TYPE-CONSTRAINED CODE GENERATION WITH LANGUAGE MODELS

Niels Mündler[1,*] **Jingxuan He**[2,*] **Hao Wang**[2] **Koushik Sen**[2] **Dawn Song**[2] **Martin Vechev**[1]
[1] ETH Zurich, [2] UC Berkeley
`{niels.muendler,martin.vechev}@inf.ethz.ch`
`{hwang628,ksen,dawnsong}@berkeley.edu`

ABSTRACT

Large language models (LLMs) have achieved notable success in code generation. However, they still frequently produce uncompilable output because their next-token inference procedure does not model formal aspects of code. Although constrained decoding is a promising approach to alleviate this issue, it has only been applied to handle either domain-specific languages or syntactic language features. This leaves typing errors, which are beyond the domain of syntax and generally hard to adequately constrain. To address this challenge, we introduce a type-constrained decoding approach that leverages type systems to guide code generation. We develop novel prefix automata for this purpose and introduce a sound approach to enforce well-typedness based on type inference and a search over inhabitable types. We formalize our approach on a simply-typed language and extend it to TypeScript to demonstrate practicality. Our evaluation on HumanEval shows that our approach reduces compilation errors by more than half and increases functional correctness in code synthesis, translation, and repair tasks across LLMs of various sizes and model families, including SOTA open-weight models with more than 30B parameters.

1 INTRODUCTION

Large language models (LLMs) are remarkably successful in diverse fields (OpenAI, 2023; Brown et al., 2020; Grattafiori et al., 2024) and increasingly used in everyday coding tasks (GitHub, [n. d.]; Vella, 2024). Despite these achievements, LLM-generated code often contains compilation errors, logic flaws, or security vulnerabilities (Pan et al., 2024a; Dou et al., 2024; Pearce et al., 2022). These issues arise because LLM-based code generation does not provide any formal guarantees.

Constrained decoding addresses this by enforcing formal rules during LLMs' generation process, rejecting invalid completions. Prior work showed that constraining to syntactic rules improves program correctness (Beurer-Kellner et al., 2024; Ugare et al., 2024; Poesia et al., 2022). However, the improvements are limited, as syntax accounts for only a small part of program correctness. Type systems detect and reject bugs at compile time (Mitchell, 1990; Matsakis and Klock, 2014) and are enforced in popular programming languages (Bierman et al., 2014; Donovan and Kernighan, 2015; Arnold and Gosling, 1996). We thus suggest using type systems for constrained decoding.

The advantages of leveraging type systems are illustrated with five completion candidates for a program in Fig. 1. Based on syntax, completions that contain line termination (`;`) could be rejected (1). However, other cases, such as (2)-(4), are syntactic but cause type errors. Candidate (2) accesses an undeclared identifier. Candidate (3) will fail at execution, as `num` is a number and can not be called. Candidate (4) passes a value of unexpected format to `parseInt`, which expects the first argument to be a string. In this example, (4) is generated by CodeLlama 34B (Rozière et al., 2023). Syntax-only constraining accepts this invalid completion, leading to a failing output. However, based on constraints derived from the type system, all but the valid completion (5) are rejected.

Implementing type-constrained decoding is challenging because type systems cannot be captured by context-free grammars (Mitchell, 1990), prohibiting the application of prior constrained decoding

[*] Co-leading authors. Long-format paper available as pre-print at https://arxiv.org/abs/2504.09246.

```
function is_int(text: string): boolean {
  const num = Number(text);
  return !isNaN(num) &&
    parseInt(num <completion>
```

	<completion>	Vanilla	Syntax	Types	Description
(1)	;	accept	reject	reject	Syntactically invalid
(2)	ber	accept	accept	reject	Undeclared identifier
(3)	()	accept	accept	reject	Disallowed operator
(4)	, 10)	accept	accept	reject	Invalid argument type
(5)	.toString()	accept	accept	accept	Well-formed option

Figure 1: The left part is a partial TypeScript program derived from instance #113 of the MBPP benchmark (Austin et al., 2021), awaiting completion. On the right is a table listing five completion options: (1)-(4) are invalid and (5) is well-formed. Our type-constrained decoding is the only approach capable of correctly rejecting invalid completion candidates and accepting the valid one.

methods for program syntax (Ugare et al., 2024; Beurer-Kellner et al., 2024). Specifically, when generating expressions, we must address a key question: Can the partial expression be completed to match the required type? Essentially, this involves solving generalized type inhabitation (Urzyczyn, 1997; Gvero et al., 2013), a challenging problem that is PSPACE-complete even for simply typed lambda calculus, in the novel context of LLM-based code generation.

In this work, we introduce *type-constrained* decoding to enforce the formal rules of type systems on LLM-based code generation. In order to reason about the completability of partial code, we develop *prefix-automata*, which ensure that every visitable automaton state has a path to an accepting state. We then present a sound algorithm that determines types expressable through completions using type-search and an incrementally updated abstract syntax tree, including type annotations. To demonstrate its practical effectiveness, we instantiate our approach on a non-trivial subset of TypeScript, currently one of the most actively used languages (Madnight, 2024; GitHub, 2022). We perform an evaluation on TypeScript-translated HumanEval (Cassano et al., 2023) on synthesis, translation, and repair. Our experimental results show that type-constrained decoding significantly enhances code generation for LLMs of various sizes (2B-34B parameters) and families. For synthesis and translation, it reduces compilation errors by more than half and increases functional correctness relatively by 3.5% to 4.5%. Additionally, it enhances relative functional correctness of repaired code by around 34% on average. Our code implementation is publicly available.[1]

LLM-based Code Generation LLM-based code generation is incremental, depicted in Alg. 1 (without blue highlights). User prompt x specifies a task. The output program s is initialized to an empty string or program prefix in x (L. 1). In each iteration (L. 3), based on concatenation $x \circ s$ of prompt x and partial program s, the LLM predicts a probability distribution $\boldsymbol{p}$ over a fixed, finite *vocabulary* of *tokens*, comprised of unicode characters, including common singleton characters (Sennrich et al., 2016). Next, based on distribution $\boldsymbol{p}$, token t is sampled (L. 5) and appended to s (L. 10). This process repeats until special token EOS is sampled (L. 9).

Algorithm 1 Vanilla and constrained decoding

Input: LLM, prompt x, completion engine CE_L
Output: Program s such that $s \in L$

1: initialize s
2: **while true do**
3: $\quad \boldsymbol{p} \leftarrow \text{LLM}(x \circ s)$
4: $\quad$ **while true do**
5: $\quad\quad t \sim \boldsymbol{p}$
6: $\quad\quad$ **if** $CE_L(s \circ t)$ **then break**
7: $\quad\quad$ **elif** $s \in L$ **and** $t = EOS$ **then break**
8: $\quad\quad$ **else** $\boldsymbol{p}[t] \leftarrow 0$; normalize $\boldsymbol{p}$
9: $\quad$ **if** $t = EOS$ **then break**
10: $\quad s \leftarrow s \circ t$
11: **return** s

Constrained Decoding *Constrained decoding* analyzes intermediate model outputs to enforce validity of next tokens. It leverages a *completion engine* CE_L, specific to language L. The engine checks whether partial program s can be completed to a well-formed program in L, meaning whether there exists a (possibly empty) string s' such that $s \circ s' \in L$. Equivalently, $CE_L(s)$ determines whether s belongs to the *prefix language* L^p of L, i.e., whether $s \in L^p$.

Definition 1. *For a given language L, its prefix language is* $L^p = \{s \mid \exists s' : s \circ s' \in L\}$.

As illustrated with blue highlights in Alg. 1, constrained decoding differs from vanilla LLM-based code generation by adding an additional sample-and-check loop in the token sampling process L. 4. A sampled token t is appended only if $s \circ t$ can be completed to a well-formed program (L. 6) or t is EOS and s is already well-formed (L. 7). Otherwise, the probability of t is set to zero at L. 8, and the sample-and-check loop repeats. A token t satisfying L. 6 or L. 7 always exists, as s is in

[1] GitHub repository: https://github.com/eth-sri/type-constrained-code-generation

l ::=	Literal
`\d+`	Numeric Literal
`"\w*"`	String Literal
`true` $\mid$ `false`	Boolean Literal
T ::=	Type
`number`	Numeric Type
`string`	String Type
`boolean`	Boolean Type
$(\overline{p})$ `=>` T	Function Type
$p ::= x : T$	Typed Identifier

$e ::= e_0 \mid e_1$	Expression
e_0 ::=	Base Expression
l	Literal
x	Identifier
$(\overline{p})$ `=>` e	Function Expression
(e)	Grouped Expression
e_1 ::=	Extension Expression
$e \odot e$	Binary Operator
$e(\overline{e})$	Function Call
$e.n$	Member Access

x ::= `\w+`	Identifier
s ::=	Statement
`let` $x : T$;	Variable Declaration
e;	Expression Statement
`return` e;	Return Statement
$\{ \overline{s} \}$	Statement Block
`function` $x\ (\overline{p}) : T\ \{ \overline{s} \}$	Function Definition
`if` (e) s `else` s	If-Then-Else Statement
$M ::= \overline{s}$	Program

Figure 2: The syntax of L_B. Expressions are categorized into base and extension expressions. The later extends a given expression with suffix operators to form more complicated expressions.

L^p and LLMs' vocabulary contains all common characters. Therefore, the iterations of the sample-and-check loop is bounded by the size of the vocabulary. In practice, only one or a few iterations are typically needed.

Constrained decoding not only provides token-level guarantees but also ensures the final program's validity based on L. This whole-program guarantee can be proved inductively. At L. 1, we start with a valid prefix in L^p, i.e., either an empty string or a valid prefix provided in the user prompt. The check at L. 6 ensures that all intermediate model outputs remain prefixes in L^p. Additionally, L. 7 and L. 9 ensure that L. 11 is reached only if $s \in L$. In practice, the only other possible outcome is that the generation does not terminate within a predefined time or token limit.

2 Type Constrained Decoding

2.1 A Simply Typed Language

We define a simply typed, Turing-complete language, L_B. The syntax of expressions, types, and statements match standard definitions and are a subset of TypeScript (Bierman et al., 2014). It includes expressions, typed declaration statements, type-annotated functions, and flow control. Its complete syntax is shown in Fig. 2 in Extended Backus Naur Form. In the spirit of Bierman et al. (2014), we use a bar to denote Kleene-Plus over repeated elements, i.e., $X^+ = \overline{X}$.

The type system of L_B matches the type system of TypeScript and other conventional programming languages. Specifically, expressions are typed based on a type environment Γ, a map from identifiers to types. We write $\Gamma \vdash e : T$ if expression e has type T in type environment Γ. The type environment is propagated through consecutive statements, block-scoped and updated by declarations.

A word is valid in L_B if it (i) is syntactically valid, (ii) adheres to derived types for expressions, and (iii) ensures returning correctly-typed values on every execution path. Detailed syntax and type inference rules are presented in App. D.

2.2 Prefix Automaton Definition

Prefix automata are standard automata that ensure a special *prefix property*[2]. This property enables using the prefix automaton as a completion engine CE_L to facilitate constrained decoding.

We consider an automaton $A := \langle \Sigma, Q, \delta, I, F \rangle$, a tuple of : (i) Σ an alphabet of input symbols; (ii) Q a set of states; (iii) $\delta : Q \times \Sigma \mapsto \mathcal{P}(Q)$ a computable transition function mapping to finite sets; (iv) $I \subseteq Q$ a finite set of initial states; and (v) $F \subseteq Q$ a decidable set of accepting states. We denote a symbol in Σ as c, a string of symbols in Σ^* as s, and an operator for concatenating symbols and strings as $\circ$. The transition function δ maps a state to all possible subsequent states. When δ is applied on a set $\mathbf{q} \subseteq Q$, we take the union, i.e., $\delta(\mathbf{q}, c) = \bigcup_{q \in \mathbf{q}} \delta(q, c)$. The transition function defines a directed graph G over Q, where every state is a node and there is an edge annotated with c from q to q' iff $q' \in \delta(q, c)$. The language parsed by A comprises all strings s such that traversing G from some state in I along the edges annotated $c \in s$, it is possible to reach some state in F. Formally, we define recursive traversal function γ for states $\mathbf{q}$ as $\gamma(\mathbf{q}, \varepsilon) := \mathbf{q}$ and $\gamma(\mathbf{q}, s \circ c) := \gamma(\delta(\mathbf{q}, s), c)$. The language accepted by A is defined as $L(A) := \{s \mid \gamma(I, s) \cap F \neq \varnothing\}$. The traversal function has two intuitive properties concerning reachability that can be shown inductively:

[2]Note that the prefix property defined in our work differs from the one discussed in classical texts, e.g., (Hopcroft and Ullman, 1979)

(P1) A path along the graph can be split arbitrarily, i.e. $\gamma(\mathbf{q}, s \circ s') = \gamma(\gamma(\mathbf{q}, s), s')$.
(P2) If $s \circ s'$ reaches some state, s reaches some state, i.e., $\gamma(\mathbf{q}, s \circ s') \neq \varnothing \implies \gamma(\mathbf{q}, s) \neq \varnothing$.

A is a *prefix automaton* if there is a path from every reachable state to some accepting state:

Definition 2. *For an automaton A, the prefix property holds iff $\forall q \in Q : q \in \gamma(I, s) \implies \exists s' : \gamma(q, s') \cap F \neq \varnothing$. The automaton is a prefix automaton if it satisfies the prefix property.*

Intuitively, for such A, reaching any state through s implies that s is prefix to some $s' \in L(A)$. The *reachable language* of A, all inputs that result in any state, is $L_r(A) := \{s \mid \gamma(I, s) \neq \varnothing\}$. We see that $L_r(A)$ is equivalent to $L(A)^p$, the prefix language of $L(A)$ as defined in Def. 1.

Lemma 1. *If A is a prefix automaton, then $L(A)^p = L_r(A)$.*

Proof. $s \in L(A)^p$ implies that there exists s' such that $s \circ s' \in L(A)$, by the definition of prefix languages. This means $\gamma(I, s \circ s') \neq \varnothing$. Then, using (P2), we further derive $\gamma(I, s) \neq \varnothing$, i.e., $s \in L_r(A)$. Therefore, $L(A)^p \subseteq L_r(A)$ holds. The other direction also holds. We first see that $s \in L_r(A) \implies \gamma(I, s) \neq \varnothing$. Then applying Def. 2 and (P1), we find $\exists s' : \gamma(I, s \circ s') \cap F \neq \varnothing$, implying $s \circ s' \in L(A)$ and thus $s \in L(A)^p$. □

From Prefix Automata to Completion Engines With Lem. 1, given a prefix automaton A, we define a completion engine for $L(A)$: $CE_{L(A)}(s) := \gamma(I, s) \neq \varnothing$. To use A to constrain generation to L, we need to establish a relationship between $L(A)$ and L. If $L(A) \subseteq L$, we are guaranteed that LLM generations constrained by $CE_{L(A)}$ lie in L. Conversely, if $L(A) \supseteq L$, we are guaranteed that every $s \in L$ can be expressed under constrained decoding, but not that all generations are valid. For example, if A permits syntactically correct programs, it guarantees all well-typed programs can be generated, but permits ill-typed programs. To enforce well-typeness, we thus require $L(A) \subseteq L$.

Building a Prefix Automaton for L_B We choose Σ to be the set of Unicode characters, making our completion engine agnostic to LLM vocabularies. When our completion engine is called during constrained decoding at L. 6 of Alg. 1, it processes the sampled token by characters.

Before proceeding, we briefly introduce several base prefix automata below, with their precise definitions detailed in App. E.1. These automata are later combined, with parts of the transition function being overwritten, to construct more complex automata.

- Union $A_X \cup A_Y$ parses the language $\{s \mid s \in L(A_X) \cup L(A_Y)\}$. It is a prefix automaton if both A_X and A_Y are prefix automata.
- Concatenation $A_{XY} = A_X \circ A_Y$ parses the language $\{s \circ s' \mid s \in L(A_X), s' \in L(A_Y)\}$. It is a prefix automaton if A_X and A_Y are both prefix automata, and $L(A_Y) \neq \varnothing$.
- Kleene-Star $A_{\overline{X}}$ parses the language $\{\overline{s} \mid s \in L(A_X)\}$, a prefix automaton if A_X is a prefix automaton.
- Terminal $A_{\texttt{S}}$ parses the language $\{\texttt{S}\}$, where $\texttt{S}$ is a fixed, non-empty string.
- Empty $A_\varnothing$ parses the empty language $\varnothing$ and is always a prefix automaton.

2.3 Prefix Automata for Identifiers, Literals, and Types

Literals The prefix automaton for literals $A_l := A_{\text{NUM}} \cup A_{\text{STR}} \cup A_{\text{BOOL}}$ accepts number, string, and boolean literals as defined in Fig. 2. The automata A_{NUM}, A_{STR}, and A_{BOOL} are defined by the deterministic finite automaton representation of their regular expressions. To ensure the prefix property, we prune states from which accepting states can not be reached.

Identifiers During parsing, we maintain the current type environment Γ. We define the identifier automaton A_x as the union of the terminal automata for identifiers defined in Γ, $A_x := \bigcup_{y \in \Gamma} A_{\texttt{y}}$.

Types The type automaton A_T accepts type annotations. It is defined as $A_T := A_{\text{TYPE-LIT}} \cup A_{\text{TYPE-FUN}}$, including type literal automaton $A_{\text{TYPE-LIT}} := A_{\texttt{string}} \cup A_{\texttt{number}} \cup A_{\texttt{boolean}}$ and function type automaton $A_{\text{TYPE-FUN}} := A_{\texttt{(}\overline{p}\texttt{) => } T}$. The latter is a concatenation of multiple prefix automata, with the parameter and return types recursing on A_T. This recursive definition is valid, since it ensures a finite set of initial states, defines a decidable accepting set, and preserves the prefix property.

2.4 PREFIX AUTOMATON FOR EXPRESSIONS

Unrestricted Expressions We differentiate two kinds as shown in Fig. 2: *Base expressions*; identifiers, literals, grouped expressions, and anonymous functions, and *extension expressions*; operator applications (binary operator, member access, or function call) that extend a given expression.

The expression automaton A_e is the union of base expression automata A_x, A_l, $A_{(e)}$, and $A_{(\overline{p})\texttt{ => }e}$, with potential extensions $A_{\odot e}$, $A_{.n}$, and $A_{(\overline{e})}$. The base and extension automata are constructed by concatenating the respective terminal automata and recursively A_e. Additionally, we restrict the type of the recursive A_e if required by the type system. We provide additional detail on the implementation in App. E.2. Since an expression can end after base and extensions, accepting states of both base and extending automata are accepting states of A_e. To implement extensions in the transition function δ_e, we start from the base expression automata and adjust δ_e with outgoing edges from accepting states to the initial states of the extending automata, formally:

$$\forall X, Y : \delta_e(q_Y^X, c) \coloneqq \begin{cases} \delta_Y(q_Y^X, c) \cup \delta_e(I_{(\overline{e})}^X, c) \cup \delta_e(I_{\odot e}^X, c) \cup \delta_e(I_{.n}^X, c) & \text{if } q_Y^X \in F_Y \\ \delta_Y(q_Y^X, c) & \text{otherwise,} \end{cases}$$

where the labels X and Y for a state q_Y^X represent that a string X has been parsed, and currently the active automaton is A_Y, one of A_x, A_l, $A_{\odot e}$, etc. The superscripts are leveraged to determine the validity of extensions and transition to type-restricted expressions based on L_B's typing rules. In general, we set $I_Y^X \coloneqq \varnothing$ when Y is an invalid extension to X. Our definition of A_e is valid, as δ_e is computable, I_e is finite and F_e is decidable.

Type-Constrained Expressions To implement $A_e \downarrow T$, we must determine whether a partial expression s can be completed to inhabit type T. Completing s may express different types, and repeated extensions can alter the result type, but do not guarantee that the desired type can be expressed. However, extensions can be applied indefinitely, prohibiting an exhaustive enumeration.

We therefore develop a two-tiered algorithm. It first identifies *derivable types* $\text{DERIVABLE}(q_s)$ of s, inhabitable without extensions, based on its current state q_s. Second, a *type reachability search* $\text{REACHABLE}(\text{DERIVABLE}(q_e), T)$ determines if T can be inhabited by extending from the derivable types of s. We prune automaton transitions when this type search returns a negative result.

Table 1: Definition of $\text{DERIVABLE}(x)$ for partial expressions in Fig. 2. $s \leq s'$ means s is a prefix of s'. $\text{pmatch}(s, T)$ returns whether prefix s partially matches regular expression of literals of type T.

s	$\text{DERIVABLE}(q_s)$
l	$\{T \mid \text{pmatch}(l, T), T \in \{\texttt{number}, \texttt{string}, \texttt{boolean}\}\}$
x	$\{T \mid x \leq n, (n : T) \in \Gamma\}$
$(\overline{p})\texttt{ => }e$	$\{(\overline{p})\texttt{ => }T \mid \text{REACHABLE}(\text{DERIVABLE}(q_e), T)\}$
$(e$	$\{T \mid \text{REACHABLE}(\text{DERIVABLE}(q_e), T)\}$
$e \odot$	$\{T \mid \exists S' : \Gamma \vdash e : S \land S \odot S' : T\}$
$e($	$\{R \mid \Gamma \vdash e : (\overline{p})\texttt{ => }R\}$
$e.a$	$\{S \mid a \leq n, \Gamma \vdash e : T, \text{LOOKUP}(T, n) = S\}$

Derivable Types For DERIVABLE, we enumerate all types inhabitable by the currently parsed expression s without extension. For a final state q of expression e, we define $\text{DERIVABLE}(q) \coloneqq T$, where $\Gamma \vdash e : T$. We implement the unique derivability rules of expressions, detailed in Tab. 1. Note that for grouped expressions and function literals, we need to enumerate reachable types by recursively contained expressions. To avoid explicitly enumerating all reachable types, we integrate the derivability and reachability algorithms, discussed in more detail in App. E.4.

Lemma 2. *For state $q \in r(I_e, s)$ of partial expression s, $\text{DERIVABLE}(q)$ returns all T s.t. exists some suffix s' with $\Gamma \vdash s \circ s' : T$ and s' does not involve an extension.*

Type Reachability To determine which types are inhabitable by extending a base expression e of type T, we analyze sequences of extensions with compatible signatures. This process is conceptualized as a search over a graph where types as nodes and extension steps are edges. For binary operator $\odot$ with signature $T \odot X : S$, an edge is created from type T to type S. As an example, the operator for numerical addition $+$ has the signature $\texttt{number} + \texttt{number} : \texttt{number}$, thereby forming an edge from `number` to itself. Furthermore, for every member n of type T, we create an edge from T to $\text{LOOKUP}(T, \text{n})$. Finally, we connect T and R for each function with type $T = (\overline{p})\texttt{ => }R$. Note that these steps are *abstract*, i.e., they focus on the expression types, disregarding textual representation.

The type reachability algorithm, Alg. 2, implements a depth-first search over this type graph. Starting from current type T, it succeeds upon finding goal type G (L. 2), marking visited types to prevent cycles (L. 3). It iterates over valid extensions (L. 4) and computes result type S of the extension (L. 5). At L. 7, we proceed to recursively search G can be reached from S. If all recursive calls are unsuccessful, the goal type G can not be reached (L. 8).

Algorithm 2 Type reachability search

Input: Current type T of expression e, goal type G
Output: Whether G can be reached by extending e.

```
1: function REACHABLE(T, G)
2:     if T = G then return true
3:     if T is marked then return false else mark T
4:     for each valid extension step ◇ from T do
5:         S ← the resulting type of applying ◇ on T
6:         if PRUNESEARCH(T, G, S) continue
7:         if REACHABLE(S, G) return true
8:     return false
```

Some programming languages define self-referential default members, e.g., `clone` in Java or `valueOf` in TypeScript, nullary functions returning the type of the callee, $() \Rightarrow T$ for T. By accessing these members, higher-order types can be derived indefinitely. To ensure termination, we therefore restrict the type search to a finite set of types. At L. 6 of Alg. 2, we use heuristic PRUNESEARCH to restrict the search. We develop simple heuristic based on Gvero et al. (2013) to prune exploration of types with higher complexity than goal or source type, if they do not contain yet unexplored primitive types. We detail this heuristic in App. E.3. While ensuring termination, the heuristic leads to incompleteness and the potential rejection of well-typed expressions. However, in practical usage, only highly complex (thus less used) types are avoided.

Lemma 3. *The type search in Alg. 2 is* sound*, i.e., for any expression e with $\Gamma \vdash e : T$, if* $\text{REACHABLE}(T, G)$ *holds, then there exists a sequence of extensions y such that $\Gamma \vdash e \circ y : G$.*

Note that any pruning heuristic at L. 6 of Alg. 2 preserves soundness, which is sufficient to preserve the prefix property of $A_e \downarrow T$. We conclude from Lems. 2 and 3 that $A_e \downarrow T$ and A_e are prefix automata that parse a subset of well-typed expressions in L_B.

Corollary 1. *If* $\text{REACHABLE}(\text{DERIVABLE}(q), G)$ *holds for any $q \in \gamma(I_e, s)$ of a partial expression s, then there exists a suffix s' such that $\Gamma \vdash s \circ s' : G$.*

Lemma 4. *The language parsed by $A_e \downarrow T$ is thus a subset of the expressions of L_B of type T, i.e., $L(A_e \downarrow T) \subseteq \{s \mid \Gamma \vdash s : T\}$. Since A_e recursively involves $A_e \downarrow T$, the language parsed by A_e is also a subset of well-typed expressions of L_B, i.e., $L(A_e) \subseteq \{s \mid \exists T : \Gamma \vdash s : T\}$.*

2.5 Prefix Automata for Statements

The statement automaton is defined recursively as $A_s := A_{\text{DECL}} \cup A_{\text{EXPR}} \cup A_{\text{RET}} \cup A_{\text{BLOCK}} \cup A_{\text{FUN}} \cup A_{\text{ITE}}$. The declaration automaton $A_{\text{DECL}} := A_{\texttt{let}\ x\ :\ T;}$ captures undefined variable names x by accepting all strings, except for existing identifiers. It is a prefix automaton as an accepting state can be reached by appending characters. A_{RET} is $A_{\varnothing}$ when outside a function and otherwise restricts the parsed expression to the return type of the surrounding function. The remaining automata are mainly concatenations of previous automata and A_s, with small variations detailed in App. E.5.

Guaranteeing Return Types When parsing function bodies, the transition function of A_{FUN} maintains information about the declared return type and encountered return statements (if any). A_{FUN} only accepts when all return values match the declared return type, and all execution paths in the function body return, as in Fig. 8. Otherwise, another statement generation is forced. The prefix automaton property is preserved as we can always express a correct return statement using literals.

The described rules are implemented without violating the prefix property by deriving restrictions from already parsed input and ensuring completion is possible. We conclude that A_s is a prefix automaton. Moreover, A_s accepts all valid statements of L_B, excluding expressions rejected by A_e.

Lemma 5. *With $A_{L_B} := A_{\overline{s}}$ it holds that A_{L_B} is a prefix automaton and $L(A_{L_B}) \subseteq L_B$*

3 Experimental Evaluation

We outline our evaluation setup, with further details and hyperparameters discussed in App. G.

Tasks and Benchmarks We evaluate on three relevant tasks of code generation: (i) *Synthesis*: Given a natural language task, generate a solution from scratch. (ii) *Translation*: Given a Python function, generate an equivalent TypeScript function. (iii) *Repair*: Given a natural language task, a non-compilable solution and compiler error, restore functionality of the flawed solution.

Table 2: Number of instances with compiler errors in unconstrained generation (Vanilla), idealized syntax-only constraining (Syntax), and our proposed type constraining (Types). On average, type constraining reduces compiler errors by 75.3% in synthesis, compared to only 9.0% through Syntax.

	Synthesis			Translation			Repair		
Model	Vanilla	Syntax	Types	Vanilla	Syntax	Types	Vanilla	Syntax	Types
Gemma 2 2B	103	92$_{\downarrow 10.7\%}$	**44**$_{\downarrow 57.3\%}$	177	149$_{\downarrow 15.8\%}$	**80**$_{\downarrow 54.8\%}$	194	181$_{\downarrow 6.7\%}$	**103**$_{\downarrow 46.9\%}$
Gemma 2 9B	45	41$_{\downarrow 8.9\%}$	**13**$_{\downarrow 71.1\%}$	75	63$_{\downarrow 16.0\%}$	**16**$_{\downarrow 78.7\%}$	113	108$_{\downarrow 4.4\%}$	**52**$_{\downarrow 54.0\%}$
Gemma 2 27B	15	13$_{\downarrow 13.3\%}$	**2**$_{\downarrow 86.7\%}$	20	20$_{\downarrow 0.0\%}$	**3**$_{\downarrow 85.0\%}$	45	40$_{\downarrow 11.1\%}$	**22**$_{\downarrow 51.1\%}$
DS Coder 33B	26	25$_{\downarrow 3.8\%}$	**5**$_{\downarrow 80.8\%}$	18	17$_{\downarrow 5.6\%}$	**7**$_{\downarrow 61.1\%}$	36	36$_{\downarrow 0.0\%}$	**15**$_{\downarrow 58.3\%}$
CodeLlama 34B	86	71$_{\downarrow 17.4\%}$	**28**$_{\downarrow 67.4\%}$	158	124$_{\downarrow 21.5\%}$	**59**$_{\downarrow 62.7\%}$	153	142$_{\downarrow 7.2\%}$	**48**$_{\downarrow 68.6\%}$
Qwen2.5 32B	17	17$_{\downarrow 0.0\%}$	**2**$_{\downarrow 88.2\%}$	24	21$_{\downarrow 12.5\%}$	**5**$_{\downarrow 79.2\%}$	36	34$_{\downarrow 5.6\%}$	**13**$_{\downarrow 63.9\%}$

Table 3: pass@1 of unconstrained generation (Vanilla) and type constraining (Types). The benefit of our type-constraining transfers from reduced compilation errors to improved functional correctness.

	Synthesis		Translation		Repair	
Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
Gemma 2 2B	29.1	**30.2**	50.2	**53.9**	11.6	**20.9**
Gemma 2 9B	56.6	**58.3**	73.7	**78.3**	24.0	**34.9**
Gemma 2 27B	69.5	**71.2**	86.6	**87.7**	38.4	**41.1**
DS Coder 33B	68.9	**71.1**	88.7	**90.1**	47.6	**50.7**
CodeLlama 34B	41.0	**43.4**	58.6	**63.5**	17.5	**27.4**
Qwen2.5 32B	79.6	**81.8**	92.1	**93.9**	65.4	**71.2**

Table 4: Median time per synthesis instance in seconds spent by Types and its relative increase to Vanilla.

Model	Overhead
Gemma 2 2B	6.7$_{\uparrow 38.3\%}$
Gemma 2 9B	8.3$_{\uparrow 29.2\%}$
Gemma 2 27B	11.7$_{\uparrow 19.9\%}$
DS Coder 33B	11.5$_{\uparrow 36.2\%}$
CodeLlama 34B	7.6$_{\uparrow 40.8\%}$
Qwen2.5 32B	7.3$_{\uparrow 39.6\%}$

The tasks are sourced from the TypeScript translated HumanEval (Chen et al., 2021) subset of MultiPL-E (Cassano et al., 2023), containing 159 instances. To obtain more comprehensive results, we evaluate 4 times with different seeds and aggregate the outcomes. For Repair, we collect non-compiling programs from unconstrained synthesis for all models, resulting in 292 instances.

Models and Methods We use 6 different open-weight LLMs, covering 3 LLMs of varying parameter sizes from the same model family and 4 models of a similar size from different model families: the Gemma 2 model family with 2B/9B/27B parameters (Team et al., 2024), DeepSeek Coder 33B (Guo et al., 2024), CodeLlama 34B (Rozière et al., 2023), and Qwen2.5 32B (Yang et al., 2024). We always choose the instruction-tuned variants to ensure adherence to the natural language task.

Unconstrained LLM sampling is reported as *Vanilla*. We measure the upper bound improvement of syntactic constraining (*Syntax*) where all Syntax errors in Vanilla instances are considered resolved. We separately sample using type-constrained decoding (*Types*) based on §2. We emulate type constraining with the entire TypeScript feature set. Concretely, if a sample compiles correctly without any constraining, we report it as-is. Otherwise, we report the result of the constrained sample.

Metrics We compute two main metrics to assess the effectiveness of the compared methods. First, we determine *compiler errors*, any case in which the TypeScript compiler (Microsoft, 2024) reports an issue. To measure *functional correctness*, we leverage the pass@1 metric (Chen et al., 2021), measuring the percentage of generations that pass provided unit tests given only one trial.

3.1 Results on Compilation and Functional Correctness

Reduction of Compilation Errors In Tab. 5, we present the number of compilation errors. For synthesis and translation, Syntax could resolve 9.0% and 11.9% of the non-compiling instances. In contrast, Types reduces compilation errors by 75.3% and 70.2%. We observe that all models and tasks benefit similarly from our constraining, with a minimum error reduction of 54.8%.

For Repair, we find that many models struggle to correctly localize and resolve compilation errors, with Gemma 2 2B for example repairing only 33.5% of the non-compiling instances in Vanilla. This is substantially increased to 56.4% in Types. On average, using type-constrained sampling, 57.1% more compilation errors are resolved than using vanilla LLM decoding.

Improving Functional Correctness Programs that do not compile are always considered functionally incorrect. With our type constraining, non-compilable generations can be turned into well-formed ones. In Tab. 6, we experimentally show that type constraining universally improves functional correctness. Employing type constraining improves LLMs' pass@1 rate, achieving an increase by 3.5%, 4.5% in synthesis and translation respectively, and 34.1% in repair tasks.

Overhead of Type Constraining For an application of our method in practice, the effective runtime increase due to constrained decoding is highly relevant. To assess it, we measure the runtime per synthesis instance. We report the median runtime per instance for Types and its relative increase to Vanilla in Tab. 7. On average over the evaluated models, we observe a relative increase of 34.0%. We note our implementation is not optimized or systems-oriented such as Dong et al. (2024).

MBPP, Case Study and Discussion in Appendix We provide results on MBPP (Austin et al., 2021) in App. A, a case study of our constraints in App. B and a discussion of our work in App. C.

4 Related Work

Code Language Models Recently, LLMs have gained traction for diverse coding tasks (Jiang et al., 2024). LLMs are typically trained on trillion token datasets and have billions of parameters, both factors contributing to improved performance in code-related benchmarks (Rozière et al., 2023; Team et al., 2024; Guo et al., 2024). Meanwhile, even SOTA LLMs are well known to frequently make mistakes (Rawte et al., 2023; Huang et al., 2023), including code errors, as we showed.

Improving Language Model Accuracy Apart from constrained decoding, three primary approaches have been proposed to enhance the accuracy of language models on code tasks: fine-tuning, retrieval augmentation (RAG), and compiler or execution feedback. Fine-tuning adapts the model weights based on specifically collected training data (Tsai et al., 2024; Weyssow et al., 2023). RAG provides additional context based on databases or related code snippets (Bassamzadeh and Methani, 2024; Poesia et al., 2022). Feedback is only available after completed generation and requires resampling (Jana et al., 2024; Deligiannis et al., 2025; Wei et al., 2023). Constrained decoding is orthogonal to these, and, as indicated by Poesia et al. (2022) and our experimental results, combining constrained decoding with RAG or compiler feedback additionally improves model performance.

Constrained Decoding Prior work on constrained decoding failed to achieve strong results due to its limitation to syntactic language features. Constraining to context-free languages has been explored extensively (Beurer-Kellner et al., 2024; Poesia et al., 2022; Beurer-Kellner et al., 2023; Willard and Louf, 2023). Simple context-sensitive syntactic features, such as the space indentation in Python have been implemented (Melcer et al., 2024; Ugare et al., 2024). However, the rarity of syntax errors significantly reduces the potential of leveraging syntax for code correctness

Type Systems for Code Synthesis Prior work that leveraged type systems for code synthesis was confined to specialized settings and unable to constrain general, complex program generation. Poesia et al. (2022) used column names to guide SQL query synthesis. Gvero et al. (2013) employed a search on the type graph for function call completion. Agrawal et al. (2023) leverage language-server-generated type annotations for object member accesses. Type constraints have also been used to direct code synthesis in specialized search procedures (Wei et al., 2023; Fiala et al., 2023; Perelman et al., 2012). However, these methods are not compatible with LLM-based code generation.

5 Conclusion

In this work, we explored how type systems in programming languages can be used to guide language models during decoding. We design and implement prefix automata to perform type constraining and implement it for TypeScript. We evaluate its impact on code synthesis, translation, and repair and observe significant reduction of compilation and functional errors.

Acknowledgements

We would like to thank the anonymous reviewers for their in-depth and constructive feedback, and the artifact reviewers for their feedback on our artifact accessibility.

References

Lakshya Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *NeurIPS*. https://openreview.net/forum?id=qPUbKxKvXq

Anthropic. [n. d.]. Claude 3 Model Card. https://assets.anthropic.com/m/61e7d27f8c8f5919/original/Claude-3-Model-Card.pdf Accessed: March 10, 2025.

Anthropic. 2025. JSON Mode. https://docs.anthropic.com/en/docs/build-with-claude/tool-use#json-mode Accessed: March 10, 2025.

Ken Arnold and James Gosling. 1996. *The Java Programming Language*.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, et al. 2021. Program Synthesis with Large Language Models. *arXiv Preprint* (2021). https://arxiv.org/abs/2108.07732

Nastaran Bassamzadeh and Chhaya Methani. 2024. A Comparative Study of DSL Code Generation: Fine-Tuning vs. Optimized Retrieval Augmentation. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2407.02742

Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *PLDI* (2023). https://doi.org/10.1145/3591300

Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. In *ICML*. https://openreview.net/forum?id=pXaEYzrFae

Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *NeurIPS*. https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, et al. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Trans. Software Eng.* (2023).

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv Preprint* (2021). https://arxiv.org/abs/2107.03374

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv Preprint* (2025). https://doi.org/10.48550/arXiv.2501.12948

Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, Rishi Poddar, and Aseem Rastogi. 2025. RustAssistant: Using LLMs to Fix Compilation Errors in Rust Code. In *ICSE*. https://www.microsoft.com/en-us/research/publication/rustassistant-using-llms-to-fix-compilation-errors-in-rust-code/

TypeScript Developers. [n. d.]. TypeScript: Documentation – More on Functions. https://www.typescriptlang.org/docs/handbook/2/functions.html#function-type-expressions Accessed: March 10, 2025.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2024. XGrammar: Flexible and Efficient Structured Generation Engine for Large Language Models. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2411.15100

Alan AA Donovan and Brian W Kernighan. 2015. *The Go programming language*.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2407.06153

Jonás Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. Leveraging Rust Types for Program Synthesis. *PLDI* (2023). https://doi.org/10.1145/3591278

Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. 2025. Enhancing Code Generation for Low-Resource Languages: No Silver Bullet. *arXiv Preprint* (2025). https://doi.org/10.48550/arXiv.2501.19085

GitHub. [n. d.]. https://github.com/features/copilot

GitHub. 2022. The top programming languages. https://octoverse.github.com/2022/top-programming-languages

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. *ArXiv Preprint* (2024). https://arxiv.org/abs/2407.21783

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2401.14196

Gusanidas. [n. d.]. Compilation Benchmark. https://github.com/Gusanidas/compilation-benchmark Accessed: March 10, 2025.

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *PLDI*. https://doi.org/10.1145/2491956.2462192

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *arXiv Preprint* (2023). https://doi.org/10.48550/arXiv.2311.05232

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. OpenAI o1 System Card. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2412.16720

Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. CoTran: An LLM-Based Code Translator Using Reinforcement Learning with Feedback from Compiler and Symbolic Execution. In *ECAI (Frontiers in Artificial Intelligence and Applications)*. https://doi.org/10.3233/FAIA240968

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2406.00515

Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. 2024. Survey on Code Generation for Low resource and Domain Specific Programming Languages. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2410.03981

Madnight. 2024. GitHut 2.0. https://madnight.github.io/githut/#/pull_requests/2024/1

Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* (2014).

Daniel Melcer, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. 2024. Constrained Decoding for Fill-in-the-Middle Code Language Models via Efficient Left and Right Quotienting of Context-Sensitive Grammars. (2024). https://arxiv.org/abs/2402.17988

Microsoft. 2024. TypeScript. https://github.com/microsoft/TypeScript. Accessed on November 9, 2024, commit #ef802b1.

John C. Mitchell. 1990. Type Systems for Programming Languages. In *Formal Models and Semantics*. https://www.sciencedirect.com/science/article/pii/B9780444880741500135

nop33. 2024. reduce initial value, wrong inferred initial value in reduce. https://github.com/microsoft/TypeScript/issues/59999.

OpenAI. 2023. GPT-4 Technical Report. *arXiv Preprint* (2023). https://doi.org/10.48550/arXiv.2303.08774

OpenAI. 2025. Structured Outputs. https://platform.openai.com/docs/guides/structured-outputs Accessed: March 10, 2025.

oxc project. 2024. oxc - The Javascript Oxidation Compiler. https://github.com/oxc-project/oxc.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024a. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *ICSE*. https://doi.org/10.1145/3597503.3639226

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024b. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *ICSE*. https://doi.org/10.1145/3597503.3639226

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *S&P*. https://doi.org/10.1109/SP46214.2022.9833571

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *PLDI*. https://doi.org/10.1145/2254064.2254098

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *ICLR*. https://openreview.net/forum?id=KmtVD97J43e

Vipula Rawte, Amit P. Sheth, and Amitava Das. 2023. A Survey of Hallucination in Large Foundation Models. *arXiv Preprint* (2023). https://doi.org/10.48550/arXiv.2309.05922

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv Preprint* (2023). https://doi.org/10.48550/arXiv.2308.12950

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *ACL*. https://doi.org/10.18653/v1/p16-1162

Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A. Seshia, and Koushik Sen. 2024. Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2412.14234

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving Open Language Models at a Practical Size. *arXiv Preprint* (2024). https://arxiv.org/abs/2408.00118

Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. 2024. Code Less, Align More: Efficient LLM Fine-tuning for Code Generation with Data Pruning. (2024). https://doi.org/10.48550/arXiv.2407.05040

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. SynCode: LLM Generation with Grammar Augmentation.

Pawel Urzyczyn. 1997. Inhabitation in Typed Lambda-Calculi (A Syntactic Approach). In *TLCA (Lecture Notes in Computer Science)*. https://doi.org/10.1007/3-540-62688-3_47

Various. 2024. TypeScript GitHub Issue 59863. https://github.com/microsoft/TypeScript/issues/59863.

Heidi Vella. 2024. Google turns to AI to write new code; Workforce reduced. https://aibusiness.com/data/google-turns-to-ai-to-write-new-code-workforce-reduced

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *ESEC/FSE*. https://doi.org/10.1145/3611643.3616271

Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari A. Sahraoui. 2023. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. *arXiv Preprint* (2023). https://doi.org/10.48550/arXiv.2308.10462

Brandon T. Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. *arXiv Preprint* (2023). https://doi.org/10.48550/arXiv.2307.09702

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2.5 Technical Report. *arXiv Preprint* (2024). https://doi.org/10.48550/arXiv.2412.15115

Table 5: Number of MBPP instances with compiler errors in unconstrained generation (Vanilla), idealized syntax-only constraining (Syntax), and our proposed type constraining (Types). Types constraining reduces compiler errors by 52.1% in synthesis, compared to only 4.9% through Syntax.

	Synthesis			Translation			Repair		
Model	Vanilla	Syntax	Types	Vanilla	Syntax	Types	Vanilla	Syntax	Types
Gemma 2 2B	67	$64_{\downarrow 4.5\%}$	$\mathbf{27}_{\downarrow 59.7\%}$	126	$111_{\downarrow 11.9\%}$	$\mathbf{79}_{\downarrow 37.3\%}$	194	$184_{\downarrow 5.2\%}$	$\mathbf{108}_{\downarrow 44.3\%}$
Gemma 2 9B	30	$29_{\downarrow 3.3\%}$	$\mathbf{10}_{\downarrow 66.7\%}$	67	$61_{\downarrow 9.0\%}$	$\mathbf{33}_{\downarrow 50.7\%}$	129	$124_{\downarrow 3.9\%}$	$\mathbf{63}_{\downarrow 51.2\%}$
Gemma 2 27B	20	$19_{\downarrow 5.0\%}$	$\mathbf{7}_{\downarrow 65.0\%}$	37	$36_{\downarrow 2.7\%}$	$\mathbf{22}_{\downarrow 40.5\%}$	71	$69_{\downarrow 2.8\%}$	$\mathbf{32}_{\downarrow 54.9\%}$
DS Coder 33B	32	$32_{\downarrow 0.0\%}$	$\mathbf{19}_{\downarrow 40.6\%}$	29	$27_{\downarrow 6.9\%}$	$\mathbf{13}_{\downarrow 55.2\%}$	90	$90_{\downarrow 0.0\%}$	$\mathbf{43}_{\downarrow 52.2\%}$
CodeLlama 34B	80	$71_{\downarrow 11.2\%}$	$\mathbf{41}_{\downarrow 48.8\%}$	126	$114_{\downarrow 9.5\%}$	$\mathbf{54}_{\downarrow 57.1\%}$	157	$148_{\downarrow 5.7\%}$	$\mathbf{76}_{\downarrow 51.6\%}$
Qwen2.5 32B	19	$18_{\downarrow 5.3\%}$	$\mathbf{13}_{\downarrow 31.6\%}$	22	$22_{\downarrow 0.0\%}$	$\mathbf{16}_{\downarrow 27.3\%}$	55	$52_{\downarrow 5.5\%}$	$\mathbf{29}_{\downarrow 47.3\%}$

Table 6: MBPP pass@1 of unconstrained generation (Vanilla) and type constraining (Types). We observe a similar transfer of improvements as in HumanEval.

	Synthesis		Translation		Repair	
Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
Gemma 2 2B	40.4	**42.4**	52.3	**56.0**	12.1	**22.6**
Gemma 2 9B	65.4	**67.4**	71.4	**75.8**	24.2	**31.9**
Gemma 2 27B	70.6	**72.1**	83.1	**84.4**	39.1	**45.2**
DS Coder 33B	65.4	**67.2**	85.9	**89.1**	35.1	**43.1**
CodeLlama 34B	42.2	**45.6**	55.7	**63.3**	15.7	**26.6**
Qwen2.5 32B	76.3	**76.6**	89.6	**90.4**	48.0	**54.0**

Table 7: Median time per synthesis instance in seconds spent by Types and its relative increase to Vanilla on MBPP.

Model	Overhead
Gemma 2 2B	$6.3_{\uparrow 35.4\%}$
Gemma 2 9B	$9.5_{\uparrow 46.8\%}$
Gemma 2 27B	$11.7_{\uparrow 32.8\%}$
DS Coder 33B	$9.4_{\uparrow 59.5\%}$
CodeLlama 34B	$7.0_{\uparrow 37.6\%}$
Qwen2.5 32B	$4.9_{\uparrow 54.8\%}$

A Experimental Evaluation on MBPP

We outline our experiment on MBPP (Austin et al., 2021), mirroring the main experiment on HumanEval, with the same tasks, methods, models and metrics as in §3.

Dataset The instances are based on TypeScript-translated MBPP tasks (Austin et al., 2021), contained in the MultiPL-E dataset (Cassano et al., 2023), with 384 instances each. Contrasting our main experiments, in MBPP, we generate each sample once. For Repair, we collect all non-compiling programs from the unconstrained synthesis task for all models, resulting in 248 instances.

A.1 Results on Compilation and Functional Correctness

Reduction of Compilation Errors In Tab. 5, we present the number of compilation errors. For synthesis and translation, Syntax could improve 4.9% and 6.7% of the non-compiling instances respectively. In contrast, Types reduces compilation errors by 52.1% and 44.7%. We observe that models across all sizes and families benefit similarly from our constraining, with a minimum error reduction of 27.3%.

For Repair, we again find that many models struggle to correctly localize and resolve compilation errors, with Gemma 2 2B for example repairing only 31.6% of the non-compiling instances in Vanilla. This is substantially increased to 58.5% in Types. On average, using type-constrained sampling, 50.3% more compilation errors are resolved than using vanilla LLM decoding.

Improving Functional Correctness Programs that do not compile are always considered functionally incorrect. With our type constraining, non-compilable generations can be turned into well-formed ones. In Tab. 6, we experimentally show that type constraining universally improves functional correctness. Employing type constraining improves LLMs' pass@1 rate, achieving an increase by 3.6%, 5.5% in synthesis and translation respectively, and 39.8% in repair tasks.

Overhead of Type Constraining For an application of our method in practice, the effective runtime increase due to constrained decoding is highly relevant. To assess it, we measure the runtime per synthesis instance. We report the median runtime per instance for Types and its relative increase to Vanilla in Tab. 7. On average over the evaluated models, we observe a relative increase of 44.5%. We note our implementation is not optimized or systems-oriented such as Dong et al. (2024).

Number of Sample-and-Check Loop Iterations To provide an in-depth analysis of the overhead of our type constraining method, we measure the number of iterations spent by the sample-and-check loop to find an admissible token. The results are provided in Fig. 3. We observe that the number of loop iterations follows a long-tail distribution. For 99.5% of cases, only one loop iteration is needed. This number is similar for stronger models, with Gemma 2 9B and 27B requiring one iteration in 99.2% and 99.7% of cases, respectively. This means that, in most instances, LLMs can generate a valid token on the first attempt, which is then verified by the completion engine. In cases where more than one iteration is needed, the completion engine intervenes to guide the selection of valid tokens. These interventions help resolve errors in many instances in our benchmarks, providing significant benefit, as discussed in App. A.1.

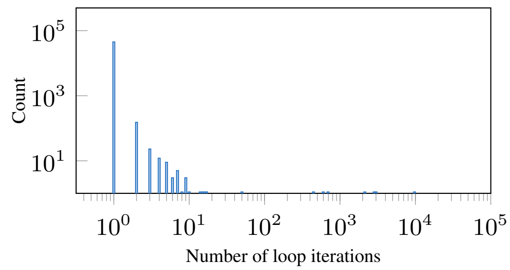

Figure 3: Histogram on the number of iterations consumed by the sample-and-check loop at L. 4 of Alg. 1 to find a valid token, measured with Gemma 2 2B for MBPP synthesis.

Prior work (Ugare et al., 2024; Beurer-Kellner et al., 2024; Poesia et al., 2022) implemented constrained decoding differently than Alg. 1. Instead of running the sample-and-check loop, they execute the completion engine for all tokens in the LLM's vocabulary, mask out all invalid tokens, and sample once from the remaining valid tokens based on their normalized likelihoods. This implementation is less efficient than ours, especially when calling the completion engine is costly. Based on the long-tail distribution depicted in Fig. 3, our implementation requires only a single invocation to the completion engine for most cases, whereas their implementation calls it for every token in the LLM's vocabulary for all cases. In our experiments, their implementation results in timeouts for all benchmark instances, with a timeout of 5 minutes per instance. This justifies our choice of using the sample-and-check loop.

B Case Study

For a qualitative evaluation, we manually inspect instances where unconstrained decoding fails and our type constraining approach successfully corrects errors. We find that our technique effectively amends various types of compilation errors. We showcase three such examples in Fig. 4 (shortened for brevity), with full outputs provided in App. H.

Wrong Parameter Count In Fig. 4a , the task is to find all words in a string that contain a number of consonants. When Gemma 2 2B attempts to solve this problem, translating from the Python version to TypeScript, it calls `split` without arguments. This is allowed in Python, but not in TypeScript, where a single `string` argument is required. Type constraining correctly detects this and guides the generation to sample a space character as parameter.

Missing Return Statement In Fig. 4b, to complete function `largest_divisor`, the model must compute a straightforward divisor loop. DeepSeek Coder 33B Instruct (Guo et al., 2024) implements a correct loop, but does not guarantee returning a value in every execution path. When the return statement in the loop is never executed, e.g., for negative inputs, the function thus returns `undefined`, violating the type rules. Our method detects this issue and forces the generation of another statement in the function body, resulting in a correct fallback return statement.

Incorrect Type Inference This example is shown in Fig. 4c. The task is to insert a delimiter between every element in an array. Gemma 2 9B solves this using the `reduce` function. This generic function accepts two arguments; the first argument is a callback function that is called consecutively for every element in the array and accumulates a result, the second argument is an initial value for the callback function. The type of the accumulator of the callback function is derived implicitly from the type of the second argument. In the given example, the second argument is an empty array,

(a) Vanilla

```
// find all words with consonants
function select_words(s: string): string[] {
  for (const word of s.split()
```

(a) Types

```
// find all words with consonants
function select_words(s: string): string[] {
  for (const word of s.split(' ')
```

(b) Vanilla

```
function largest_divisor(n: number): number {
  for(let i = n - 1; i > 0; i--) {
      if(n % i === 0) return i;
  }
  // No return
}
```

(b) Types

```
function largest_divisor(n: number): number {
  for(let i = n - 1; i > 0; i--) {
      if(n % i === 0) return i;
  }
  return 1;
}
```

(c) Vanilla

```
// insert a delimiter between every element
  return numbs.reduce(
    (acc, curr, index) => {
        acc.push(curr);
        return acc;
  }, []);
```

(c) Types

```
// insert a delimiter between every element
  return numbs.reduce(
    (acc: number[], num, index) => {
        acc.push(curr);
        return acc;
  }, []);
```

Figure 4: Three examples illustrating for the effect of type-constrained sampling. Left are unconstrained generations with problematic tokens highlighted in red, and right are type-constrained results with corrected tokens highlighted in green, adapted for clarity. In (a), Gemma 2 2B attempts to call `split`, missing required arguments. In (b), DeepSeek Coder 33B attempts to complete a function without a guaranteed return. The issue is resolved by forcing generation of another statement after the main loop. In (c), Gemma 2 9B calls `reduce` with an anonymous function without type annotation. This leads to an incorrect type inference for the first parameter. The issue is solved by guiding the model to add type annotation.

which should collect the new outputs. However, TypeScript infers a special type `never[]` for the empty array, disallowing pushing the `number` array elements into the array. As a result, the program fails to compile. This inference is a well-known problem and a limitation of the TypeScript compiler that often confuses even expert developers (Various, 2024; nop33, 2024). Our method resolves the issue by enforcing adequate type annotation on the first argument of the callback function.

C Discussion

Our general type constraining approach, backed by strong experimental results, opens exciting avenues for future research, which we discuss below.

Remaining Compiler Errors We observe that, even though constrained decoding guarantees a valid result upon termination, a considerable amount of compilation errors remain due to non-termination within the given token or time limit. We find this to be caused by generation loops, entered when generation is forced to comply with a constraint and the LLM is unable to recover. An example is depicted in Fig. 5, where CodeLlama 34B tries to access the invalid member `sort` on an expression of type `number`. Future work may add additional constraints to force stopping such unconstructive loops and steer the model more strictly, e.g., by limiting the complexity of generated expressions.

Implementation Effort Developing a completion engine for a target programming language currently requires manual efforts. However, we expect that the involved effort to adopt our method to other languages will be reduced significantly, as many features transfer from our implementation for L_B and TypeScript. Moreover, we believe that, due to the huge impact of LLMs on code generation, the effort will pay off. Future programming language developers may consider generally writing their compilers as an incremental completion engine, which additionally enables automatic adoption for constrained code generation, besides conventional grammar parsing and type checking.

Broader Application to More Complex Tasks and Stronger LLMs Stronger LLMs, such as the latest OpenAI models (Jaech et al., 2024), may make fewer typing errors on the HumanEval and MBPP datasets. Our evaluation results in Tab. 5 also demonstrate that compilation errors decrease with increasing model size for the Gemma family. However, recent findings showed that currently, even the strongest LLMs struggle with generating compilable code for more complex coding tasks, stricter typing rules, and low-resource languages (e.g., newly invented DSLs). Gu-

```
function sort_third(l: number[], r: number[]): number[] {
    for (let i = 0; i < l.length; i++) {
        r.push(l[i] .toString ().slice(0, 3).concat(l[i].toString().slice(3).split('')
        .reverse().join('')).split('').reverse().join('').toString() + l[i] ...
```

Figure 5: Compilations errors remain when the model does not terminate after a corrected token. In this example for synthesis on the HumanEval task #33, CodeLlama 34B is steered away from accessing non-existing member `.sort` and instead accesses `.toString`.

sanidas ([n. d.]) evaluated various state-of-the-art LLMs on difficult code synthesis tasks in Rust, reporting compilation error rates of 18% for OpenAI o1-mini (Jaech et al., 2024), 39% for DeepSeek R1 (DeepSeek-AI et al., 2025) and 27% for Anthropic's Claude 3.5 Sonnet (Anthropic, [n. d.]). For OCaml and Haskell, which are sparsely represented in LLMs' training data, the error rate is even higher at $40\% - 60\%$ for all models, matching a trend of worse performance on low-resource languages (Joel et al., 2024; Giagnorio et al., 2025). Pan et al. (2024b) compiled a large dataset of code translation and found 44.3% of GPT-4-generated code to contain compilation errors. Similarly, Shetty et al. (2024) report around 25% compilation errors for C-to-Rust translation using OpenAI o1 models. Our type constraining approach is broadly applicable to all these scenarios. Future work can consider extending our approach to address these challenges.

Constrained decoding in general requires access to the next-token probability distributions produced by LLMs. Currently, commercially available black-box LLM APIs only return sampled tokens and do not offer complete next-token distributions. A possible solution is to integrate our method into the backend of model providers, as was recently implemented for guaranteeing adherence to JSON Schemas (Anthropic, 2025; OpenAI, 2025).

D A Simply Typed Language

We define a simply typed, Turing-complete language, L_B. Its grammar and type system are generic, resembling the principles found in popular statically typed languages, such as TypeScript, Java, and Go. However, there may be a slight bias towards TypeScript, as our implementation is based on it.

Syntax The syntax of L_B is shown in Fig. 2. The language includes expressions, type-annotated variable and function definitions, and control flows. Overall, it is based on a core subset of TypeScript (Bierman et al., 2014) but can be adapted for other statically typed languages. Similar to Bierman et al. (2014), we represent Kleene-Star repetitions using an overline, e.g., $\overline{s}$ represents a sequence of statements s, and adhere to the TypeScript documentation to annotate parameter types in function signatures with argument names (Developers, [n. d.]). We make a distinction between base and extension expressions. The latter applies operators to previous expressions, leading to more complex expressions. This differentiation is useful later in §2.4 for constructing the prefix automaton for parsing expressions.

Expression Typing Rules The typing rules for L_B's expressions are detailed in Fig. 6. These rules form a subset of safeFTS, a type-safe portion of TypeScript described by Bierman et al. (2014), allowing us to leverage their soundness results. The type rules for L_B use the standard concept of a *type environment*, denoted as Γ, which is a collection of pairs $(x : T)$ of identifiers x and types T. We write $\Gamma \vdash e : T$ if the expression e has type T in the type environment Γ. An expression e is considered valid if its type can be derived by applying the given typing rules.

Literals are evaluated to their respective types ($\text{LIT} - \{\text{NUM}, \text{STR}, \text{BOOL}\}$). Identifiers x are evaluated based on the corresponding type in the type environment (IDENT). Anonymous functions are typed according to their annotated parameter types, with the return type determined by the returned expression (ANON). Grouping preserves the type of the inner expression (GROUP).

Binary operators have predefined signatures $S_1 \odot S_2 : T$, such as `number + number : number` for addition and T = $T : T$ for assignments. These signatures must be satisfied in well-typed expressions (OP). Function calls require parameters to match the function signature (CALL). The type of member accesses $e.n$ is determined using an auxiliary function $\text{LOOKUP}(S, n)$, which fetches the type of member n for type S. An instantiation of LOOKUP for TypeScript is provided by Bierman et al. (2014).

Statements and Type Environments The typing rules for statements are presented in Fig. 7. Type environments are modified by statements, in particular variable declarations and function definitions. We use the notation $\Gamma_1 \vdash s \rightarrowtail \Gamma_2$ to indicate that after executing statement s in type environment Γ_1, the new environment is Γ_2.

Variable declarations introduce the identifier with declared type into the type environment, provided the identifier is not already defined (DECL). The type environment defines the context to evaluate expressions (EXPR) and return statements (RET). Return statements are only well-typed inside function bodies. The statements inside statement blocks and if-then-else statements must maintain valid type environments, though they do not have an external effect (BLOCK, ITE). This also applies to function definitions; however, the defined function is finally added to the external type environment (FUN). Lastly, empty statements do not alter the type environment (NOP), while statement sequences propagate the type environment along the execution (SEQ).

Return Types The rules for checking return types are presented in Fig. 8. Firstly, return statements must contain expressions matching the function's declared return type. Secondly, such an expression must be returned on every execution path. We use the notation $\Gamma \vdash \overline{s} : R$ to indicate the sequence of statements $\overline{s}$ ensures a return value of type R.

For variable declarations and expression statements, the return type of the subsequent statements is considered (R-DECL, R-EXPR). The return type of a return statement directly corresponds to the type of the returned expression (R-RET). For statement blocks, the return type is decided by either the block itself or the subsequent statements (R-BLOCK-SELF, R-BLOCK-NEXT). In function definitions, the return type is determined by the type of the subsequent statements, similar to expression statements. It is additionally required that the function body returns a type matching the declared return type (R-FUN). For if-then-else statements, both branches must return the same type (R-ITE-SELF), or the return type is determined by the following statements (R-ITE-NEXT).

Language Definition In summary, a program s is in language L_B if both (i) s conforms to the grammar in Fig. 2 and (ii) s is well-typed according to the typing rules in Figs. 6–8.

E Detailed Prefix Automaton Definitions

In this section, we provide more detailed definitions and analysis of the various automata for L_B.

E.1 Base Automata

We now provide detailed definitions for the base prefix automata introduced at the end of §2.2: union, concatenation, Kleene-Star, and terminal.

Union For the union $A_X \cup A_Y$, we define the resulting sets of initial states and accepting states as $I := I_X \cup I_Y$ and $F := F_X \cup F_Y$, respectively. The transition function is defined as follows:

$$\delta(q, c) := \begin{cases} \delta_X(q, c) & \text{if } q \in Q_X \\ \delta_Y(q, c) & \text{if } q \in Q_Y \end{cases}$$

To show that the language parsed by this automaton is indeed the union $L(A_X \cup A_Y) = L(A_X) \cup L_(A_Y)$, we employ a short helper lemma, which can be shown inductively.

Lemma 6. *The set of the reachable states from a set of states* $\mathbf{q}$ *is equal to the union of reachable states from each state in* $\mathbf{q}$*, i.e.* $\gamma(\mathbf{q}, s) = \bigcup_{q \in \mathbf{q}} \gamma(q, s)$.

Since the states are distinct and we merely combine the transition functions of both automata, using the lemma, we can quickly see that the language parsed is indeed the union. Moreover, if both A_X and A_Y are prefix automata, this also holds for $A_X \cup A_Y$.

$$[\textsc{Lit-Num}]\ \frac{}{\Gamma \vdash \texttt{\textbackslash d+} : \texttt{number}} \qquad [\textsc{Lit-Str}]\ \frac{}{\Gamma \vdash \texttt{"\textbackslash w*"} : \texttt{string}} \qquad [\textsc{Lit-Bool}]\ \frac{}{\Gamma \vdash \texttt{true}, \texttt{false} : \texttt{boolean}}$$

$$[\textsc{Ident}]\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad [\textsc{Anon}]\ \frac{\Gamma \cup \overline{p} \vdash e : T}{\Gamma \vdash \texttt{(}\overline{p}\texttt{)} \texttt{ => } e : (\overline{p}) \texttt{ => } T} \qquad [\textsc{Call}]\ \frac{\Gamma \vdash f : (\overline{x} : \overline{S}) \texttt{ => } T \quad \Gamma \vdash \overline{e} : \overline{S}}{\Gamma \vdash f\texttt{(}\overline{e}\texttt{)} : T}$$

$$[\textsc{Group}]\ \frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{(}e\texttt{)} : T} \qquad [\textsc{Op}]\ \frac{\Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2 \quad S_1 \odot S_2 : T}{\Gamma \vdash e_1 \odot e_2 : T} \qquad [\textsc{Mem}]\ \frac{\Gamma \vdash e : S \quad \textsc{lookup}(S, n) = T}{\Gamma \vdash e.n : T}$$

Figure 6: Typing rules for L_B's expressions.

$$[\textsc{Decl}]\ \frac{x \notin \Gamma}{\Gamma \vdash \texttt{let}\ x : T\texttt{;} \rightarrowtail \Gamma \cup (x : T)} \qquad [\textsc{Expr}]\ \frac{\Gamma \vdash e : T}{\Gamma \vdash e\texttt{;} \rightarrowtail \Gamma} \qquad [\textsc{Ret}]\ \frac{\text{inside function body} \quad \Gamma \vdash e : T}{\Gamma \vdash \texttt{return}\ e\texttt{;} \rightarrowtail \Gamma}$$

$$[\textsc{Block}]\ \frac{\Gamma \vdash \overline{s_B} \rightarrowtail \Gamma_B}{\Gamma \vdash \texttt{\{}\ \overline{s_B}\ \texttt{\}} \rightarrowtail \Gamma} \qquad [\textsc{Fun}]\ \frac{x \notin \Gamma \qquad \Gamma \cup (x : (\overline{p}) \texttt{ => } T) \cup (\overline{p}) \vdash \overline{s_x} \rightarrowtail \Gamma_x}{\Gamma_1 \vdash \texttt{function}\ x\ \texttt{(}\overline{p}\texttt{)} : T\ \texttt{\{}\ \overline{s_x}\ \texttt{\}} \rightarrowtail \Gamma \cup (x : (\overline{p}) \texttt{ => } T)}$$

$$[\textsc{Ite}]\ \frac{\Gamma \vdash s_{if} \rightarrowtail \Gamma_{if} \quad \Gamma \vdash s_{else} \rightarrowtail \Gamma_{else}}{\Gamma \vdash \texttt{if (}e\texttt{)}\ s_{if}\ \texttt{else}\ s_{else} \rightarrowtail \Gamma} \qquad [\textsc{Nop}]\ \frac{}{\Gamma \vdash \bullet \rightarrowtail \Gamma} \qquad [\textsc{Seq}]\ \frac{\Gamma_1 \vdash \overline{s} \rightarrowtail \Gamma_2 \quad \Gamma_2 \vdash s \rightarrowtail \Gamma_3}{\Gamma_1 \vdash \overline{s}\ s \rightarrowtail \Gamma_3}$$

Figure 7: Type environment extension rules for sequences of statements in L_B.

$$[\textsc{R-Decl}]\ \frac{\Gamma \vdash \overline{s} : R}{\Gamma \vdash \texttt{let}\ x : T\texttt{;}\overline{s} : R} \qquad [\textsc{R-Expr}]\ \frac{\Gamma \vdash \overline{s} : R}{\Gamma \vdash e\texttt{;}\overline{s} : R} \qquad [\textsc{R-Ret}]\ \frac{\Gamma \vdash e : R}{\Gamma \vdash \texttt{return}\ e\texttt{;}\overline{s} : R}$$

$$[\textsc{R-Block-Self}]\ \frac{\Gamma \vdash \overline{s_B} : R \quad \Gamma \vdash \overline{s}}{\Gamma \vdash \texttt{\{}\ \overline{s_B}\ \texttt{\}}\ \overline{s} : R} \qquad [\textsc{R-Block-Next}]\ \frac{\Gamma \vdash \overline{s_B} \quad \Gamma \vdash \overline{s} : R}{\Gamma \vdash \texttt{\{}\ \overline{s_B}\ \texttt{\}}\ \overline{s} : R}$$

$$[\textsc{R-Fun}]\ \frac{\Gamma \cup (x : (\overline{p} \texttt{ => } R)) \vdash \overline{s} : R' \quad \Gamma \cup (x : (\overline{p}) \texttt{ => } R) \cup (\overline{p}) \vdash \overline{s_x} : R}{\Gamma \vdash \texttt{function}\ x\ \texttt{(}\overline{p}\texttt{)} : R\ \texttt{\{}\ \overline{s_x}\ \texttt{\}}\ \overline{s} : R'}$$

$$[\textsc{R-Ite-Self}]\ \frac{\Gamma \vdash s_{if} : R \quad \Gamma \vdash s_{else} : R}{\Gamma \vdash \texttt{if (}e\texttt{)}\ s_{if}\ \texttt{else}\ s_{else}\ \overline{s} : R} \qquad [\textsc{R-Ite-Next}]\ \frac{\Gamma \vdash \overline{s} : R}{\Gamma \vdash \texttt{if (}e\texttt{)}\ s_{if}\ \texttt{else}\ s_{else}\ \overline{s} : R}$$

Figure 8: L_B's typing rules for function returns.

Concatenation For the concatenation automaton $A_{XY} = A_X \circ A_Y$, we define $I := I_X$, $F := F_Y$, and the transition function as follows:

$$\delta_{XY}(q, c) := \begin{cases} \delta_X(q, c) & \text{if } q \in Q_X \backslash F_X \\ \delta_X(q, c) \cup \delta_Y(I_Y, c) & \text{if } q \in F_X \\ \delta_Y(q, c) & \text{if } q \in Q_Y \end{cases}$$

Informally, concatenation preserves the parsing behaviour of both A_X and A_Y in their respective states. When A_{XY} reaches an accepting state of A_X and receives another input character, it either remains in A_X or transitions to A_Y, as defined in the second case of δ_{XY}. Essentially, this maintains outgoing edges from accepting states in A_X while adding edges from these accepting states to initial states of A_Y.

It follows from a similar argument that $L(A_{XY}) = L(A_X) \circ L(A_Y)$, where $L(A_X) \circ L(A_Y)$ is defined as $\{s_X \circ s_Y \mid s_X \in L(A_X), s_Y \in L(A_Y)\}$. We first show $L(A_{XY}) \subseteq L(A_X) \circ L(A_Y)$. Due to (P1), we can always split any $s \in L(A_{XY})$ into s_X that extends from I_X to F_X and s_Y that extends from I_Y to F_Y. Then $s_X \in L(A_X)$ and $s_Y \in L(A_Y)$. For $L(A_X) \circ L(A_Y) \subseteq L(A_X \circ A_Y)$, we pick any $s_X \circ s_Y$ from $L(A_X) \circ L(A_Y)$ and parse it using A_{XY}. We observe that it will first traverse from I_X to F_X consuming s_X, and then transition through I_Y to F_Y by consuming s_Y.

Moreover A_{XY} is a prefix automaton, if A_X and A_Y are prefix automata and $L(A_Y) \neq \varnothing$. Since A_X is a prefix automaton, we can reach F_X from any state in Q_X. From F_X we additionally reach $I_Y \subseteq Q_Y$. Since A_Y is a prefix automaton, we can reach F_Y for any state in Q_Y. This construction is a prefix automaton only if $I_Y \neq \varnothing$, which, due to the prefix property, is equivalent to $L(A_Y) \neq \varnothing$.

Kleene-Star We define the Kleene-Star automaton $A_{\overline{X}}$ that parses indefinite repetitions of words accepted by X. First, we consider all initial states as final states, i.e., we ensure $I_X \subseteq F_{\overline{X}}$. Then we add transitions to the transition function δ_X from the final states F_X back to the initial states I_X.

$$\delta_{\overline{X}}(q_X, c) := \begin{cases} \delta_X(q_X, c) & \text{if } q \notin F_X \\ \delta_X(q_X, c) \cup \delta(I_X, c) & \text{if } q_X \in F_X \end{cases}$$

We can quickly see that $L(A_{\overline{X}}) = \{\overline{s} \mid s \in L(A_X)\}$, with the same argument as the concatenation automaton. Additionally, because the initial states are accepting, the empty word (zero repetitions) is in $L(A_{\overline{X}})$. We similarly see that this is prefix automaton if A_X is a prefix automaton. Note that here $L(A_X) \neq \varnothing$ is not required. This is because if $L(A_X) \neq \varnothing$, then $A_{\overline{X}} = A_X = A_\varnothing$, which is still a prefix automaton.

Terminals The terminal automaton $A_{\mathtt{S}}$ parses exactly the terminal $\mathtt{S}$. They accept the usual alphabet Σ and feature the states $Q := \{q_{\mathtt{s}} \mid \mathtt{s} \text{ is a suffix of } \mathtt{S}\}$, $F := \{q_\varepsilon\}$, $I := \{q_{\mathtt{S}}\}$. The transition function δ is defined as follows:

$$\delta(q_{\mathtt{s}}, c) := \begin{cases} \{q_{\mathtt{s'}}\} & \text{if } c \circ \mathtt{s'} = \mathtt{s} \\ \varnothing & \text{otherwise} \end{cases}$$

Clearly $A_{\mathtt{S}}$ is a prefix automaton. We can show inductively that for any $\mathtt{s}$: $\gamma(q_{\mathtt{s}}, \mathtt{s'}) = \{q_\varepsilon\} \iff \mathtt{s} = \mathtt{s'}$, and thus $L(A_{\mathtt{S}}) = \{\mathtt{S}\}$. With a simple modification, we introduce $A^W_{\mathtt{s}}$, where W denotes whitespace characters. The transition function is defined as $\delta(q^W_{\mathtt{s}}, c) := \{A^W_{\mathtt{s}}\}$ if $c \in W$; otherwise, $\delta(A^W_{c \circ \mathtt{s}}, t) := \{A^W_{\mathtt{s}}\}$. This allows arbitrary whitespaces before parsing $\mathtt{s}$. This is how we implement syntactic indifference to whitespace between terminals.

Notational Details In the following, we will implicitly assume that $\delta(q, c) = \varnothing$ if not explicitly defined otherwise, making notation more concise. For any state, we access the following information through dot notation or the special notation on the state, which we assume is passed to subsequent states through the transition function (unless otherwise stated). This information is alternatively passed through to entire automata in composite automata, e.g., in A_{XY} from A_X to A_Y.

- $q \downarrow T$: Type T to which state q is constrained (introduced in more detail later).
- $q \in F_X$: Whether state q is an accepting state of the automaton A_X.
- $q.\Gamma$: The type environment based on state q currently being parsed.

- $q.\text{LHS}$: The left-hand side expression of an extending expression represented by state q, i.e., when extending X with Y and currently parsing q_Y, then $q_Y.\text{LHS} = X$.
- $q.\text{TYP}$: The described type of the last coherent expression that this state belongs to. This is only defined for accepting states. Generally, we ensure that when some expression e was parsed, the corresponding state q_e has attribute $q_e.\text{TYP}$ such that $q_e.\Gamma \vdash e : q_e.\text{TYP}$.

When accessing the properties of A, we access the property of the current state of the automaton $q \in Q$, e.g., $A.\text{LHS} = q.\text{LHS}$. For parsed automata, the current state is the final, accepting state.

E.2 Expressions

Expressions are parsed using recursive automatons as introduced in §2.4. In this part of the appendix, we describe in more detail how information is passed between states, using the additional properties of states introduced in App. E.1.

In the case of expressions, the TYP attribute expresses the type of the expression parsed so far. In expression states q, we leverage the LHS to accurately determine $q.\text{TYP}$.

$$
\begin{aligned}
q_{\text{STR}}.\text{TYP} &:= \texttt{string} \\
q_{\text{NUM}}.\text{TYP} &:= \texttt{number} \\
q_{\text{BOOL}}.\text{TYP} &:= \texttt{boolean} \\
q_x.\text{TYP} &:= T \text{ where } q_x.\Gamma \vdash x : T \\
q_{(\overline{p})\texttt{=>}e}.\text{TYP} &:= \texttt{(}A_{\overline{p}}.\text{TYP}\texttt{)} \texttt{ => } A_e.\text{TYP} \\
q_{(e)}.\text{TYP} &:= A_e.\text{TYP} \\
q_{\odot e}.\text{TYP} &:= R, \text{ for } q_{\odot e}.\text{LHS}.\text{TYP} = S, A_e.\text{TYP} = T \text{ and } S \odot T : R \\
q_{(\overline{e})}.\text{TYP} &:= T, \text{ for } q_{(\overline{e})}.\text{LHS}.\text{TYP} = \texttt{(}\overline{p}\texttt{)} \texttt{ => } T \\
q_{.n}.\text{TYP} &:= T, \text{ for } \text{LOOKUP}(q_{.n}.\text{LHS}.\text{TYP}, n) = T
\end{aligned}
$$

Unrestricted Expressions The left-hand side of the currently parsed expression is used in the definition of automata for three extending expressions; arithmetic operators, function call, and member access. The arithmetic operator automaton constrains its states to those with valid operators, i.e.:

$$
A_{\odot e} := \bigcup_{\exists R: A_{\odot e}.\text{LHS}.\text{TYP} \odot T = R} A_{\odot}(\circ A_e \downarrow T)
$$

For function call, the automaton is only valid if the left-hand side is a function, and accepts only the valid signature.

$$
A_{(\overline{e})} := \begin{cases} A_{(} \circ (A_{\overline{e}} \downarrow A_{\overline{p}}.\text{TYP}) \circ A_{)} & \text{if } A_{(\overline{e})}.\text{LHS}.\text{TYP} = \texttt{(}\overline{p}\texttt{)} \texttt{ => } T \\ A_{\varnothing} & \text{otherwise} \end{cases}
$$

Finally, the member access automaton is a union of the automata that parses the attributes of the left-hand side expression. Or formally,

$$
A_{.n} := \bigcup_{\exists T: \text{LOOKUP}(A_{.n}.\text{LHS}.\text{TYP}, m) = T} A_{.\texttt{m}}.
$$

Type-Restricted Expressions The type restricted versions of the automata are all covered by the definition presented in §2.4. Note that the DERIVABLE function presented in Tab. 1 implicitly leverages access to $q_s.\text{LHS} = e$ when accessing the expression e in the extension expressions.

E.3 Pruning the Type Search

In this section, we present our heuristic for pruning the type search recursion from §2.4, i.e., our implementation of PRUNESEARCH at L. 6 of Alg. 2. The heuristic is based on the complexity and novelty of candidate types to explore.

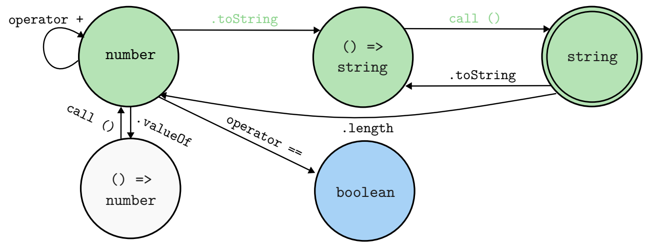

Figure 9: An example search through the graph for type reachability, starting from $T =$ `number` with the goal `string`, e.g., after parsing `let x : string; x = 1`. States and edges along the final path are marked in green and explored nodes in blue. The `() => number` node is not explored, as complex types are avoided by our heuristic. The node `() => string` is explored as it enables reaching new type `string`.

Based on the assumptions about the LOOKUP function and operators in §2.1, we observe a restriction in the reachable types by extensions: From any given type, we reach itself, result types of arithmetic operators via OP, return types through CALL, and member types through MEMBER. A higher-order type `() =>` T does not allow access to types not reachable from T. Consequently, we avoid exploring such higher-order types unless the target type is of higher order, or the higher-order type offers novel, yet unexplored types. For instance, in Fig. 9, the type `() => number` is not explored, because it is more complex than both the initial and goal types, `number` and `string`, and does not contain any unexplored type.

To formalize this understanding, we introduce the concepts about the *depth* and *root types* of a given type, denoted as DEPTH and ROOT, respectively. DEPTH measures the complexity of a type, specifically the order of a function, while ROOT returns all types of minimal depth (e.g., `string`, `number`, and `boolean`) that constitute a higher-order type. They are defined as follows:

$$\text{DEPTH}(T) := \begin{cases} \text{DEPTH}(S) + 1 & \text{if } T = (\overline{p}) \texttt{ => } S, \\ 0 & \text{otherwise.} \end{cases} \quad \text{ROOT}(T) := \begin{cases} \text{ROOT}(S) & \text{if } T = (\overline{p}) \texttt{ => } S, \\ \{T\} & \text{otherwise.} \end{cases}$$

We leverage DEPTH and ROOT to implement PRUNESEARCH(T, G, S) for a current type T, a goal type G, and a type S after an extension is applied on T. In general, if G is not directly accessible from T, it will also not be accessible from expressions with the same root types but greater depth, such as $() \Rightarrow T$. When G is of higher order, exploring up to the depth of G can be required, such as when $G =$ `() => (() => number)`. Based on these two ideas, we stop exploring S when $\text{DEPTH}(S) > \max(\text{DEPTH}(G), \text{DEPTH}(T))$.

Further, if a higher-depth function returns an unexplored type, we need to explore it. Sticking to the example in Fig. 9, type `number` has the member `toString` of type `() => string`. The type `string` can only be reached by exploring the member access at depth 1. On the contrary, we do not explore a higher-depth function if it does not introduce novel types other than explored. To achieve this, we adapt Alg. 2 to additionally define a set of root types R, which is initialized to an empty set and is updated by $R = R \cup \text{ROOT}(T)$. We do not explore S if $\text{ROOT}(S) \subseteq R$.

Taking the conjunction of the aforementioned two aspects, our pruning heuristic is implemented as $\text{PRUNESEARCH}(T, G, S) := \text{DEPTH}(S) > \max(\text{DEPTH}(T), \text{DEPTH}(S)) \wedge \text{ROOT}(S) \subseteq R$. The restrictions based on depth and root types are based on the results of the rigorously analyzed search over succinct types by Gvero et al. (2013). This provides a robust heuristic for exploring as many relevant inhabitable types as possible. However, due to the additional complexity introduced by the lookup function, we can not guarantee completeness and instead refer to the strong empirical results in our evaluation in §3 as evidence of the search's high coverage.

E.4 IMPLEMENTATION OF DERIVABLE

Recall that in Tab. 1, DERIVABLE for function expressions are defined as: $\text{DERIVABLE}(q_{(\overline{p}) \texttt{ => } e}) = \{(\overline{p}) \texttt{ => } T \mid \text{REACHABLE}(\text{DERIVABLE}(q_e), T)\}$. This involves constructing a type reachability graph and collecting all T types reachable from $\text{DERIVABLE}(q_e)$. However, this process is intractable because T can be of arbitrarily high-order, as such there are infinitely many T to explore. A similar issue exists for grouped expressions, as their DERIVABLE function is also defined to enumerate reachable types. We introduce two optimization heuristics to address this problem.

We first observe that DERIVABLE is always called within the context of an invocation of REACHABLE with target type G, e.g., $\text{REACHABLE}(\text{DERIVABLE}(q_{(\overline{p})\texttt{ => }e}), G)$ for function expressions. To compute $\text{DERIVABLE}(q_{(\overline{p})\texttt{ => }e})$, we enumerate all types present on the type graph represented by $\text{REACHABLE}(\text{DERIVABLE}(q_e), G)$, which is finite with due to application of the pruning heuristics in App. E.3. In other words, we bound the maximum complexity of considered types T using the pruning heuristic for reachability of target type G. This leads to a sound but potentially incomplete version of DERIVABLE. However, since the final goal is to reach G, this heuristic provides a practically useful set of all relevant derivable types.

Second, we observe that the resulting two-tiered call $\text{REACHABLE}(\text{DERIVABLE}(q_{(\overline{p})\texttt{ => }e}), G)$ can be integrated into a single call to further reduce the amount of explored types. Concretely, when discovering some type M in $\text{REACHABLE}(\text{DERIVABLE}(q_e), G)$, as per the previous heuristic, we allow transitioning directly to $\text{REACHABLE}(\texttt{(}\overline{p}\texttt{) => }M, G)$ to allow a depth-prioritizing exploration of the search graph. This allows us to efficiently discover a path to G if it exists.

E.5 STATEMENTS

We define the remaining automata to capture the complete language from §2.1. To correctly handle function return types, we pass on related information when entering function bodies:

- $q.R$: The expected return type of the current state q.
- $q.\text{RETURNED}$: Whether the currently parsed program block has returned in all branches.
- $q.\text{MUSTRETURN}$: Whether the currently parsed program block must return (i.e., If-Then-Else branches do not need to contain return statements even if a return type is expected of the surrounding code block).

The single statement automaton is another recursive definition, since some statements, e.g., If-Then-Else, can themselves contain statements. The statement automaton is defined recursively as $A_s :=$ $A_{\text{DECL}} \cup A_{\text{EXPR}} \cup A_{\text{RET}} \cup A_{\text{BLOCK}} \cup A_{\text{FUN}} \cup A_{\text{ITE}}$. The expression statement automaton and block automaton are simply defined as $A_{\text{EXPR}} := A_{e;}$ and $A_{\text{BLOCK}} := A_{\{\overline{s}\}}$. The declaration automaton $A_{\text{DECL}} := A_{\texttt{let } x:T;}$ captures variable names x using an automaton for non-existing identifiers, which works the same way as A_x except that it rejects terminals that match an existing variable. This automaton is a prefix automaton as well, since indefinite additional characters can be added to the variable name and there are only finitely many defined variables. The If-Then-Else automaton is defined using standard concatenation: $A_{\text{ITE}} := A_{\texttt{if(}e\texttt{) } s \texttt{ else } s}$. The statements automaton $A_{\overline{s}}$, based on the Kleene-Star automaton definition and the single statement automaton. Return statements are only non-empty when the expected return type is set, i.e. when parsing inside a function:

$$A_{\text{RET}} := \begin{cases} A_{\texttt{return}} \circ A_e \downarrow T & \text{if } A_{\text{RET}}.R = T \\ A_{\varnothing} & \text{otherwise} \end{cases}$$

For functions, the automaton is based on the standard concatenation $A_{\text{FUN}} = A_{\texttt{function } x \texttt{ (}\overline{p}\texttt{) : } T \texttt{ \{}\overline{s}\texttt{\}}}$. However, the transition function updates the states of the statement automata inside the function:

- $q.R = T$, i.e., the return type of these statements is set to the return type of the function. This value is propagated recursively to all sub-automata.
- $q.\text{MUSTRETURN} = \texttt{true}$, for the outermost statement block automaton. It is set to false for deeper nested statement blocks and as soon as a parsed statement X has $q_X.\text{RETURNED}$ set to true - i.e. one of the main body statements returned in every branch.
- $q.\text{RETURNED} = \texttt{false}$, per default in every statement, except a) in return automata, b) inside a multi-statement automaton where the previous statement has $\text{RETURNED} = \texttt{true}$ and c) in ITE-automata where both branching statements have $\text{RETURNED} = \texttt{true}$.

As long as a state q in a multi-statement automaton has $X.\text{RETURNED} = \texttt{false}$ and $q.\text{MUSTRETURN} = \texttt{true}$, it can not accept but instead forces the generation of another statement.

Since we can always express the requested type through literals and can always simply generate a return statement to fulfill this requirement, the prefix automaton property is not violated.

Tracking Type Environments Generally, we follow the typing rules in Fig. 7. Identifiers are passed on through all state transitions, matching the rule SEQ, where the type environment of consecutive statements needs to be compatible. However, in the cases of BLOCK, ITE and FUN, we discard the local type environment after parsing, matching the respective typing rules. In FUN additionally, the function signature and parameters are added into the type environment of the function body automaton.

F Extension to TypeScript

We extend our completion engine described in §2 to handle a core subset of modern TypeScript. In this section, we selectively discuss the implementation of interesting language features. We provide a complete list of supported features in Tab. 8 of App. G.

Constant Variable Declarations In addition to variable declaration using `let`, TypeScript supports constant declarations using `const`. This defines immutable identifiers. We thus additionally track mutability of each identifier in the type environment and disallow applying the assignment operator to immutable identifiers.

Arrays We add support for array type annotation, parsing array expressions, and reading from and assigning to array fields. In array expressions, we enforce that all array elements have the same type. Moreover, array types introduce another dimension of type nesting. Therefore we adapt the type reachability pruning heuristic to handle this additional dimension to ensure termination.

Loops TypeScript supports various loop constructs, including `for`, `while`, `do-while`, and `for...of` loops. These are implemented mostly as variations of the statement block parser. The `for...of` loop uniquely constrains the right-hand side of the `...of` operator to an array of any type. To adapt the type search, we introduce a generic array type •[], which matches any array type. For example, both types `number[]` and `string[]` match •[] in L. 2 of Alg. 2.

Additional Operators and Types We add several arithmetic and logic operators, such as modulo `%`, exact equality `===`, logical or `||`, and the ternary operator `?:`. To handle these operators, we add additional edges to the type search graph. Moreover, we add support for post- and prefix operators such as `--` and `++`, which are only valid extensions to mutable expressions.

Operator Precedence TypeScript defines an operator precedence, which determines the implicit grouping of expressions. For example `1 + 2.toString()` is parsed as `1 + (2.toString())`. We adapt our expression parsing algorithm in two places to handle operator precedences. First, in the expression automaton, we leverage the knowledge about previously parsed extensions to determine the implicit grouping and thus where the next operator is applied. For example, for state $q^{\texttt{1 + 2}}$, the member access extension $.n$ is applied to `2`, as opposed to `1 + 2`. Second, we adapt the type search in Alg. 2. Concretely, we ensure that only extensions that can be validly applied based on operator precedence are iterated over. For this, we track the operator precedence of previously parsed extensions and extensions considered during the traversal of the type graph and omit operators in L. 5 that violate operator precedence.

Global Identifiers and Imports In TypeScript, many identifiers are defined globally and available in any execution. These global identifiers are incorporated by initializing the type environment of the program automaton accordingly. Identifiers such as `Math` introduce additional types, which we additionally implement. We also model the import of the `crypto` library using `require`.

Polymorphic Built-In Members The TypeScript LOOKUP implementation defines a few polymorphic members for built-in types. For example, for array `x` of type `T[]`, `x.map(f)` takes a callback function `f` and returns a new array `[f(x[0]), f(x[1]), ...]`. If `f` has type `(T) => P`, the returned array has type `P[]`. Here `P` is a *type parameter*, which is instantiated by matching the type of the passed function to the type pattern.

We support such polymorphisms by adapting the type search. We track type patterns and enforce that type parameters are instantiated before the goal type is reached. We then continue the search from the instantiated version. In the `map` example, when searching completions of `x.map`, we first search for functions that instantiate the type parameter, and then continue the search from the instantiated type. When anonymous functions are generated as call parameters, we enforce that the function matches the searched type pattern.

Mandatory Type Annotation TypeScript is designed to be flexible, allowing many type annotations to be omitted when they can be automatically inferred. While this is generally beneficial, such as inferring types from initial values, it can lead to unexpected types when annotations are omitted, often confusing even experienced developers (nop33, 2024; Various, 2024). Moreover, in the context of LLM-based code generation, having more type annotations can provide valuable information for both the model and our type constraining algorithms. We have identified three situations where generated code often fails to compile without type annotations, prompting us to enforce them. First, we require annotations for all function parameters and return types. Second, all variable declarations must either have a type annotation or be initialized with a value. Third, we enforce type annotations for the first parameter of anonymous functions used as callbacks in the polymorphic built-in member `reduce`. These constraints trade-off practical correctness with theoretical language completeness.

G Details about Experimental Evaluation

In this section, we detail how executable code is extracted from the model responses and a slight modification to the decoding algorithm used, that increases throughput heuristically.

Implementation Details Our implementation is written in Python and contains 11249 lines of code. To ensure robust implementation, we built a large set of around four hundred unit tests and frequently compared the behaviors of our implementation with the official TypeScript compiler (Microsoft, 2024).

We have two main external dependencies. To implement the regular-expression-based literal automata, we leverage the `regex` library,[3] as it allows checking if the current string can be completed to match a regular expression. To implement LLM inference, we leverage the `transformers` library.[4] We provide an exhaustive list of supported features and unsupported features of the TypeScript language in our final implementation in Tab. 8.

Hyperparameters We run the models on A100 NVidia GPUs with 80 GB of VRAM. We use CUDA 12.4 and the Hugging Face library. We set the sampling temperature to 1. We set seeds to 0 to 4 on the four HumanEval runs and 0 on the one MBPP run respectively. We limit the completions to 1000 tokens and time out after 300 seconds. We compute syntactic correctness using the Oxidation toolchain (oxc project, 2024) as the TypeScript Compiler does not clearly distinguish between syntactic and semantic errors.

Excluded MBPP Instances We discovered that a number of TypeScript translations contained invalidly generated nested tuples. After reporting them to the developers, they have been resolved in the latest version of MBPP and we include them in our evaluation. Still, we find that the TypeScript translation of a number of MBPP instances contains too broad type annotation, annotating elements as `any` or array of `any`. We therefore exclude the following 6 instances from the evaluation:

- `mbpp_405_check_tuplex`
- `mbpp_563_extract_values`
- `mbpp_580_extract_even`
- `mbpp_612_merge`

[3] https://pypi.org/project/regex/

[4] https://huggingface.co/docs/transformers

Table 8: Supported and Missing TypeScript Features

Supported TypeScript Features	**Examples**
Expressions, Statements, Function Declarations	(L_B as introduced in §2)
Additional Literals: BigInt, Regex, Template Strings	10n, /\d*/, ‘hello ${user}‘
Additional Types: void, null, undefined	void, undefined, null
Index Signature Types and Literals	let x: {[y: number]: string} = 1: "hi";
Anonymous Functions	function (): bool {return true}
Lambda Functions with and without Function Bodies	x => {return y}, x => y
Ternary and LogicOperators	? :, ||, &&
Arithmetic and Boolean Operations	+, -, **, &, !
Assigning Pre-and Postfix Operators	++, --
Arrays	[1, 2, 3]
Access and Assignment to Computed Members	x[10] = y[i];
Constructors and "new" Calls	let x = new Number(1);
Calls with Optional and Rest Parameters	function foo(x?: number, y...: string)
Sets and Maps	Map<string, number>()
Parameterized Constructor Calls	new Set<string>()
Tuples	let x : [int, string] = [1, "hello"];
Optional Chaining	x.get("hi")?.get("world")
Spread Operator	[...xs]
Type Assertions	"hello" as any
For Loops	for(int x = 0; i < 10; i++)
For Of Loops	for(x of xs)
For Of Loops with Tuple Destructuring	for([x, y] of xys)
Do-While and While Loops	while (true) {...}
Typed and Untyped Variable Declarations	let x: number = 1; let y = 100;
Comments, Multiline Comments	// Comment
Returning without Expressions	return;
Try-Catch Statements with a Fixed Exception Type	try { ... } catch (e) { ... }
Throw Statements	throw new Error("...")
Importing the crypto Library	require("crypto")
Global Scope Objects	Math, parseInt
Automatic Semicolon Insertion	

Missing Features	**Examples**
General Library Imports	require("example")
Use of Functions Before Declaration	
For In Loops	for(x in y)
Type Declaration	
User-Defined Classes	
Declaration and Parameterized Call of General Parameterized Functions	
Destructuring Assignment	[x, y] = z
Uninitialized, Unannotated Variable Declarations	let x;
Return Type Inference	
Literal Types	
Enums	
Symbols	

- `mbpp_725_extract_quotation`
- `mbpp_791_remove_nested`

Complete Prompts We provide the complete prompts to the language models for the settings synthesis, translation, and repair. The prompts are templates, instantiated with the prompt from the Multipl-E dataset (Cassano et al., 2023). Since the models were instruction tuned, there are various formats for the chat template. We therefore provide the prompts on a high level. The system prompt has been prepended to the first user prompt where no system prompts were available. The model instruction is the comment preceding the problem in the original benchmark presentation. To match the chat interaction template, we remove the comment prefix `//` and insert it as a user prompt. Sample test cases are preserved when present. The model completion starts from a pre-filled assistant response. The prompts for synthesis and translation are presented in Figs. 11 and 12.

In the repair setting, we add the model output, annotated with line numbers, as assistant output and use a second turn to pass back compiler outputs as demonstrated in Fig. 13. Compiler errors contain line numbers for localization, which is why we annotate the output with line numbers.

We find that Qwen2.5 32B tends to always generate test cases, which lead to errors during compilation. We therefore append the sentence `Do not include test cases in the code.` to its prompt.

Executable Code Generation and Extraction When generating code, we employ two techniques to speed up generation and extract the generated code. First, we stop generation after encountering a closing code-fence (` ``` `). This drastically reduces generation time, as the model usually continues to generate some explanation or additional code that is irrelevant to our evaluation. Then we try to extract the generated function and all relevant subsequent function definitions.

We found that unconstrained models frequently generate non-compiling code caused by generated unsolicited demonstrations and additional test cases. In order to remove them and decrease such irrelevant compilation errors for unconstrained generation, we try to detect and extract the relevant code snippet.

Concretely, we first extract the corresponding TypeScript code block (i.e., ` ```typescript...``` `), or do not cut off if the block is not closed. Inside the code block, we cut off after the closing curly brace of the last balanced pair of curly braces, if it is followed by a newline or semicolon. This determines the last statement block generated, and avoids cutting off, e.g., inside a template literal. Again, if no such case is found, we do not prune the output. We demonstrate the operation of this cutoff algorithm in Fig. 10.

H Case Study Full Outputs

In this section, we provide the full outputs of the examples shown in the case study in Figs. 14, 15a, 15b and 16. Details about the functionality for each instance are provided in the respective captions.

````
```typescript
function next_smallest_palindrome(num: number): number {
  const numStr = num.toString();
  const reversedNumStr = numStr.split('').reverse().join('');
  let min = Math.min(Number(numStr), Number(reversedNumStr));

  if (min === Number(numStr)) {
    return Number(numStr);
  } else {
    return parseInt(min.toString() + 1);
  }
}

const findNextSmallestPalindrome = next_smallest_palindrome(12321);
console.log(findNextSmallestPalindrome); // Output: 12322
```

**Explanation**

1. **Input:**  The function takes a number as input (`num`) in the given criteria.
...
````

Figure 10: The code generated by Gemma 2 2B for MBPP #0. The cutoff algorithm first extracts the fenced code, last line marked in blue. To avoid the usage of the `console` object from the browser extension of TypeScript in the generated examples, we further cut off after the last code block, marked in red.

```
System:
  You are an expert in TypeScript programming.  Solve the given problem by writing solution code in TypeScript.
  When answering, insert the solution code in a ```typescript...``` block.
  Do not include test cases in the code.

User:
  Check if in given array of numbers, are any two numbers closer to each other than
  given threshold.
  >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
  false
  >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
  true function

Assistant:
  ```typescript
  function has_close_elements(numbers:  number[], threshold:  number):  boolean {
```

Figure 11: The full prompt for the synthesis setting. Text in green is based on the problem instance, in this case HumanEval #0. The red sentence is only added for Qwen2.5 32B due to its tendency to include non-compiling test cases.

```
System:
  You are a helpful and expert programmer in Python and TypeScript.  You will be given an input program in Python and your task is
  to translate this program into TypeScript.  You may assume that the input program is correct and that the translation should be
  semantically equivalent.
  When answering, insert the solution code in a ```typescript...``` block.
  Do not include test cases in the code.

User:
  The following is the source program in Python:
  ```python
  from typing import List

  def has_close_elements(numbers: List[float], threshold: float) -> bool:
      for idx, elem in enumerate(numbers):
          for idx2, elem2 in enumerate(numbers):
              if idx != idx2:
                  distance = abs(elem - elem2)
                  if distance < threshold:
                      return True

      return False
  ```

  Please translate the source program to TypeScript.

Assistant:
  ```typescript
  function has_close_elements(numbers:  number[], threshold:  number):  boolean {
```

Figure 12: The full prompt for the translation setting. Text in green is based on the problem instance, blue is the original Python canonical solution, in this case HumanEval #0. The red sentence is only added for Qwen2.5 32B due to its tendency to include non-compiling test cases.

````
System:
  You are an expert in TypeScript programming.  Solve the given problem by writing solution code in TypeScript.
  When answering, insert the solution code in a ```typescript...``` block.

User:
  Given an array of positive integers x.  return a sorted array of all
  elements that hasn't any even digit.
  Note:  Returned array should be sorted in increasing order.
  For example:
  »> unique_digits([15, 33, 1422, 1])
  [1, 15, 33]
  »> unique_digits([152, 323, 1422, 10])
  []

Assistant:
  ```
  001: function unique_digits(x: number[]): number[] {
  002:   const filteredArray: number[] = []; // initialize empty array
  003:   for (let i = 0; i < x.length; i++) {
  004:     let strx = "" + x[i];
  005:     let filteredStr = strx
  006:         .toString()
  007:         .split("")
  008:         .filter((digit) => parseInt(digit) % 2 !== 0);
  009:     if (filteredStr.length > 0) {
  010:       filteredArray.push(...filteredStr);
  011:     }
  012:   }
  013:   return filteredArray.sort((a, b) => a - b);
  014: }
  ```

User:
  This output produced an error:
  ../../../../../tmp/tmpaewlckfv.ts(10,26):  error TS2345:  Argument of type 'string' is not assignable to parameter of type
  'number'.


  Write the program again, and make sure to fix the error this time.

Assistant:
  ```typescript
  function unique_digits(x:  number[]):  number[] {
````

Figure 13: The full prompt for the repair setting. Text in green is based on the problem instance, red is model output from the synthesis step, blue is compiler output. In this case HumanEval #104 and Gemma 2 2B on its own previous output. Generations from the synthesis step are annotated with line numbers to aid localizing the compiler error.
````

```
function select_words(s: string, n: number): string[] {
  const result: string[] = [];
  for (const word of s. split() ) {
    let nConsonants = 0;
    for (let i = 0; i < word.length; i++) {
      const char = word[i].toLowerCase();
      if (!['a', 'e', 'i', 'o', 'u'].includes(char)) {
        nConsonants++;
      }
    }
    if (nConsonants === n) {
      result.push(word);
    }
  }
  return result;
}
```

(a) The code generated by the unconstrained Gemma 2 2B when translating a Python solution. The model tries to call the `split` member of a *string* object without parameters. This is legal in Python but not allowed in TypeScript.

```
function select_words(s: string, n: number): string[] {
  const result: string[] = [];
  for (const word of s. split(' ') ) {
    let nConsonants = 0;
    for (let i = 0; i < word.length; i++) {
      const letter = word[i].toLowerCase();
      if (!['a', 'e', 'i', 'o', 'u'].includes(letter)) {
        nConsonants++;
      }
    }
    if (nConsonants === n) {
      result.push(word);
    }
  }
  return result;
}
```

(b) The code generated by Gemma 2 2B with type constraints. The constraint guides the model and help it generate the required argument to `split`, which determines the characters around which to split the string.

Figure 14: Comparison between the function generated by unconstrained Gemma 2 2B (Team et al., 2024) and the code generated with the guidance of type constraints on the problem `select_words` in HumanEval (Chen et al., 2021). This problem requires a function that selects words from a given list that contain a specified number of consonants.

```
function largest_divisor(n: number): number {
  for(let i = n - 1; i > 0; i--) {
      if(n % i === 0) {
          return i;
      }
  }
}
```

(a) The code generated by the unconstrained Deepseek Coder 33B Instruct. The function loops over all numbers smaller than `n` to determine the largest divisor. Even though functionally correct for positive numbers, this function returns `undefined` when the return statement in the loop is never called (e.g., for negative inputs), violating the type system rules.

```
function largest_divisor(n: number): number {
  for(let i = n - 1; i > 0; i--) {
      if(n % i === 0) {
          return i;
      }
  }
   return 1; // Return 1 if no such divisor found
}
```

(b) The code generated by Deepseek Coder 33B Instruct with type constraints. The system ensures that every path returns a value matching the annotated type. This forces the LLM to generate another line after the loop, and the model correctly chooses to generate a fallback return statement.

Figure 15: Comparison between the function generated by the unconstrained DeepSeek Coder 33B Instruct (Guo et al., 2024) and the code generated with the guidance of type constraints on the problem `largest_divisor` in HumanEval (Chen et al., 2021). The problem asks to compute the largest divisor of a given number.

```
function intersperse(numbers: number[], delimeter: number): number[] {
  return numbers.reduce(( acc , curr, index) => {
    acc.push(curr);
    if (index < numbers.length - 1) {
      acc.push(delimeter);
    }
    return acc;
  }, []);
}
```

(a) The code generated by the unconstrained Gemma 2 9B. The code uses the `reduce` member of arrays in TypeScript. The `reduce` member is a generic function that accepts a callback as first argument. This callback has three parameters: An accumulator, a current value and an index. The types of the second and third parameter are fixed to the element type of the callee array and `number`. The first argument type of the callback function is automatically inferred from the type of the second argument to the `reduce` function, in this case an empty array. The type of this array is `never`[], a special type denoting arrays without members. Since we can assign this type to the callee type `number`[], the accumulator paramter obtains type `never`[]. However, calling `push` on this type results in a type violation as the pushed value of type `number` is not compatible with the inferred element type `never`. This issue is well known in TypeScript and even human developers frequently stumble upon this issue (Various, 2024; nop33, 2024)

```
function intersperse(numbers: number[], delimeter: number): number[] {
  return numbers.reduce(( acc:  number[] , num: number, index: number) => {
    if (index > 0) {
      acc.push(delimeter);
    }
    acc.push(num);
    return acc;
  }, []);
}
```

(b) The code generated by Gemma 2 9B (Team et al., 2024) with type constraints. We force the model to generate type annotations for the callback function in `reduce` and thus enable the model to generate the correct type annotation and avoid the issue in the unconstrained code.

Figure 16: Comparison between the function generated by the unconstrained Gemma 2 9B (Yang et al., 2024) and the code generated with the guidance of type constraints on the problem `intersperse` in HumanEval (Chen et al., 2021). The task in this problem is to insert a delimiter number between consecutive elements of an input list.