2792 27     } if lopd == pattern => Some(Role::LetStmtInner {
2793 28         pattern,
2794 29         initial_value: ropd,
2795 30     }),
2796 31     _ => None,
2797 32 },
2798 33 Role::LetStmtInner {
2799 34     pattern,
2800 35     initial_value,
2801 36 } => {
2802 37     if idx == pattern {
2803 38         match ast.data {
2804 39             AstData::Ident(ident) => Some(Role::LetStmtIdent),
2805 40             AstData::Binary {
2806 41                 lopd,
2807 42                 lopd_ident,
2808 43                 opr,
2809 44                 ropd,
2810 45             } => Some(Role::LetStmtTypedVariables {
2811 46                 variables: lopd,
2812 47                 ty: ropd,
2813 48             }),
2814 49             _ => todo!(),
2815 50         }
2816 51     } else {
2817 52         None
2818 53     }
2819 54 }
2820 55 Role::LetStmtIdent => todo!(),
2821 56 Role::FnParameterIdent { scope } => todo!(),
2822 57 Role::StructDefn(ident) => match ast.data {
2823 58     AstData::Literal(_) => todo!(),
2824 59     AstData::Ident(_) => None,
2825 60     AstData::Prefix { opr, opd } => todo!(),
2826 61     AstData::Binary {
2827 62         lopd,
2828 63         opr,
2829 64         ropd,
2830 65         lopd_ident,
2831 66     } => todo!(),
2832 67     AstData::Suffix { opd, opr } => todo!(),
2833 68     AstData::Delimited {
2834 69         left_delimiter_idx,
2835 70         left_delimiter,
2836 71         right_delimiter,
2837 72     } => Some(Role::StructFields(ident)),
2838 73     AstData::SeparatedItem { content, separator } => todo!(),
2839 74     AstData::Call { .. } => todo!(),
2840 75     AstData::LetInit {
2841 76         expr,
2842 77         pattern,
2843 78         initial_value,
2844 79     } => todo!(),
2845 80     AstData::Return { result } => todo!(),
2846 81     AstData::Assert { condition } => todo!(),
2847 82     AstData::If { condition, body } => todo!(),
2848 83     AstData::Else { if_stmt, body } => todo!(),
2849 84     AstData::Defn {
2850 85         keyword,
2851 86         ident_idx,
2852 87         ident,
2853 88         content,
2854 89     } => todo!(),
2855 90 },
2856 91 Role::EnumDefn(_) => None, // ad hoc
2857 92 Role::FnDefn(fn_ident) => match ast.data {
2858 93     AstData::Literal(_) => todo!(),
2859 94     AstData::Ident(_) => None,
2860 95     AstData::Prefix { opr, opd } => todo!(),
2861 96     AstData::Binary {
2862 97         lopd,
2863 98         opr,
2864 99         ropd,
2865 100        lopd_ident,
2866 101    } => todo!(),
2867 102     AstData::Suffix { opd, opr } => todo!(),

```

```

2868 103     AstData::Delimited {
2869 104         left_delimiter_idx,
2870 105         left_delimiter,
2871 106         right_delimiter,
2872 107     } => todo!(),
2873 108     AstData::SeparatedItem { content, separator } => todo!(),
2874 109     AstData::Call {
2875 110         delimited_arguments,
2876 111         ..
2877 112     } => Some(Role::FnDefnCallForm {
2878 113         fn_ident,
2879 114         scope: match scope {
2880 115             Some(scope) => scope.append(delimited_arguments),
2881 116             None => Scope::new(delimited_arguments),
2882 117         },
2883 118     }),
2884 119     AstData::LetInit {
2885 120         expr,
2886 121         pattern,
2887 122         initial_value,
2888 123     } => todo!(),
2889 124     AstData::Return { result } => todo!(),
2890 125     AstData::Assert { condition } => todo!(),
2891 126     AstData::If { condition, body } => todo!(),
2892 127     AstData::Else { if_stmt, body } => todo!(),
2893 128     AstData::Defn {
2894 129         keyword,
2895 130         ident_idx,
2896 131         ident,
2897 132         content,
2898 133     } => todo!(),
2899 134 },
2900 135     Role::FnDefnCallForm { fn_ident, scope } => match ast.data {
2901 136         AstData::Literal(_) => todo!(),
2902 137         AstData::Ident(_) => todo!(),
2903 138         AstData::Prefix { opr, opd } => todo!(),
2904 139         AstData::Binary {
2905 140             lopd,
2906 141             opr,
2907 142             ropd,
2908 143             lopd_ident,
2909 144         } => {
2910 145             if opr == BinaryOpr::LightArrow {
2911 146                 Some(Role::FnParametersAndReturnType {
2912 147                     fn_ident,
2913 148                     parameters: lopd,
2914 149                     return_ty: ropd,
2915 150                     scope,
2916 151                 })
2917 152             } else {
2918 153                 unreachable!()
2919 154             }
2920 155         }
2921 156     AstData::Suffix { opd, opr } => todo!(),
2922 157     AstData::Delimited {
2923 158         left_delimiter_idx,
2924 159         left_delimiter,
2925 160         right_delimiter,
2926 161     } => match left_delimiter.delimiter() {
2927 162         Delimiter::Parenthesis => Some(Role::FnParameters {
2928 163             fn_ident,
2929 164             has_return_ty: false,
2930 165             scope,
2931 166         }),
2932 167         Delimiter::Box => todo!(),
2933 168         Delimiter::Curly => Some(Role::FnBody(fn_ident)),
2934 169     },
2935 170     AstData::SeparatedItem { content, separator } => todo!(),
2936 171     AstData::Call { .. } => todo!(),
2937 172     AstData::LetInit {
2938 173         expr,
2939 174         pattern,
2940 175         initial_value,
2941 176     } => todo!(),
2942 177     AstData::Return { result } => todo!(),
2943 178     AstData::Assert { condition } => todo!(),
2944 179     AstData::If { condition, body } => todo!(),
2945 180     AstData::Else { if_stmt, body } => todo!(),
2946 181     AstData::Defn {
2947 182         keyword,
2948 183         ident_idx,

```

```

2949 184         ident,
2950 185         content,
2951 186     } => todo!(),
2952 187 },
2953 188 Role::FnParameters {
2954 189     fn_ident, scope, ..
2955 190 } => match ast.data {
2956 191     AstData::Binary {
2957 192         lopd,
2958 193         opr,
2959 194         ropd,
2960 195         lopd_ident,
2961 196     } => {
2962 197         if opr == BinaryOpr::TypeIs {
2963 198             Some(Role::FnParameter {
2964 199                 fn_ident,
2965 200                 fn_ident_idx: lopd,
2966 201                 rank: rank.unwrap(),
2967 202                 ty: ropd,
2968 203                 scope,
2969 204             })
2970 205         } else {
2971 206             unreachable!()
2972 207         }
2973 208     }
2974 209     AstData::SeparatedItem { .. } => Some(Role::FnParameterSeparated {
2975 210         fn_ident,
2976 211         rank: rank.unwrap(),
2977 212         scope,
2978 213     }),
2979 214     _ => unreachable!(),
2980 215 },
2981 216 Role::FnBody(_) => None,
2982 217 Role::StructFields(ty_ident) => match ast.data {
2983 218     AstData::Binary {
2984 219         lopd,
2985 220         opr,
2986 221         ropd,
2987 222         lopd_ident,
2988 223     } => {
2989 224         assert_eq!(opr, BinaryOpr::TypeIs);
2990 225         Some(Role::StructField {
2991 226             ty_ident,
2992 227             field_ident: lopd_ident.unwrap(),
2993 228             ty_idx: ropd,
2994 229         })
2995 230     }
2996 231     AstData::SeparatedItem { content, separator } => {
2997 232         Some(Role::StructFieldSeparated(ty_ident))
2998 233     }
2999 234     _ => None,
3000 235 },
3001 236 Role::FnParameter {
3002 237     fn_ident,
3003 238     fn_ident_idx,
3004 239     rank,
3005 240     ty,
3006 241     scope,
3007 242     ..
3008 243 } => {
3009 244     if idx == ty {
3010 245         Some(Role::FnParameterType { fn_ident, rank })
3011 246     } else if idx == fn_ident_idx {
3012 247         Some(Role::FnParameterIdent { scope })
3013 248     } else {
3014 249         None
3015 250     }
3016 251 }
3017 252 Role::FnParameterSeparated {
3018 253     fn_ident,
3019 254     rank,
3020 255     scope,
3021 256 } => match ast.data {
3022 257     AstData::Binary {
3023 258         lopd,
3024 259         opr,
3025 260         ropd,
3026 261         lopd_ident,
3027 262     } => {
3028 263         if opr == BinaryOpr::TypeIs {
3029 264             Some(Role::FnParameter {

```



```

3030 265         fn_ident,
3031 266         fn_ident_idx: lopd,
3032 267         rank,
3033 268         ty: ropd,
3034 269         scope,
3035 270     })
3036 271     } else {
3037 272         unreachable!()
3038 273     }
3039 274     }
3040 275     _ => unreachable!(),
3041 276 },
3042 277 Role::StructField {
3043 278     ty_ident,
3044 279     field_ident,
3045 280     ty_idx,
3046 281 } => {
3047 282     if idx == ty_idx {
3048 283         Some(Role::StructFieldType {
3049 284             ty_ident,
3050 285             field_ident,
3051 286         })
3052 287     } else {
3053 288         None
3054 289     }
3055 290 }
3056 291 Role::StructFieldSeparated(ty_ident) => match ast.data {
3057 292     AstData::Binary {
3058 293         lopd,
3059 294         opr,
3060 295         ropd,
3061 296         lopd_ident,
3062 297     } => {
3063 298         assert_eq!(opr, BinaryOpr::TypeIs);
3064 299         Some(Role::StructField {
3065 300             ty_ident,
3066 301             field_ident: lopd_ident.unwrap(),
3067 302             ty_idx: ropd,
3068 303         })
3069 304     }
3070 305     _ => unreachable!(),
3071 306 },
3072 307 Role::FnParameterType { .. } | Role::StructFieldType { .. } | Role::TypeArgument
3073 => {
3074 308     match ast.data {
3075 309         AstData::Delimited {
3076 310             left_delimiter_idx,
3077 311             left_delimiter,
3078 312             right_delimiter,
3079 313         } => Some(Role::TypeArguments),
3080 314         _ => None,
3081 315     }
3082 316 }
3083 317 Role::TypeArguments => match ast.data {
3084 318     AstData::Ident(_) => Some(Role::TypeArgument),
3085 319     AstData::Delimited {
3086 320         left_delimiter_idx,
3087 321         left_delimiter,
3088 322         right_delimiter,
3089 323     } => todo!(),
3090 324     AstData::SeparatedItem { content, separator } => todo!(),
3091 325     AstData::Call {
3092 326         caller,
3093 327         caller_ident,
3094 328         left_delimiter,
3095 329         right_delimiter,
3096 330         delimited_arguments,
3097 331     } => todo!(),
3098 332     _ => None,
3099 333 },
3100 334 Role::FnParametersAndReturnType {
3101 335     fn_ident,
3102 336     parameters,
3103 337     return_ty,
3104 338     scope,
3105 339 } => {
3106 340     if idx == parameters {
3107 341         Some(Role::FnParameters {
3108 342             fn_ident,
3109 343             has_return_ty: true,
3110 344             scope,

```

```

3111 345         })
3112 346     } else if idx == return_ty {
3113 347         Some(Role::FnOutputType { fn_ident })
3114 348     } else {
3115 349         unreachable!()
3116 350     }
3117 351 }
3118 352 Role::FnOutputType { fn_ident } => todo!(),
3119 353 Role::LetStmtTypedVariables { variables, ty } => {
3120 354     if idx == variables {
3121 355         Some(Role::LetStmtVariables)
3122 356     } else if idx == ty {
3123 357         Some(Role::LetStmtVariablesType)
3124 358     } else {
3125 359         unreachable!()
3126 360     }
3127 361 }
3128 362 Role::LetStmtVariablesType => todo!(),
3129 363 Role::LetStmtVariables => todo!(),
3130 364 }
3131 365 }

```

3132 G.4 Defns

```

3133 1 #[derive(Debug, Clone, Copy, PartialEq, Eq)]
3134 2 pub struct SymbolDefn {
3135 3     pub symbol: Symbol,
3136 4     pub scope: Option<Scope>,
3137 5 }

```

```

3138 1 pub fn calc_symbol_defns(
3139 2     asts: Seq<Option<Ast>>,
3140 3     scopes: Seq<Option<Scope>>,
3141 4     n: usize,
3142 5 ) -> Seq<Option<SymbolDefn>> {
3143 6     let roles = calc_roles(asts, scopes, n);
3144 7     calc_symbol_defn.apply_enumerated(asts, roles, scopes)
3145 8 }

```

```

3146 1 fn calc_symbol_defn(
3147 2     idx: Idx,
3148 3     ast: Option<Ast>,
3149 4     role: Option<Role>,
3150 5     scope: Option<Scope>,
3151 6 ) -> Option<SymbolDefn> {
3152 7     match ast?.data {
3153 8         AstData::Ident(ident) => match role? {
3154 9             Role::LetStmt { .. } => unreachable!(),
3155 10            Role::LetStmtVariables | Role::LetStmtIdent => Some(SymbolDefn {
3156 11                symbol: Symbol {
3157 12                    ident,
3158 13                    source: idx,
3159 14                    data: SymbolData::Variable,
3160 15                },
3161 16                scope,
3162 17            }),
3163 18            Role::FnParameterIdent { scope } => Some(SymbolDefn {
3164 19                symbol: Symbol {
3165 20                    ident,
3166 21                    source: idx,
3167 22                    data: SymbolData::Variable,
3168 23                },
3169 24                scope: Some(scope),
3170 25            }),
3171 26            _ => None,
3172 27         },
3173 28         AstData::Defn {
3174 29             keyword,
3175 30             ident_idx,
3176 31             ident,
3177 32             content,
3178 33         } => Some(SymbolDefn {
3179 34             symbol: Symbol {
3180 35                 ident,
3181 36                 source: idx,
3182 37                 data: SymbolData::Item {
3183 38                     kind: keyword.into(),
3184 39                 },

```

```

3185     },
3186     scope,
3187   }},
3188   _ => None,
3189 }
3190 }

```

3191 G.5 Resolutions

```

3192 1 pub enum SymbolResolution {
3193 2   Ok(Symbol),
3194 3   Err(SymbolResolutionError),
3195 4 }

```

```

3196 1 pub enum SymbolResolutionError {
3197 2   NotResolved,
3198 3   NotYetDeclared(Symbol),
3199 4 }

```

```

3200 1 pub fn calc_symbol_resolutions(asts: Seq<Option<Ast>>, n: usize) ->
3201 2   Seq<Option<SymbolResolution>> {
3202 3   let scopes = infer_scopes(asts, n);
3203 4   let symbol_defns = calc_symbol_defns(asts, scopes, n);
3204 5   let idents = asts.map(|ast| match ast?.data {
3205 6     AstData::Ident(ident) => Some(ident),
3206 7     _ => None,
3207 8   });
3208 9   let symbols = symbol_defns
3209 10  .map(|symbol_defn| Some(symbol_defn?.symbol))
3210 11  .first_filtered_by_attention(
3211 12    (|ident, scope| (ident, scope).apply(idents, scopes),
3212 13    symbol_defns,
3213 14    |(ident, scope), symbol_defn| {
3214 15      let Some(ident) = ident else { return false };
3215 16      let Some(symbol_defn) = symbol_defn else {
3216 17        return false;
3217 18      };
3218 19      if let Some(symbol_defn_scope) = symbol_defn.scope {
3219 20        if !symbol_defn_scope.contains(scope.unwrap()) {
3220 21          return false;
3221 22        }
3222 23        symbol_defn.symbol.ident == ident
3223 24      },
3224 25    )
3225 26    .map(|s| s.flatten());
3226 27   finalize.apply_enumerated(idents, symbols)
3227 28 }

```

3229 In the above code, we use a somehow complicated attention which we should illustrate why it's
3230 representable by transformers. The essence is to prove `symbol_defn_scope.contains(scope.unwrap())`
3231 can be represented as part of the inner product in $Q^T K$. This can be done by looking closer to
3232 what `contains` does. Consider two scopes, `scope1` and `scope2`, which are sequences of bracket ast
3233 indices (can be null). The function returns true if the sequence of `scope1` contains the sequence of
3234 `scope2` as prefix, which can be achieved by $\sum_i x_i^T y_i$ where x_i, y_i are the encoding of i th ast indices
3235 of `scope1` and `scope2` after some transformations (different transformations because the function
3236 is asymmetric) so that $x_i^T y_i = 0$ if and only if either x_i is a None or x_i represents the same thing
3237 as y_i , and $x_i^T y_i < 0$ otherwise. More concretely, if x_i is a None, $x_i = \mathbf{0}$ by choice, and equal to
3238 $(1, u_i)$ otherwise where u_i corresponds to the encoding of the i th ast index of `scope1`; if y_i is a
3239 None, $y_i = \mathbf{0}$ by choice, and equal to $(-1, v_i)$ otherwise where $A > 0$ and v_i corresponds to the
3240 encoding of the i th ast index of `scope2`. We should choose the encoding u_i, v_i such that $u_i^T v_i = 1$
3241 if and only if they encode the same index, which is obviously easy enough.

```

3242 1 fn finalize(idx: Idx, ident: Option<Ident>, symbol: Option<Symbol>) ->
3243 2   Option<SymbolResolution> {
3244 3   let _ = ident?;
3245 4   let Some(symbol) = symbol else {
3246 5     return Some(SymbolResolution::Err(SymbolResolutionError::NotResolved));
3247 6   };

```

```

3248 6     match symbol.data {
3249 7         SymbolData::Item { .. } => (),
3250 8         SymbolData::Variable => {
3251 9             if idx < symbol.source {
3252 10                return Some(SymbolResolution::Err(
3253 11                    SymbolResolutionError::NotYetDeclared(symbol),
3254 12                ));
3255 13            }
3256 14        }
3257 15    }
3258 16    Some(SymbolResolution::Ok(symbol))
3259 17 }

```

3260 H Transformer Type Checking Proof

3261 Here we lay down the code for type analysis. It should be noted that we didn't completely implement
3262 all the details. Things like struct fields, enum variant fields are left out. However, we already cover
3263 the essential mechanism of type analysis, making it sufficient for proof purposes.

3264 H.1 Type Signatures

```

3265 1 #[derive(Debug, PartialEq, Eq, Clone, Copy)]
3266 2 pub struct TypeSignature {
3267 3     pub key: TypeSignatureKey,
3268 4     pub ty: Type,
3269 5 }

```

```

3270 1 #[derive(Debug, PartialEq, Eq, Clone, Copy)]
3271 2 pub enum TypeSignatureKey {
3272 3     FnParameter { fn_ident: Ident, rank: Rank },
3273 4     FnOutput { fn_ident: Ident },
3274 5     StructField { ty_ident: Ident, field_ident: Ident },
3275 6 }

```

```

3276 1 pub(super) fn calc_ty_signatures(
3277 2     asts: Seq<Option<Ast>>,
3278 3     roles: Seq<Option<Role>>,
3279 4     ty_terms: Seq<Option<Type>>,
3280 5 ) -> Seq<Option<TypeSignature>> {
3281 6     calc_ty_signature.apply(roles, ty_terms)
3282 7 }

```

```

3283 1 fn calc_ty_signature(role: Option<Role>, ty_term: Option<Type>) -> Option<TypeSignature> {
3284 2     let key = match role? {
3285 3         Role::FnParameterType { fn_ident, rank } => {
3286 4             TypeSignatureKey::FnParameter { fn_ident, rank }
3287 5         }
3288 6         Role::StructFieldType {
3289 7             ty_ident,
3290 8             field_ident,
3291 9         } => TypeSignatureKey::StructField {
3292 10            ty_ident,
3293 11            field_ident,
3294 12        },
3295 13         Role::FnOutputType { fn_ident } => TypeSignatureKey::FnOutput { fn_ident },
3296 14         Role::FnParameters {
3297 15             fn_ident,
3298 16             has_return_ty: false,
3299 17             scope,
3300 18         } => {
3301 19             let key = TypeSignatureKey::FnOutput { fn_ident };
3302 20             let ty = Type::new_ident(Ident::new("unit"));
3303 21             return Some(TypeSignature { key, ty });
3304 22         }
3305 23         _ => return None,
3306 24     };
3307 25     // put it here!
3308 26     let ty = ty_term?;
3309 27     Some(TypeSignature { key, ty })
3310 28 }

```

3311 H.2 Type Inference

```

3312 1 pub struct TypeInference {
3313 2     pub ty: Type,
3314 3 }

3315 1 pub fn calc_ty_inferences(
3316 2     asts: Seq<Option<Ast>>,
3317 3     symbol_resolutions: Seq<Option<SymbolResolution>>,
3318 4     roles: Seq<Option<Role>>,
3319 5     ty_terms: Seq<Option<Type>>,
3320 6     ty_signatures: Seq<Option<TypeSignature>>,
3321 7     n: usize,
3322 8 ) -> Seq<Option<TypeInference>> {
3323 9     let mut ty_inferences = infer_tys_initial(asts, ty_signatures);
3324 10    let mut ty_designations =
3325 11        calc_initial_ty_designations(asts, roles, symbol_resolutions, ty_inferences,
3326 12        ty_terms);
3327 12    for _ in 0..n {
3328 13        ty_inferences |= infer_tys_step(asts, symbol_resolutions, ty_inferences,
3329 14        ty_designations);
3330 14        ty_designations |= calc_ty_designations_step(roles, symbol_resolutions,
3331 15        ty_inferences);
3332 15    }
3333 16    ty_inferences
3334 17 }

3335 1 fn infer_tys_initial(
3336 2     asts: Seq<Option<Ast>>,
3337 3     ty_signatures: Seq<Option<TypeSignature>>,
3338 4 ) -> Seq<Option<TypeInference>> {
3339 5     inference_literal_tys(asts).or(infer_fn_call_tys(asts, ty_signatures))
3340 6 }

3341 1 fn inference_literal_tys(asts: Seq<Option<Ast>>) -> Seq<Option<TypeInference>> {
3342 2     asts.map(|ast| match ast?.data {
3343 3         AstData::Literal(lit) => match lit {
3344 4             Literal::Int(_) => Some(TypeInference {
3345 5                 ty: Type::new_ident(Ident::new("Int")),
3346 6             }),
3347 7             Literal::Float(_) => Some(TypeInference {
3348 8                 ty: Type::new_ident(Ident::new("Float")),
3349 9             }),
3350 10        },
3351 11        _ => None,
3352 12    })
3353 13 }

3354 1 fn infer_fn_call_tys(
3355 2     asts: Seq<Option<Ast>>,
3356 3     ty_signatures: Seq<Option<TypeSignature>>,
3357 4 ) -> Seq<Option<TypeInference>> {
3358 5     ty_signatures
3359 6         .first_filtered_by_attention(asts, ty_signatures, |ast, ty_signature| {
3360 7             let Some(ast) = ast else { return false };
3361 8             let Some(TypeSignature {
3362 9                 key: TypeSignatureKey::FnOutput { fn_ident },
3363 10                ..
3364 11            }) = ty_signature
3365 12         else {
3366 13             return false;
3367 14         };
3368 15         match ast.data {
3369 16             AstData::Call {
3370 17                 caller,
3371 18                 caller_ident,
3372 19                 left_delimiter,
3373 20                 right_delimiter,
3374 21                 delimited_arguments,
3375 22             } if caller_ident == Some(fn_ident) => true,
3376 23             _ => false,
3377 24         }
3378 25     })
3379 26     .map(|ty_inference| {
3380 27         Some(TypeInference {
3381 28             ty: ty_inference?.ty,
3382 29         })
3383 30     })
3384 31 }

```

3385 H.3 Type Expectations

```
3386 1 pub struct TypeExpectation {
3387 2     pub ty: Type,
3388 3     pub source: TypeExpectationSource,
3389 4 }

3390 1 pub enum TypeExpectationSource {
3391 2     CallArgument { caller_ident: Ident, rank: Rank },
3392 3 }

3393 1 pub fn calc_ty_expectations(
3394 2     asts: Seq<Option<Ast>>,
3395 3     ranks: Seq<Option<Rank>>,
3396 4     ty_signatures: Seq<Option<TypeSignature>>,
3397 5 ) -> Seq<Option<TypeExpectation>> {
3398 6     let parent_ast = asts.index(asts.map(|ast| ast?.parent)).map(Option::flatten);
3399 7     let grandparent_ast = asts
3400 8         .index(parent_ast.map(|parent_ast| parent_ast?.parent))
3401 9         .map(Option::flatten);
3402 10    let ty_expectation_sources = calc_ty_expectation_source.apply(grandparent_ast, ranks);
3403 11    let retrieved_ty_signatures = ty_signatures
3404 12        .first_filtered_by_attention(
3405 13            ty_expectation_sources,
3406 14            ty_signatures,
3407 15            |ty_expectation_source, ty_signature| {
3408 16                let Some(type_expectation_source) = ty_expectation_source else {
3409 17                    return false;
3410 18                };
3411 19                let Some(type_signature) = ty_signature else {
3412 20                    return false;
3413 21                };
3414 22                match (type_expectation_source, type_signature.key()) {
3415 23                    (
3416 24                        TypeExpectationSource::CallArgument {
3417 25                            caller_ident,
3418 26                            rank: rank0,
3419 27                        },
3420 28                        TypeSignatureKey::FnParameter {
3421 29                            fn_ident,
3422 30                            rank: rank1,
3423 31                        },
3424 32                    ) if caller_ident == fn_ident && rank0 == rank1 => true,
3425 33                    _ => false,
3426 34                }
3427 35            },
3428 36        )
3429 37        .map(Option::flatten);
3430 38    (|ty_expectation_source: Option<TypeExpectationSource>,
3431 39     retrieved_ty_signature: Option<TypeSignature>| {
3432 40        Some(TypeExpectation {
3433 41            ty: retrieved_ty_signature?.ty(),
3434 42            source: ty_expectation_source?,
3435 43        })
3436 44    })
3437 45    .apply(ty_expectation_sources, retrieved_ty_signatures)
3438 46 }

3439 1 fn calc_ty_expectation_source(
3440 2     grandparent_ast: Option<Ast>,
3441 3     rank: Option<Rank>,
3442 4 ) -> Option<TypeExpectationSource> {
3443 5     let grandparent_ast = grandparent_ast?;
3444 6     let rank = rank?;
3445 7     match grandparent_ast.data {
3446 8         AstData::Call {
3447 9             caller,
3448 10            caller_ident: Some(caller_ident),
3449 11            left_delimiter,
3450 12            right_delimiter,
3451 13            delimited_arguments,
3452 14        } => Some(TypeExpectationSource::CallArgument { caller_ident, rank }),
3453 15        _ => None,
3454 16    }
3455 17 }
```

3456 H.4 Type Errors

```

3457 1 pub enum TypeError {
3458 2     TypeMismatch { expected: Type, actual: Type },
3459 3 }

3460 1 pub fn calc_ty_errors(
3461 2     ty_inferences: Seq<Option<TypeInference>>,
3462 3     ty_expectations: Seq<Option<TypeExpectation>>,
3463 4 ) -> Seq<Option<TypeError>> {
3464 5     calc_ty_error.apply(ty_inferences, ty_expectations)
3465 6 }

3466 1 fn calc_ty_error(
3467 2     ty_inference: Option<TypeInference>,
3468 3     ty_expectation: Option<TypeExpectation>,
3469 4 ) -> Option<TypeError> {
3470 5     let ty_inference = ty_inference?;
3471 6     let ty_expectation = ty_expectation?;
3472 7     if ty_inference.ty == ty_expectation.ty {
3473 8         None
3474 9     } else {
3475 10        Some(TypeError::TypeMismatch {
3476 11            expected: ty_expectation.ty,
3477 12            actual: ty_inference.ty,
3478 13        })
3479 14    }
3480 15 }

```

3481 I Lower Bounds

```

3482 1 struct <ty-ident-1> {}
3483 2 struct <ty-ident-2> {}
3484 3 struct <ty-ident-3> {}
3485 4 struct <ty-ident-4> {}
3486 5
3487 6 fn <f-ident-1>(a: <arg-ty-ident-1>) {}
3488 7 fn <f-ident-2>(a: <arg-ty-ident-2>) {}
3489 8 fn <f-ident-3>(a: <arg-ty-ident-3>) {}
3490 9 fn <f-ident-4>(a: <arg-ty-ident-4>) {}
3491 10
3492 11 fn g() {
3493 12     let x: <ty-ident> = ...;
3494 13     <f-ident>(x);
3495 14 }

```

3496 I.1 Lower bounds for RNN: Easy Bounds due to Memory

3497 *Proof of Theorem 4.* Our proof resonates with the proof of Theorem 4.6 in [Wen et al. \(2024\)](#)
3498 and Theorem 8 in [Bhattamishra et al. \(2024\)](#). For $L, D, H \in \mathbb{N}$, suppose that D makes
3499 $\text{MiniHuskyAnnotated}_{D,H}$ to be nontrivial, i.e., one can define functions with one parameter and
3500 use function calls. Simple calculations shows we can choose $D = 7$ and $H = 1$. If a RNN rep-
3501 represents a function maps any token sequence of length L in $\text{MiniHuskyAnnotated}_{D,H}$ to its type
3502 errors represented as a sequence of values of type `Option<TypeError>`, then the memory right be-
3503 fore type checking must store all previous type signatures, the number of which can be as many as
3504 $\Omega(L)$ in the worst case. Assuming proper numerical discretization, the memorization of these type
3505 signatures would require the memory size to be $\Omega(L)$ in the worst case. \square

3506 J Additional Experiment Details

3507 J.1 Setups

3508 Model details are shown in Table 1, and other hyperparameters are shown in Table 2.

3509 J.2 Additional Results

3510 Figures 4,5,6,7 are other metrics in the experiments. Here the loss function is the summation of cross
3511 entropies for each sub task.

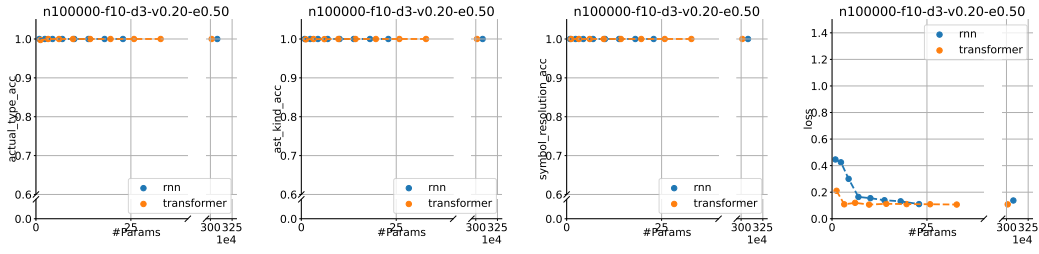


Figure 4: Figures for the dataset with $(f, d, v, e) = (10, 3, 0.2, 0.5)$.

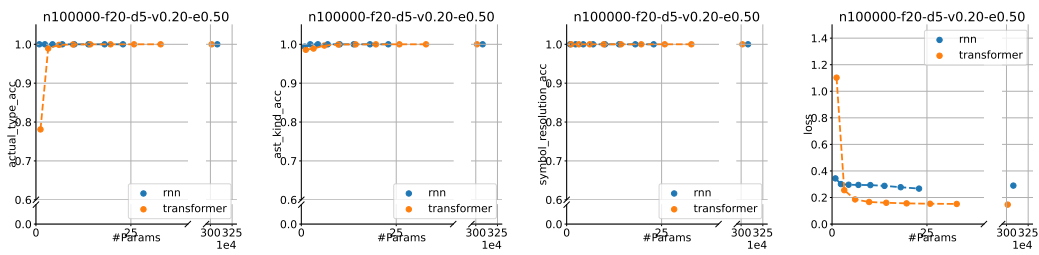


Figure 5: Figures for the dataset with $(f, d, v, e) = (20, 5, 0.2, 0.5)$.

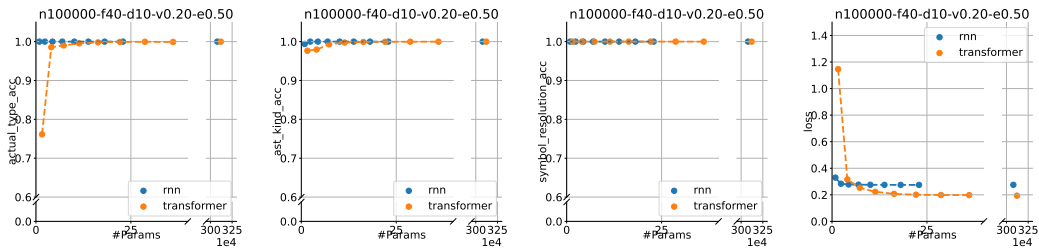


Figure 6: Figures for the dataset with $(f, d, v, e) = (40, 10, 0.2, 0.5)$.

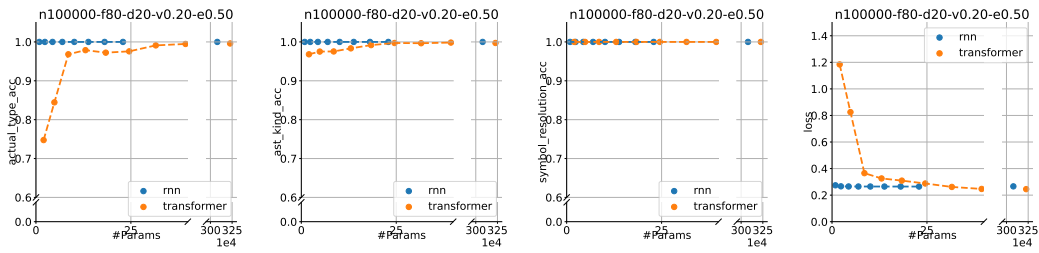


Figure 7: Figures for the dataset with $(f, d, v, e) = (80, 20, 0.2, 0.5)$.

Table 1: Model specification

Specification	Value
Transformer	
- Hidden size (d_h)	$\{8k 1 \leq k \leq 8\}$ $\cup \{208\}$
- Num attention heads	$\min\{4, d_h\}$
- Num hidden layers	8
- Intermediate size	$2d_h$
- Max position embeddings	≤ 2048
RNN	
- Hidden size	$\{8k 1 \leq k \leq 8\}$ $\cup \{256\}$
- Num layers	8

Table 2: Hyperparameters of experiments

Hyperparameter	Value
Dataset	
- (f, d)	$\{(10, 3), (20, 5)$ $(40, 10), (80, 20)\}$
- (n, v, e)	$(10^5, 0.2, 0.5)$
Number of epochs	20
Train batch size	512
Optimizer	Adam
LR scheduler	Linear warmup-decay
- Warmup min lr	1×10^{-5}
- Warmup max lr	1×10^{-3}
- Warmup steps	990