TYPE-AWARE CONSTRAINING FOR CODE LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Models (LLMs) have achieved notable success in code generation. However, they still frequently produce invalid code, as they do not precisely model formal aspects of programming languages. Constrained decoding is a promising approach to alleviate this issue and has been successfully applied to domain-specific languages and syntactic features, but is not able to enforce more semantic features, such as well-typedness. To address this issue, we introduce *type-aware constrained decoding*. We develop a novel prefix automata formalism and introduce a sound approach to guarantee existence of a type-safe completion of a partial program based on type inference and a search over inhabitable types. We implement type-aware constraining first for a foundational simply-typed language, then extend it to TypeScript. In our evaluation across state-of-the-art open-weight LLMs of up to 34B parameters and various model families, type-aware constraining reduces compilation errors by on average 70.9% and increases functional correctness by 16.2% in code synthesis, translation, and repair tasks.

1 INTRODUCTION

Large language models (LLMs) are increasingly utilized in everyday coding tasks (GitHub, 2025; Vella, 2024; Rozière et al., 2023; Gemma Team, 2024). They excel at generating code from natural language descriptions (Rozière et al., 2023; Lozhkov et al., 2024), translating between programming languages (Rozière et al., 2023), and repairing programs (Muennighoff et al., 2024; Zhang et al., 2024). Despite these achievements, LLMs often produce incorrect code, leading to compilation errors, logic flaws, and security vulnerabilities (Pan et al., 2024; Dou et al., 2024; Pearce et al., 2022). This issue arises from the underlying probabilistic nature of LLM-based code generation, which is derived solely from training data, without leveraging the strict rules of programming languages.

A promising solution to this challenge is *constrained decoding*, which enforces hard constraints during generation. Prior work employed this to enforce syntactic rules of programming languages on LLM generations (Beurer-Kellner et al., 2024; Ugare et al., 2024; Poesia et al., 2022).

LLMs Struggle to Generate Type-Safe Code However, we observe that LLMs struggle to generate well-typed code (Dou et al., 2024; Tambon et al., 2024), as type systems significantly complicate the generation of valid code (Szabo et al., 2024). In our evaluation on average over 90% of compilation errors result from failing type checks. Meanwhile, type systems are crucial for detecting and rejecting bugs at compile time (Mitchell, 1990; Matsakis and Klock, 2014) and are therefore employed in many popular programming languages (Bierman et al., 2014; Donovan and Kernighan, 2015; Arnold et al., 2000). A natural path forward is to leverage type systems for constrained code generation. This is challenging, because type systems, even in simply typed lambda calculus, cannot be described by context-free grammars (Mitchell, 1990), inhibiting the application of existing constraining methods (Ugare et al., 2024; Beurer-Kellner et al., 2024). In addition to deriving and maintaining type information, partially generated expressions need to be accurately assessed. This task extends beyond traditional type inference, which determines *inhabitation* of a desired type (Appel, 1998), to a generalized version of the PSPACE-complete Type Inhabitation Problem, which seeks to determine whether *any* expression *can* inhabit a desired type (Urzyczyn, 1997).

Syntactic Constraining is Insufficient However the challenge is worth tackling. Consider the example in Fig. 1, in which an LLM generates TypeScript code. Based on syntax alone, completions such as non-arithmetic characters and line termination are rejected (1), however, the suggested,

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

<completion>	Standard	Syntax	Types	Rejection Reason
`;`	–	⊘	⊘	(1) Syntactically invalid
`ber`	–	–	⊘	(2) Undeclared identifier
`[0]`	–	–	⊘	(3) Disallowed operator
`, 10)`	–	–	⊘	(4) Invalid argument type
`.toString()`	–	–	–	(5) –

Figure 1: A partial solution by CodeLlama 34B (Rozière et al., 2023). The completion `", 10)"` is accepted (–) by standard and syntax-only constraining, but correctly rejected (⊘) by type-aware constraining, since it violates the type signature of `parseInt`. Instead `.toString()` is accepted.

invalid completion does not violate syntactic rules. More meaningful constraints can be derived from the type system. For instance, no other identifier beginning with `num` is defined, leading to a rejection of, e.g., `number` (2). Also, `num` is of type *number*, which disallows computed member access (3). Critically, we recognize that `num` is not a valid first argument for `parseInt` due to mismatching types (4). Consequently, the prediction `", 10)"` is correctly rejected by type-aware constraints. Meanwhile, a valid completion for the given code exists, since, e.g., the completion `.toString()` yields type *string*, satisfying `parseInt` (5). Our approach correctly detects this admissible output, guiding CodeLlama (Rozière et al., 2023) to complete the program correctly.

Type-Aware Constrained Decoding In this work, we address the challenge of constrained generation of type-safe code, referred to as *type-aware constrained decoding*. We develop a sound algorithm based on a novel prefix automaton formalism, to determine if a partial output can be completed into a well-typed program. This non-deterministic automaton incrementally constructs potential abstract syntax trees described by the partial input, annotated with type-relevant context, such as inhabited types, declared identifiers, and expected return types, and maintains a *prefix property* to ensure that only valid language prefixes result in a non-empty set of states. To guarantee this property when parsing expressions, we design a sound type search algorithm to determine whether a partially written expression can inhabit a desired type. We carefully analyze the search graph to restrict the search to a finite subgraph covering sufficiently complex types. We develop the automaton for a generic, simply-typed Turing-complete language. For our evaluation, we adapt the automaton to support a comprehensive subset of TypeScript (Bierman et al., 2014; Gao et al., 2017), a typed superset of JavaScript, which is currently one of the most actively used languages in open-source projects on GitHub (Madnight, 2024; GitHub, 2022).

We evaluate the resulting system code on code synthesis, translation, and repair of a TypeScript version of HumanEval (Chen et al., 2021; Cassano et al., 2023). Our results show that type-aware constrained decoding significantly enhances LLMs of 2B to 34B parameters and various model families in across all tasks, reducing compilation errors by on average 70.9% and increasing functional correctness by 3.3%, 6.9% and 38.7% respectively.

Main Contributions Our main contributions are: (i) We design a prefix automaton for type-aware constrained decoding based on a generic, simply-typed Turing-complete language. (ii) We extend the automaton to the popular and widely used programming language TypeScript. (iii) We conduct an extensive evaluation of the impact of type-aware constraining on model performance, demonstrating both reductions in compilation errors and increases in functional correctness.

2 Background: Constraining LLMs

LLMs struggle to infer rules for programming languages (Ebrahimi et al., 2020; Bhattamishra et al., 2020; Angluin et al., 2023), may only derive incomplete grammars for less common languages (Cassano et al., 2023; Orlanski et al., 2023), and do not consistently follow grammatical rules during the decoding process, due to the probabilistic nature of code generation. For instance, in our evaluation, Gemma 2 9B (Rozière et al., 2023) achieves a notable pass@1 accuracy of 70.8% translating Python solutions to TypeScript in HumanEval (Cassano et al., 2023; Chen et al., 2021), yet over half of the non-passing samples do not even pass syntax and type checks, and thus are invalid programs.

To address this issue, *constrained decoding* ensures generated outputs are valid according to a language L's rules. It works by forcing partial outputs to remain valid prefixes of language L (Poesia et al., 2022). This requires a *completion engine* CE_L, which verifies whether a string x is a prefix of some word in L, i.e., $CE_L(x) = \exists y : x{+}y \in L$. Shown in Algorithm 1, constrained decoding starts with an initial string or prompt, prefix to some word in L (e.g., ϵ), and appends tokens sampled from the LLM, rejecting tokens that would not result in a prefix of any word in L. Upon rejection, another token is sampled from the LLM until a valid token is returned. Since x is a prefix of *some* word in L, such a token must exist.

Algorithm 1 Constrained Decoding

Input: Prompt x, LLM M, Completion Engine CE_L
Output: o, such that $x + o \in L$

```
o := []
loop
    v := logits(M, x + o)
    do t := sample(v)
    while not CE_L(x + o + t)
    if t = EOS and o ∈ L then
        return o
    o.append(t)
```

It is thus guaranteed that all sampled strings are valid prefixes of L. This process continues until, while the current output is in L, the LLM predicts the "end of sequence" token (EOS), signaling the completion of the generation. The only invalid outcome is due to non-termination.

Crucially, constrained decoding steers the model towards correct completions when it attempts to generate invalid completions. Recent work has explored the use of programming languages' syntactic rules for constraining outputs; however, the benefits of such approaches are limited (Poesia et al., 2022; Ugare et al., 2024; Beurer-Kellner et al., 2024) as syntactic correctness is insufficient to guarantee compilability and executability in the presence of type systems.

3 Type-Aware Constrained Decoding

3.1 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 propagated type environment Γ, which is a map from identifiers to types, updated by assignments. We write $\Gamma \vdash e : T$ if the expression e has type T in the type environment Γ. Statements propagate the type environment.

A word is in L_B if it (i) is syntactically valid, (ii) permits the derivation of type environments for statements and types for expressions, and (iii) ensures a value of the indicated return type is returned on every execution path. Detailed syntax and type inference rules are presented in App. B.

3.2 Prefix Automata

We define an automaton $A(\Sigma, \mathcal{S}, f, I, Q)$ as a five-tuple of (i) Σ: An alphabet of input symbols. (ii) $\mathcal{S}$: A potentially infinite set of states. (iii) $f : \mathcal{S} \times \Sigma \mapsto \mathcal{P}(\mathcal{S})$: A computable transition function that maps a state and an input symbol to a finite set of next states. (iv) $I \subseteq \mathcal{S}$: A finite set of initial states. (v) $Q \subseteq \mathcal{S}$: A potentially infinite, decidable set of accepting states.

```
l ::= \d+ | "\w*" | true | false
x ::= \w+
p ::= x : T
T ::= string | number | boolean | ( p̄ ) => T
e ::= l | x | ( p̄ ) => e | ( e ) | e ∘ e | e ( ē ) | e . n

s ::= { s̄ }
      let x : T ;
      e ;
      if ( e ) s else s
      function x ( p̄ ) : T { s̄ }
      return e ;
M ::= s̄
```

Figure 2: The grammar for identifiers x, literals l, expressions e, types T (left), statements s and the complete program M (right) in L_B. x and l are based on regular expressions.

In the following, we assume the parameters are clear from context, and denote the automaton as A. The transition function f maps a given state to all possible subsequent states. When the first argument of f is a set, we implicitly take the union of the results, with $f(\mathcal{S}, t) \coloneqq \bigcup\{f(S, t) \mid S \in \mathcal{S}\}$.

The transition function defines a directed graph G over $\mathcal{S}$. The language parsed by A comprises all strings $s \in \Sigma^*$ such that traversing G from any state in I according to f results in states S of which at least one is in Q. Formally, we define a reachability function r for states S and input symbol t recursively as $r(S, s+t) \coloneqq f(r(S,s),t)$ and $r(S,\epsilon) \coloneqq S$. The language described by A is equivalently formulated as $L(A) \coloneqq \{s \in \Sigma^* \mid r(I,s) \cap Q \neq \emptyset\}$.

The reachability function has intuitive and useful properties concerning graph reachability. We use $x \leq y$ to notate that x is a prefix of y, i.e., $\exists s \in \Sigma^* : x + s = y$.

(P1) An empty string does not alter the state, i.e., $r(S, \epsilon) = S$.

(P2) A path along the graph can be split arbitrarily, i.e., $r(S, s + s') = r(r(S, s), s')$.

(P3) For a reachable state, all prefixes are valid, i.e., $r(S, s) \neq \emptyset \implies \forall p \leq s : r(S, p) \neq \emptyset$.

(P4) The reachable states from some states equals the union of reachable states from each state, i.e. $r(\bigcup X, s) = \bigcup_{X' \in X} r(X', s)$.

Property (P1) follows directly from the definition of the transition function, while properties (P2), (P3), (P4) can be shown inductively. Further, to parse the prefix language efficiently, we need to ensure that a path exists from every reachable state to an accepting state, the *prefix property*.

Definition 1. *A is a* prefix automaton *iff* $\forall p \in \Sigma^*, \forall S \in r(I,p), \exists s \in \Sigma^*$ *s.t.* $r(S,s) \cap Q \neq \emptyset$.

For such automata, reaching any state through string p implies p is a prefix to some word in L. The *reachable language* of A, all strings that result in any state, is $L_r(A) \coloneqq \{p \in \Sigma^* \mid r(I,p) \neq \emptyset\}$. The set of all prefixes of a language is $L^p \coloneqq \{p \in \Sigma^* \mid \exists w \in L : p \leq w\}$.

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

Proof. From (P3) follows that $L(A)^p \subseteq L_r(A)$ since $s \in L(A) \implies r(I,s) \neq \emptyset$. If A is a prefix automaton, the reverse also holds, i.e., $L_r(A) \subseteq L(A)^p$, since $s \in L_r(A) \implies \exists S \in r(I,s) \implies \exists s' \in \Sigma^* : s' \leq s \wedge r(I,s) \cap Q \neq \emptyset$, which by definition means that s is a prefix of $s' \in L(A)$. □

We use the reachability function of A as completion engine $CE_A(s) \coloneqq r(I,s) \neq \emptyset$. It now suffices to show that $L = L(A)$ and that A is a prefix automaton, such that $CE_A(p) \iff \exists w \in L : p \leq w$.

We further introduce some base automata in Fig. 3. Their precise definitions are presented in App. C.

3.3 Automata for Identifiers, Literals, and Types

We define several automata X to parse expressions of L_B as well as their *type-restricted* versions $X \downarrow T$, which only accept if the parsed expression has type T.

Literals The automata L_{string}, L_{boolean}, and L_{number} accept strings, booleans, and numbers. They are defined by regular expressions and implemented using corresponding deterministic finite automata. To ensure the prefix property, states that have no paths to accepting states are pruned. The restricted version $L_x \downarrow T$ is $A_\emptyset$ if the literal's type does not match T. $A_{lit} \coloneqq L_{\text{string}} \cup L_{\text{boolean}} \cup L_{\text{number}}$.

Automaton	Accepted Language	Prefix Automaton when…
$A_X \cup A_Y$	$\{w \mid w \in L(A_X) \cup L(A_Y)\}$	A_X, A_Y are prefix automata
$A_X \oplus A_Y$ or A_{XY}	$\{w + v \mid w \in L(A_X), v \in L(A_Y)\}$	A_X, A_Y are prefix automata, $L(A_Y) \neq \emptyset$
A_X^+ or $A_{\overline{X}}$	$\{w^k \mid k \in \mathbb{N}, w \in L(A_X)\}$	A_X is a prefix automaton
θ_x	$\{x\}$	$x \neq \epsilon$
$A_\emptyset$	$\emptyset$	Always

Figure 3: Base prefix automata

Identifiers During parsing, we maintain the current type environment by passing it to the next state via the f. The identifier automaton A_{ident} is the union of terminal automata accepting defined identifiers, i.e., $A_{\text{ident}} := \bigcup_{(x:T)\in\Gamma} \theta_x$. For $A_{\text{ident}} \downarrow T$, we only include identifiers of type T.

Types The type automaton A_T accepts type expressions T. Its co-recursive definition is $A_T := A_{\text{type-lit}} \cup A_{\text{fun-type}}$ with type literal automaton $A_{\text{type-lit}} := \theta_{string} \cup \theta_{number} \cup \theta_{boolean}$ and function type automaton $A_{\text{fun-type}} := C_{(\overline{p})\Rightarrow T}$, using parameter automaton $A_p := C_{x:T}$. This definition preserves the prefix property since we maintain a finite initial state set and a decidable accepting set.

3.4 Expression Automaton

The expression automaton A_e is defined co-recursively as $A_{\text{ident}} \cup A_{lit} \cup A_{(e)}$, and $A_{(\overline{p})\Rightarrow e}$, with *extensions* via operator automata $A_{\circ e}$, $A_{.n}$, and $A_{(\overline{e})}$. These automata are constructed by concatenating the respective terminal automata and (recursively) A_e, detailed in App. C.3. To implement extensions, f_e is adjusted to add outgoing edges from accepting states to initial states of extending automata.

$$\forall X, Y : f_e(S_Y^X, t) := \begin{cases} f_Y(S_Y^X, t) \cup f_e(I_{(\overline{e})}^{XY}, t) \cup f_e(I_{\circ e}^{XY}, t) \cup f_e(I_{.n}^{XY}, t) & \text{if } S \in Q_e \\ f_Y(S_Y^X, t) & \text{otherwise} \end{cases}$$

Accepting states of extending automata are considered accepting states of A_e, and previously parsed expressions are passed to the extending automaton, indicated by the superscripts X and Y. Information about preceding expressions is used to impose restrictions on operands; e.g., to ensure that parameters match the respective argument types in a parsed function's signature.

For $A_e \downarrow T$, we determine whether any completion of input p *can* inhabit T. Notably, repeated application of extensions can alter the result type entirely. To address this issue, we first identify the inhabitable types $\text{DERIVABLE}(p)$ of p without extensions, then perform a type-level search $\text{REACHABLE}(\text{DERIVABLE}(p), T)$ to determine if type T can be inhabited by applying admissible operators. For negative results, we prune transitions from f_e, since they violate the prefix property.

Derivable Types The *derivable types* of state $S \in r(I_e, p)$ are types p can inhabit without operators. If S is accepting, $\text{DERIVABLE}(S) := T$, where $\Gamma \vdash p : T$. Different expressions impose different rules on derivability, as shown in Fig. 4, using pmatch(p, L) if prefix p matches the regular expression of literal L partially.

e	$\text{DERIVABLE}(e)$
x	$\{T \mid x \leq n, (n : T) \in \Gamma\}$
l	$\{L \mid \text{pmatch}(l, L), L \in \{number, string, boolean\}\}$
$(\ e$	$\{T \mid \text{REACHABLE}(\text{DERIVABLE}(e), T)\}$
$e\ \circ$	$\{T \mid \Gamma \vdash e : S \wedge \exists S'. S \circ S' : T\}$
$e\ ($	$\{R \mid \Gamma \vdash e : (\overline{p}) \Rightarrow R\}$
$e\ .\ p$	$\{T \mid p \leq n, \Gamma \vdash e : T, \text{lookup}(e, n) = T\}$
$(\overline{p})$ => e	$\{(\overline{p}) \Rightarrow T \mid \text{REACHABLE}(\text{DERIVABLE}(e), T)\}$

Figure 4: Definition of $\text{DERIVABLE}(x)$ when extending expressions introduced in Fig. 2.

For group expressions, function literals, and array expressions, we need to enumerate potentially infinitely types subexpressions could inhabit. To address this, we integrate enumeration with the type reachability search in App. C.6.

Lemma 2. *For state $S \in r(I_e, p)$ of partial expression p, $\text{DERIVABLE}(S)$ returns all T such that there exists some suffix s with $\Gamma \vdash ps : T$ and s does not involve an extension.*

Type Reachability To determine all reachable types of some expression of type T, we analyze sequences of operators with compatible signatures. These implicitly define a search graph, illustrated in Fig. 5. Specifically, there is an edge from T to S for operator $\circ$ with signature $T \circ X : S$. We treat function calls and member accesses as operators with $T\ ()\ \overline{p} : R$ and $T\ .\ n : \text{LOOKUP}(T, n)$, respectively, where LOOKUP returns the type of member n of T.

We observe a pattern: From type T, we reach (i) itself, (ii) result types of arithmetic operators, (iii) return types, and (iv) member types. Thus, exploring higher-order type $() \Rightarrow T$ does not yield other types than T, when $() \Rightarrow T$ has no members with new types. Consequently, we avoid exploring higher-order types unless necessary for soundness, soundly restricting the search to a finite subgraph by limiting the maximum *depth* of explored types that do not provide new *root types*. The validity of this observation hinges on the definition of the LOOKUP function and operators introduced in §3.1, particularly that no new root types can be reached through the same type of a higher order.

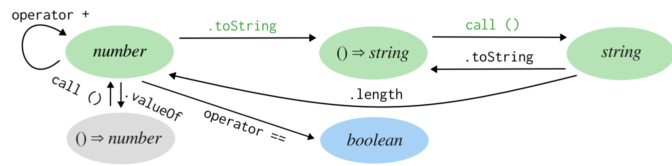

Figure 5: Example type reachability search, starting from $t = number$ with the goal $string$, resulting from, e.g., $x =$ `"let x : string; x = 1"`. States and edges along the final path are green, explored nodes blue. Note how $() \Rightarrow number$ is not explored because of its depth and providing no additional root types, while $() \Rightarrow string$ is explored as it provides access to new root type $string$.

We define the depth DEPTH of a type T as an integer that represents the function's order, and don't explore S on the search for G when $\text{DEPTH}(S) > \max(\text{DEPTH}(G), \text{DEPTH}(T))$. This addresses the fact that we may need to explore up to higher order, if the target type is of higher order itself. Further, if a new type is reachable through a higher-depth function, we also need to explore it. Shown in Fig. 5, type $number$ has the member `toString` of type $() \Rightarrow string$, which is only accessible by traversing through the higher-depth node. We define the root types of some type T as the types of depth 0 (e.g., $string$, $number$, $boolean$) that it comprises. All such types are potentially reachable through T. The described search is sound.

Lemma 3. *For any expression e with $\Gamma \vdash e : T$, if $\text{REACHABLE}(T, G)$ holds, then there exists an extension y such that $\Gamma \vdash ey : G$.*

We defer the proof for Lemma 3 to App. D.1 and present the resulting search algorithm, REACHABLE, and the formal definitions of depth and root types in App. C.3. Combined with the DERIVABLE results, we derive the search algorithm to determine most types a partial input can inhabit.

Corollary 1. *For states $S = r(I_e, p)$ of a partial expression p, if $\text{REACHABLE}(\text{DERIVABLE}(S), G)$ holds, then there exists a suffix y such that $\Gamma \vdash py : G$.*

Proof. Follows directly from Lemma 3 and Lemma 2. □

We apply the reachability and derivability algorithms to prune transitions of operator $\circ$ if T can not be inhabited after applying $\circ$ to e. Importantly, however, in any case $L(A_e \downarrow T) \neq \emptyset$, as any type is expressable using literals and anonymous functions.

Lemma 4. *$L(A_e \downarrow T) \subseteq \{w \mid \Gamma \vdash w : T\}$ and A_e is a prefix automaton.*

3.5 Statement Automaton

We define the remaining automata to capture the complete language from §3.1. The single statement automaton A_s is $A_{e;} \cup A_{\text{decl}} \cup A_{\text{block}} \cup A_{\text{ITE}} \cup A_{\text{fun}} \cup A_{\text{ret}}$. The declaration automaton $A_{decl} \coloneqq C_{let\ x:T;}$ captures undefined variable names x, by accepting on all strings, except if it matches an existing identifier. The return statement automaton is $A_\emptyset$ when outside a function and restricts the parsed expression to the declared return type otherwise. The remaining automata are detailed in App. C.5.

Guaranteeing Return Types When parsing the function body, the transition function maintains state to track the expected return type and enforce return in all execution paths. Whether all execution paths return can be determined based on previously parsed statements, i.e., for any multi-statement automaton $A_{\overline{s}}$, if it is inside a function and must return, and the most recently parsed statement did not return in all execution paths, it can not accept. Instead, it forces the generation of another statement. Since we can always express the requested type through literals and generate a return statement, the prefix automaton property is not violated.

Tracking Type Environments Identifiers are passed through all state transitions. However, in the cases of BLOCK, ITE and FUN, the local type environment is discarded after parsing. In FUN, the function signature and parameters are added to the environment of the function body automaton.

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

Table 1: Instances with compiler errors in Standard, Syntax, and our Types constraining. Type-aware constraining on average reduces errors by 74.8% in the Synthesis of HumanEval, compared to only 3.3% through ideal syntax-only constraining. For Translation and Repair, we observe similar improvements with 76.0% and 62.0% respectively.

	Synthesis			Translation			Repair		
Model	Standard	Syntax	Types	Standard	Syntax	Types	Standard	Syntax	Types
Gemma 2 2B	100	$97_{\downarrow 3.0\%}$	$\mathbf{40}_{\downarrow 60.0\%}$	195	$184_{\downarrow 5.6\%}$	$\mathbf{80}_{\downarrow 59.0\%}$	200	$189_{\downarrow 5.5\%}$	$\mathbf{105}_{\downarrow 47.5\%}$
Gemma 2 9B	44	$43_{\downarrow 2.3\%}$	$\mathbf{12}_{\downarrow 72.7\%}$	97	$90_{\downarrow 7.2\%}$	$\mathbf{15}_{\downarrow 84.5\%}$	121	$114_{\downarrow 5.8\%}$	$\mathbf{46}_{\downarrow 62.0\%}$
Gemma 2 27B	15	$15_{\downarrow 0.0\%}$	$\mathbf{1}_{\downarrow 93.3\%}$	28	$28_{\downarrow 0.0\%}$	$\mathbf{2}_{\downarrow 92.9\%}$	71	$44_{\downarrow 38.0\%}$	$\mathbf{29}_{\downarrow 57.7\%}$
DeepSeek C. 33B	25	$25_{\downarrow 0.0\%}$	$\mathbf{7}_{\downarrow 72.0\%}$	20	$20_{\downarrow 0.0\%}$	$\mathbf{6}_{\downarrow 70.0\%}$	48	$48_{\downarrow 0.0\%}$	$\mathbf{14}_{\downarrow 68.8\%}$
CodeLlama 34B	82	$70_{\downarrow 14.6\%}$	$\mathbf{30}_{\downarrow 63.4\%}$	190	$150_{\downarrow 21.1\%}$	$\mathbf{66}_{\downarrow 65.3\%}$	155	$145_{\downarrow 6.5\%}$	$\mathbf{54}_{\downarrow 65.2\%}$
Qwen2.5 32B	32	$32_{\downarrow 0.0\%}$	$\mathbf{4}_{\downarrow 87.5\%}$	32	$27_{\downarrow 15.6\%}$	$\mathbf{5}_{\downarrow 84.4\%}$	41	$38_{\downarrow 7.3\%}$	$\mathbf{12}_{\downarrow 70.7\%}$

Table 2: Functional correctness of outputs, measured in pass@1 (in %) of unconstrained (Standard) and type-aware constrained (Types) generated code for the tasks Synthesis, Repair, and Translation.

	Synthesis		Translation		Repair	
Model	Standard	Types	Standard	Types	Standard	Types
Gemma 2 2B	29.4	**30.3**	48.6	**53.5**	10.1	**19.1**
Gemma 2 9B	56.9	**58.6**	70.8	**78.1**	22.1	**32.9**
Gemma 2 27B	69.5	**71.4**	85.1	**87.3**	33.9	**38.9**
DeepSeek Coder 33B	68.7	**70.6**	88.7	**90.4**	44.3	**48.3**
CodeLlama 34B	41.4	**43.2**	54.7	**62.1**	16.8	**26.8**
Qwen2.5 32B	77.5	**80.3**	91.0	**93.9**	62.8	**69.5**

4 Experimental Evaluation

We adapt our method presented in §3 for language-specific features of TypeScript, described in App. D, and present extensive evaluation results in this section.

Tasks and Datasets We evaluate three relevant tasks, in which the model completes a function, given its header. (i) *Synthesis*: Synthesize a program that solves a task in natural language. (ii) *Translation*: Translate a Python function into TypeScript. (iii) *Repair*: Resolve the compilation error of a buggy solution. The full prompts to all tasks are in App. H. The tasks are based on TypeScript-translated HumanEval instances in MultiPL-E (Cassano et al., 2023; Chen et al., 2021). For Synthesis and Translate, we use the original prompts and the original Python version from the HumanEval dataset (Chen et al., 2021) and run evaluations 4 times. For Repair, we collect all non-compiling programs from unconstrained Synthesis for all models, resulting in 298 non-compiling programs and run evaluation once. We report aggregated instances with TypeScript compiler errors (Microsoft, 2024) and the overall percentage passing all functional tests (pass@1).

Models and Sampling We evaluate six open-weight code LLMs of different sizes and model families: instruction-tuned Gemma 2 2B, 9B, and 27B (Gemma Team, 2024), DeepSeek Coder 33B Instruct (Guo et al., 2024), CodeLlama 34B Instruct (Rozière et al., 2023), and Qwen2.5 32B Instruct (Hui et al., 2024). We report unconstrained sampling as *Standard*, the ideal achievable improvement using syntactic constraining, i.e., assuming syntactic constraining always resolves the respective instance, including potential typing errors, as *Syntax*, and the result of our type-aware constrained sampling method as *Types*. Further settings and hyperparameters are detailed in App. E.

4.1 Experimental Results

Reduction of Compilation Errors In Synthesis, shown in Table 1, on average only 3.3% of compiler errors are due to syntactic errors (Syntax), with Gemma 2 27B, DeepSeek Coder 33B and Qwen2.5 32B even making no syntactic errors at all. Using type-aware constraining drastically reduces compilation errors, on average by 74.8%. We observe that models of all sizes and model families benefit similarly from the constraining, with a minimum reduction of 60.0%, even though the total amount of compiler errors varies strongly, e.g., between 1 (Gemma 2 27B) and 40 (Gemma

	Standard	Types

(a)	`// use crypto to calculate MD5 of text`
	`const hash = crypto`
	`.createHash("md5").update(text);`
		`// use crypto to calculate MD5 of text`
		`const hash = require("crypto")`
		`.createHash("md5").update(text);`

(b)	`function fibfib(n: number): number {`
	`// initialize helpers`
	`let result;`
	`// calculate fibfib value`
	`return result;`
	`}`
		`function fibfib(n: number): number {`
		`// initialize helpers`
		`let result = 1;`
		`// calculate fibfib value`
		`return result;`
		`}`

(c)	`// check if numStr contains even digit`
	`return !numStr.some(`
	`digit => parseInt(digit) % 2 === 0);`
		`// check if numStr contains even digit`
		`return !numStr.split("").some(`
		`digit => parseInt(digit) % 2 === 0);`

Figure 6: Three example impacts of type-aware constrained sampling. Left are unconstrained generations with errors highlighted in red, right constrained results with corrected tokens highlighted in green, adapted for clarity.

2 2B) compiler errors. For Translation, a similar pattern can be observed, with a reduction of 76.0% of compiler errors as opposed to only 8.25% ideal syntax-only improvement. We notice that this often results from the model incorrectly transferring builtins and methods of the source language into the target language, which our constraining prevents.

In Repair, we find that many models struggle to correctly localize and resolve compilation errors, with Gemma 2 2B for example repairing only 35.7% of instances, increased to 66.2% through type-aware constraining. On average, 62.0% of the errors unresolved by compiler feedback alone are solved through type-aware constrained sampling.

Improving Functional Correctness We compute the pass@1 performance of the generated code on the test cases provided in the translated datasets of MultiPL-E (Cassano et al., 2023) and present the results in Table 2. We find that our constraining impressively increases success rate by 3.25%, 6.9%, and 38.6% in the Synthesis, Translation, and Repair tasks respectively. Especially weaker models benefit in Repair, with Gemma 2 2B increasing functional correctness by 89%.

MBPP in Appendix We further run the same experiment on MBPP (Austin et al., 2021), observe similar results to HumanEval and present them in more detail in App. F.

4.2 Case Study

To evaluate the performance of our approach qualitatively, we manually inspect successfully corrected, originally failing instances. We find that our technique effectively amends various types of compilation errors, shown adapted for clarity in Fig. 6. The complete outputs are in App. G.

Missing Import In Fig. 6a (HumanEval #162, TypeScript Translation, 2023), model is tasked to compute an MD5 hash. During the generation of Gemma 2 2B, a library called `crypto` is invoked. However, the library has not been imported yet. With type-aware constraints, the generation first invokes `require` to import the `crypto` library, correctly gaining access to the library API methods.

Type Mismatch In Fig. 6b, the task is to write code for a specialized Fibonacci sequence. During the generation of DeepSeek Coder 33B,variable `result` is declared to store the final result value. However, the variable `result` is not initialized, resulting in type $number \mid undefined$, causing a mismatch with the function return type $number$. Using type-aware constraints, the model is forced to either annotate declared variables or initialize them, resulting in a correct initialization.

Hallucinated Method In Fig. 6c the task is to filter out numbers with even digits. In the code generated by Qwen2.5 32B Instruct the member method `some` on the string representation of the number `numStr` is called. However, `some` only exists for lists. Type-aware constraints restrict the completion to only valid members of the *string* type, resulting in `split("")`, correctly decomposing the string into a list of digits.

5 Related Work

Type Systems for Code Synthesis Prior work has shown that leveraging type systems for code completion (Gvero et al., 2013; Agrawal et al., 2023; Wei et al., 2023) and SQL query generation (Poesia et al., 2022) is effective. Notably, Gvero et al. (2013) employed a search on the type graph, using constraints from *succinct types* to address the type inhabitation problem. However, unlike our approach, prior work is confined to specific scenarios (e.g., function call completion, identifier completion) and did not encompass entire type systems or generate entire programs. Specifically, they do not tackle the challenge of determining types that can be extended from partial expressions.

Code LLMs LLMs achieve outstanding results on tasks such as code synthesis, repair, or translation (Jiang et al., 2024). Specialized models have been trained on code-specific datasets, contributing to improved performance in code-related benchmarks (Rozière et al., 2023; Gemma Team, 2024; Guo et al., 2024; Mündler et al., 2024). Meanwhile LLMs are well known to frequently make mistakes (Rawte et al., 2023; Huang et al., 2023; Mündler et al., 2023), and, as we show in this work, even cutting-edge open-weight models with over 30 billion parameters frequently make errors in code generation.

Improving LLM Accuracy Three primary approaches have been proposed to enhance the accuracy of LLMs on code tasks: fine-tuning (Tsai et al., 2024; Weyssow et al., 2024), retrieval augmentation (Bassamzadeh and Methani, 2024), and compiler- or execution feedback (Jana et al., 2024; Deligiannis et al., 2023; Polikarpova et al., 2016). Such methods are complementary to our approach, e.g. type-aware constraining can be integrated with additional context from compiler feedback (as demonstrated in §4).

Constrained Decoding Context-free constrained decoding has been explored extensively in recent work (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 and the scope markers in Go, have been implemented (Melcer et al., 2024; Ugare et al., 2024). As demonstrated in §4, syntax errors account for on average 5.8% of compilation errors. Meanwhile, our work presents an implementation of constrained decoding for TypeScript, including its type system, that reduces compilation errors by 70.9% and significantly increases functional correctness.

6 Conclusion

In this work, we explored how the type system of programming languages can be used to guide LLMs during decoding. Concretely, we design and implement prefix automata to perform type-aware constraining for a foundational simply typed language and then extend it to the popular language TypeScript. We extensively evaluate the impact of using such constraints for code synthesis, translation, and repair and observe that we almost halve compilation errors on a diverse set of models and consistently increase functional correctness. We further explore qualitatively with concrete examples, how the constraining positively impacts code generation. We conclude that such type-aware constraining should be implemented for more programming languages, and has the potential to generally improve code generation in many domains.

REFERENCES

Lakshya A. Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Guiding Language Models of Code with Global Context using Monitors. *CoRR* abs/2306.10763 (2023). https://doi.org/10.48550/arXiv.2306.10763

Dana Angluin, David Chiang, and Andy Yang. 2023. Masked Hard-Attention Transformers and Boolean RASP Recognize Exactly the Star-Free Languages. *CoRR* (2023).

Andrew W. Appel. 1998. *Modern Compiler Implementation in C*.

Ken Arnold, James Gosling, and David Holmes. 2000. *The Java programming language*. Vol. 2.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. 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. https://arxiv.org/abs/2407.02742

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 186 (June 2023), 24 pages. 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. *CoRR* abs/2403.06988 (2024). https://doi.org/10.48550/arXiv.2403.06988

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. 2020. On the Ability and Limitations of Transformers to Recognize Formal Languages. In *EMNLP (1)*.

Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*. 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Trans. Software Eng.* 49, 7 (2023), 3675–3691. https://doi.org/10.1109/TSE.2023.3267446

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). https://arxiv.org/abs/2107.03374

Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. 2023. Fixing Rust Compilation Errors using LLMs. https://arxiv.org/abs/2308.05177

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, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Rongxiang Weng, Jingang Wang, Xunliang Cai, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. https://arxiv.org/abs/2407.06153

Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. 2020. How Can Self-Attention Networks Recognize Dyck-n Languages?. In *EMNLP (Findings)*.

Ecma International. 2016. *ECMAScript 2016 Language Specification*. Ecma International. https://262.ecma-international.org/7.0/index.html#sec-rules-of-automatic-semicolon-insertion Section: Rules of Automatic Semicolon Insertion.

Shimon Even. 2011. *Graph algorithms*.

Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 758–769.

Gemma Team. 2024. Gemma. (2024). https://www.kaggle.com/m/3301

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

GitHub. 2025. GitHub Copilot - Your AI pair programmer. https://github.com/features/copilot. https://github.com/features/copilot Accessed on February 07, 2025.

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

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 27–38. https://doi.org/10.1145/2491956.2462192

Chris Hokamp and Qun Liu. 2017. Lexically Constrained Decoding for Sequence Generation Using Grid Beam Search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, Vancouver, Canada, 1535–1546. https://doi.org/10.18653/v1/P17-1141

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 arXiv:2311.05232* (2023).

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. *CoRR* abs/2409.12186 (2024). https://doi.org/10.48550/arXiv.2409.12186

HumanEval #104, TypeScript Translation 2023. https://huggingface.co/datasets/nuprl/MultiPL-E/viewer/humaneval-ts/test?p=1&row=101

HumanEval #162, TypeScript Translation 2023. https://huggingface.co/datasets/nuprl/MultiPL-E/viewer/humaneval-ts/test?p=1&row=157

HumanEval #63, TypeScript Translation 2023. https://huggingface.co/datasets/nuprl/MultiPL-E/viewer/humaneval-ts/test?row=60

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*. http://dx.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. https://arxiv.org/abs/2406.00515

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. https://arxiv.org/abs/2402.19173

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* 34, 3 (2014), 103–104.

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. 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. CHAPTER 8 - Type Systems for Programming Languages. In *Formal Models and Semantics*. Amsterdam, 365–458. https://www.sciencedirect.com/science/article/pii/B9780444880741500135

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. OctoPack: Instruction Tuning Code Large Language Models. https://arxiv.org/abs/2308.07124

Niels Mündler, Jingxuan He, Slobodan Jenko, and Martin Vechev. 2023. Self-contradictory hallucinations of large language models: Evaluation, detection and mitigation. *arXiv preprint arXiv:2305.15852* (2023).

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2024. Code Agents are State of the Art Software Testers. *arXiv preprint arXiv:2406.12952* (2024).

Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the Impact of Programming Language Distribution. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*. 26619–26645. https://proceedings.mlr.press/v202/orlanski23a.html

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. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. 82:1–82:13. 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 *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

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*.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 522–538. https://doi.org/10.1145/2908080.2908093

Vipula Rawte, Amit Sheth, and Amitava Das. 2023. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922* (2023).

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/arXiv.2308.12950

Vince Szabo, Dominik Winterer, and Zhendong Su. 2024. Compilation Quotient (CQ): A Metric for the Compilation Hardness of Programming Languages. https://arxiv.org/abs/2406.04778

Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2024. Bugs in large language models generated code. *arXiv preprint arXiv:2403.08937* (2024).

Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. 2024. Code Less, Align More: Efficient LLM Fine-tuning for Code Generation with Data Pruning. https://arxiv.org/abs/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 *Typed Lambda Calculi and Applications*. Berlin, Heidelberg, 373–389.

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 *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. 172–184. https://doi.org/10.1145/3611643.3616271

Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2024. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. https://arxiv.org/abs/2308.10462

Brandon T. Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. https://arxiv.org/abs/2307.09702

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=mXpq6ut8J3

Quanjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. *arXiv preprint arXiv:2405.01466* (2024).

```
function sub_elem( tup1: number[], tup2: number[]
  ): number[] {
    const res = Array(tup1 [0]  - tup2[0] ).concat(tup1[1] - tup2[1] + 150 - 50 * 100 /
500 * 200000 / 500 / 500 + 4582 * 4599 / 10000 - 9090 / 1000 + 3200 / 2 / 3 / 45890 *
458901 + 458901 / 45890 - 9090 / 2 * 90 - 9090 / 2 / 3 / 45890 * 458901 + 458901 / ...
```

Figure 7: Compilations errors remain when the model does not terminate after a rejected token, like in this showcased sample for MBPP #273 by CodeLlama 34B.

A Limitations and Future Work

Some limitations of the presented approach remain, which present opportunities for future work.

Effort of Implementing Type-Aware Constraining Algorithm There is some amount of manual effort associated with implementing a completion engine as described in this work, and it has to be done for every language separately. However, we expect many features to transfer from our implementation to the base language L_B, as it did for TypeScript, significantly reducing the effort involved. Moreover, we believe that due to the impact on code synthesis, the effort may pay off, and future programming language developers may consider generally writing such incremental language compiler front-ends to aid code generation.

Remaining Compiler Errors Finally, we observe that while constrained decoding guarantees a valid result when the generation terminates, a considerable amount of compilation errors remain. We investigate their source and find that these are caused by the LLM not terminating within the given token or time limit. We find this to be caused by the model entering loops when forced to comply with an unexpected constraint. An example is depicted in Fig. 7. One approach to resolve this is by limiting the complexity of generated code and thus effectively forcing the model to stay withing given token bounds. A minor amount of errors remain due to timeouts. We hypothesize that loops are cause by forcing unexpected constraints on the model can be resolved orthogonally to our work by allowing the model to pursue alternative paths upon encountering constraints, as implemented by alternative sampling techniques like beam-search (Hokamp and Liu, 2017). Future work may add additional constraints on, e.g., the complexity of expressions generated by the LLM, to force stopping such unconstructive loops and steer the model more strictly.

B More details on the defined language

In this section, we provide more details on the syntax, expression, statement, and return type inference rules.

Expression Type Rules The type rules for L_B utilize the standard concept of a *type environment*, denoted as Γ, which is a collection of pairs $(x_i : T_i)$ of identifiers x_i and type T_i. The typing rules for the language are detailed in Figs. 8–10. These rules form a subset of the safe TypeScript rules outlined by Bierman et al. (2014), allowing us to leverage their soundness results.

An expression e is considered valid in L_B if it adheres to the type rules specified in Fig. 8. Type annotations T represent the syntactically matching type. Literal numbers, strings, and booleans are evaluated to their respective types (LIT_x). Identifiers x are evaluated based on the type according to Γ(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 (GROUP).

Operators have predefined signatures, denoted by $S_1 \circ S_2 : T$, such as $number + number : number$ or $T = T : T$; these signatures must be satisfied in well-typed expressions (OP). Function calls require all parameters to match specified types in the function's signature (CALL). Expression e of type T can have named members n, accessible via $e.n$, with the type determined by lookup(T, n), an auxiliary partial function that returns either the type of member n if it exists or the type `undefined` otherwise.

The operators and the lookup function are part of the type system definition. We examine prevalent languages such as Java, Go, and TypeScript to align their behavior with common patterns. Firstly, the lookup function may assign default members to every type. For any two types, T and S, the default members of T mirror those of S, with the exception that occurrences of S within the member definitions are syntactically replaced by T. For instance, a member function signature $() \to S$ would become $() \to T$. Secondly, for operators $\circ$ with signature $S_1 \circ S_2 : T$, the result type T must be either S_1 or a primitive type, such as *boolean* or *string*.

Statements and Type Environments Type environments are modified by statements as detailed in Fig. 9. 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 types 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, which can be formally tracked by duplicating all inference rules with a respective annotation; however, this duplication is omitted here for brevity. Statement blocks, if-then-else statements, and functions must maintain valid type environments consistent with the surrounding state without altering it (BLOCK, ITE, FUN). Lastly, empty statement sequences do not alter Γ(NOP), while sequences of statements propagate it along the execution (SEQ).

Return Types Function return types necessitate the inference of the correctness of the return statements, which is detailed in Fig. 10. Return statements must contain expressions matching the function's declared return type. Additionally, a value must be returned on every execution path. We use the notation $\Gamma \vdash \overline{s} : R$ to indicate that the sequence of statements $\overline{s}$ ensures a return value of type R. The return type of a return statement directly corresponds to the type of the returned expression (R-RET). In the case of declarations and expression statements, the return type of subsequent statements is considered (R-DECL, R-EXPR). For if-then-else blocks, both branches must return the same type (R-ITE-SELF), or the return type is determined by statements following the block (R-ITE-NEXT). This logic also applies to statement blocks (R-BLOCK-SELF, R-BLOCK-NEXT). In function definitions, the return type is determined by the type of the subsequent return statements, akin to expression statements. However, it is additionally required that the function body has a guaranteed return type, matching the function's declared return type.

C Detailed Prefix Automaton Definitions

In this section, we provide more detailed definitions and analysis of the automaton for L_B. We further assume the existence of type assignment compatibility between types, $T \leq G$, if a value of type T may be (type) safely stored in a variable of type G.

C.1 Basic Combination Automata

In this section, we introduce basic automata for concatenation and union of automaton-accepted languages.

Union The union $A_X \cup A_Y$ for two automata on distinct $\mathcal{S}_X$ and $\mathcal{S}_Y$ is defined as follows. We define $I = I_X \cup I_Y$, $Q = Q_X \cup Q_Y$.

$$f(S,t) := \begin{cases} f_X(S,t) & \text{if } S \in \mathcal{S}_X \\ f_Y(S,t) & \text{if } S \in \mathcal{S}_Y \end{cases}$$

Since the states are distinct, and we merely combine the transition functions of both automata, using (P4), that $L(A_X \cup A_Y) = L(A_X) \cup L_(A_Y)$. If both A_X and A_Y are prefix automata, this also holds for $A_X \cup A_Y$.

Concatenation For the concatenation automaton $A_X \oplus A_Y$ of states $\mathcal{S}_X$ and $\mathcal{S}_Y$, we define $I = I_X$, $Q = Q_Y$.

$$[\text{LIT}_{num}]\frac{}{\Gamma \vdash \backslash d+ : number} \quad [\text{LIT}_{str}]\frac{}{\Gamma \vdash "\backslash w*" : string} \quad [\text{LIT}_{bool}]\frac{}{\Gamma \vdash \text{true, false} : boolean}$$

$$[\text{IDENT}]\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \quad [\text{ANON}]\frac{\Gamma \vdash e : T}{\Gamma \vdash (\overline{p}) \text{ => } e : (\overline{p}) \Rightarrow T}$$

$$[\text{GROUP}]\frac{\Gamma \vdash e : T}{\Gamma \vdash (e) : T} \quad [\text{OP}]\frac{\Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2 \quad S_1 \circ S_2 : T}{\Gamma \vdash e_1 \circ e_2 : T}$$

$$[\text{CALL}]\frac{\Gamma \vdash f : (\overline{p}) \Rightarrow T \quad \Gamma \vdash \overline{e} : \overline{p}}{\Gamma \vdash f(\overline{e}) : T} \quad [\text{MEMBER}]\frac{\Gamma \vdash e : S \quad \text{lookup}(S, n) = T}{\Gamma \vdash e.n : T}$$

Figure 8: Expressions type inference rules in L_B.

$$[\text{DECL}]\frac{x \notin dom(\Gamma)}{\Gamma \vdash \text{let } x : T \rightarrowtail \Gamma \cup (x : T)} \quad [\text{EXPR}]\frac{\Gamma \vdash e : T}{\Gamma \vdash e \rightarrowtail \Gamma} \quad [\text{BLOCK}]\frac{\Gamma \vdash \overline{s_B} \rightarrowtail \Gamma_B}{\Gamma \vdash \{\ \overline{s_B}\ \} \rightarrowtail \Gamma}$$

$$[\text{RET}]\frac{\text{inside function body} \quad \Gamma \vdash e : T}{\Gamma \vdash \text{return } e; \rightarrowtail \Gamma} \quad [\text{FUN}]\frac{x, \overline{p} \notin dom(\Gamma) \quad \Gamma \cup (x : (\overline{p}) \Rightarrow T) \cup (\overline{p}) \vdash \overline{s_x} \rightarrowtail \Gamma_x}{\Gamma_1 \vdash \text{function } x\ (\ \overline{p}\ ) : T\ \{\ \overline{s_x}\ \} \rightarrowtail \Gamma \cup (x : (\overline{p}) \Rightarrow T)}$$

$$[\text{ITE}]\frac{\Gamma \vdash s_{if} \rightarrowtail \Gamma_{if} \quad \Gamma \vdash s_{else} \rightarrowtail \Gamma_{else}}{\Gamma \vdash \text{if } (\ e\ )\ s_{if} \text{ else } s_{else} \rightarrowtail \Gamma} \quad [\text{NOP}]\frac{}{\Gamma \vdash \bullet \rightarrowtail \Gamma} \quad [\text{SEQ}]\frac{\Gamma_1 \vdash \overline{s} \rightarrowtail \Gamma_2 \quad \Gamma_2 \vdash s \rightarrowtail \Gamma_3}{\Gamma_1 \vdash \overline{s} \quad s \rightarrowtail \Gamma_3}$$

Figure 9: Type environment extension rules for sequences of statements in L_B.

$$[\text{R-RET}]\frac{\Gamma \vdash e : R}{\Gamma \vdash \text{ return } e; \overline{s} : R} \quad [\text{R-DECL}]\frac{\Gamma \vdash \overline{s} : R}{\Gamma \vdash \text{let } x : T; \overline{s} : R} \quad [\text{R-EXPR}]\frac{\Gamma \vdash \overline{s} : R}{\Gamma \vdash e; \overline{s} : R}$$

$$[\text{R-ITE-SELF}]\frac{\Gamma \vdash s_{if} : R \quad \Gamma \vdash s_{else} : R}{\Gamma \vdash \text{if } (\ e\ )\ s_{if} \text{ else } s_{else} \quad \overline{s} : R} \quad [\text{R-ITE-NEXT}]\frac{\Gamma \vdash \overline{s} : R}{\Gamma \vdash \text{if } (\ e\ )\ s_{if} \text{ else } s_{else} \quad \overline{s} : R}$$

$$[\text{R-BLOCK-SELF}]\frac{\Gamma \vdash \overline{s_B} : R \quad \Gamma \vdash \overline{s}}{\Gamma \vdash \{\ \overline{s_B}\ \}\ \overline{s} : R} \quad [\text{R-BLOCK-NEXT}]\frac{\Gamma \vdash \overline{s_B} \quad \Gamma \vdash \overline{s} : R}{\Gamma \vdash \{\ \overline{s_B}\ \}\ \overline{s} : R}$$

$$[\text{R-FUN}]\frac{\Gamma \cup (x : (\overline{p} \Rightarrow R)) \vdash \overline{s} : R' \quad \Gamma \cup (x : (\overline{p}) \Rightarrow R) \cup (\overline{p}) \vdash \overline{s_x} : R}{\Gamma \vdash \text{function } x\ (\ \overline{p}\ ) : R\ \{\ \overline{s_x}\ \}\ \overline{s} : R'}$$

Figure 10: Statement return type inference rules of L_B.

$$f(S,t) := \begin{cases} f_X(S,t) & \text{if } S \in \mathcal{S}_X \backslash Q_X \\ f_X(S,t) \cup f_Y(I_Y,t) & \text{if } S \in Q_X \\ f_Y(S,t) & \text{if } S \in \mathcal{S}_Y \end{cases}$$

Informally, concatenation preserves the parsing behavior of both A_X and A_Y on their respective states. When A_X is in an accepting state, f allows both the first, "active" automaton A_X to continue parsing tokens from the alphabet and allows transitioning to A_Y. This maintains outgoing edges from accepting states in A_X while adding edges from accepting states of A_X to initial states of A_Y, similar to how PDAs and DFAs are concatenated.

It follows from a similar argument that $L(A_X \oplus A_Y) = L(A_X) \oplus L(A_Y)$. More formally, we can see $L(A_X \oplus A_Y) \subseteq L(A_X) \oplus L(A_Y)$ because of (P2) we can always split any s into a word w_X that extends from I_X in Q_X and the suffix w_Y. Then $w_X \in L(A_X)$ and $w_Y \in L(A_Y)$. For $L(A_X) \oplus L(A_Y) \subseteq L(A_X \oplus A_Y)$ we pick any take word $w_X w_Y$ from $L(A_X) \oplus L(A_Y)$ and feed into $A_X \oplus A_Y$, observing that it will first traverse from I_X to Q_X consuming w_X and from transition through I_Y to Q_Y by consuming w_Y.

Moreover, $L(A_X \oplus A_Y)$ is a prefix automaton, if A_X and A_Y are prefix automaton. For any state in $\mathcal{S}_X$ we know we can reach Q_X. At Q_X we may transition further as though starting at I_Y, from where we can always reach Q_Y. This construction is *only* a prefix automaton when $I_Y \neq \emptyset$, which, due to the prefix property, is equivalent to $L(A_Y) \neq \emptyset$.

Kleene-Plus We finally define the Kleene-Plus automaton A^+ that parses indefinite repetitions of words from a language.

$$f^+(S,t) := \begin{cases} f(S,t) & \text{if } S \notin Q \\ f(S,t) \cup f(I,t) & \text{if } S \in Q \end{cases}$$

We can quickly see that $L(A^+) = L(A)^+$, with basically the same argument as the concatenation case. We similarly see that this is a prefix automaton if A is a prefix automaton.

C.2 Terminals and Notation

Terminals Terminal automatons θ_x parse exactly the terminal x. They accept the usual alphabet Σ and feature the states $\mathcal{S} = \{S_s \mid s \leq x\}$, $Q = \{S_\epsilon\}$, $I = \{S_x\}$. f is defined as follows.

$$f(S_x,t) := \begin{cases} \{S_y\} & \text{if } t + y = x \\ \emptyset & \text{otherwise} \end{cases}$$

In the following we will implicitly assume that $f(X,t) = \emptyset$ if not explicitly defined otherwise, making notation more concise. Clearly, θ_x is a prefix automaton. We can show inductively that for any x: $r(\theta_x, s) = \{\theta_\epsilon\} \iff s = x$, and thus $L(\theta_x) = \{x\}$.

With a simple modification we introduce θ_x^W, where $f(\theta_x^W, w) := \{\theta_x^W\}$ (W denoting whitespace characters, $w \in W$) and $f(\theta_{t+s}^W, t) := \{\theta_s\}$, which allows arbitrary whitespace before parsing x i.e. parses $r/\backslash s^* x/^p$. This is how we implement syntactic indifference to whitespace between tokens.

Empty Automaton The empty automaton $A_\emptyset$ is the only valid prefix automaton that accepts no word. Due to the prefix condition, it has no states.

Notational Details For any state, we access the following information through dot notation or the special notation on the state, which we assume is passed onto subsequent states through the transition function (unless otherwise stated). This information is alternatively passed through to entire automata i.e. in concatenated automata.

1. $X \downarrow T$: The type to which the automaton is constrained (introduced in more detail later)
2. $X \in Q_Y$: Whether the state is an accepting state of the automaton A_Y

3. $X.\Gamma$: The currently valid identifiers/type environment according to the expression being currently parsed.

4. X.lhs: The left-hand side of an extending expression (introduced in more detail with the expression and type annotation automaton)

5. X.typ: The described type of the last coherent expression that this state belongs to. Only defined for accepting states. Generally, we ensure that when some expression e was parsed, the corresponding state S_e has attribute S_e.typ such that $S_e.\Gamma \vdash e : S_e$.typ.

Further, to correctly handle function return types, we pass on related information when entering function bodies:

1. $X.R$: The currently expected return type

2. X.returned: Whether the currently parsed program block has returned in all branches

3. X.mustReturn: Whether the currently parsed program block must return (i.e. "ITE" branches do not need to contain return statements even if a return type is expected of the surrounding code block)

Further, we write concatenations of existing automata by the shorthand of $C_{X \dots Y} = A_X \oplus \dots \oplus A_Y$. Where clear from context, these parameters may also refer to terminals, i.e. for $C_{\{X\}} = \theta_{\{} \oplus A_X \oplus \theta_{\}}$. We use the overline as shorthand for Kleene-star automata, i.e. $A_{\overline{s}} = A_s^+$. This is especially useful in combination with the above concatenation shorthand.

We assume that knowledge about previously parsed states in the concatenation automaton is preserved in states. We access relevant final states of concatenated automata by using their shorthand in the concatenation automaton, i.e. we can access the last accepting state $S_Y \in Q_Y$ of automaton A_Y, when the active state $S \in \mathcal{S}_{C_{XY}}$ was originally part of $\mathcal{S}_Y$.

The following automata are implemented as subclasses to classes implementing concatenation, union and Kleene-Star, which each store the respectively relevant information.

C.3 Expressions

Expressions are parsed using co-recursive automatons. For this, we define the extension automaton A_e and co-recursively the automata $A_{(e)}$, $A_{[\overline{e}]}$, $A_{\circ e}$, $A_{[e]}$, $A_{.n}$ and $A_{(\overline{e})}$ which are mainly defined by the concatenation of the respective terminal automata and (recursively) A_e, with some variation described in more detail below. Assuming we have automata A_x and A_{lit} for identifiers and literals. $I_e = I_{lit} \cup I_x \cup I_{(e)} \cup I_{[\overline{e}]}$. Note this set is finite because $I_{(e)} = \{S_(\}$ and $I_{[\overline{e}]} = \{S_[\}$. The transition function is defined for all states $S_x \in \mathcal{S}_X$ ($X \in \{(e), [\overline{e}], \circ e, [e], .n, (\overline{e})\}$), where we denote via superscript how the lhs parameter is passed on.

$$f_e(S_X, t) \coloneqq \begin{cases} f_X(S_X, t) \cup f_e(I^X_{(e)}, t) \cup f_e(I^X_{\circ e}, t) \cup f_e(I^X_{[e]}, t) \cup f_e(I^X_{.n}, t) & \text{if } S \in Q_e \\ f_X(S_X, t) & \text{otherwise} \end{cases}$$

Similarly to extendable types, we accommodate for expressions extending other expressions. For example, `123` allows as next tokens not only digits, like `4` to form `1234` (as A_l would), but also, because `123` is an accepted literal, would allow $\circ$ as next token, initiating an arithmetic expression like `123 + 4`. The language accepted by this automaton is the language described by the syntactic rules in Fig. 2. In expression states, the lhs attribute is crucial to accurately evaluate the `typ` attribute, which again is required to define the remaining recursive automata. In the case of expressions, the `typ` attribute expresses the type of the parsed expression.

$$
\begin{aligned}
S_{L_{string}}.\text{typ} &\coloneqq \text{string} \\
S_{L_{number}}.\text{typ} &\coloneqq \text{number} \\
S_{L_{boolean}}.\text{typ} &\coloneqq \text{boolean} \\
S_x.\text{typ} &\coloneqq T \text{ where } S.\Gamma \vdash x : T \\
S_{(e)}.\text{typ} &\coloneqq e.\text{typ} \\
S_{[\overline{e}]}.\text{typ} &\coloneqq \overline{e}[0].\text{typ}[] \\
S_{[e]}.\text{typ} &\coloneqq M \text{ for lhs.typ} = M[] \\
S_{(\overline{e})}.\text{typ} &\coloneqq R \text{ for lhs.typ} = (\overline{P}) \Rightarrow R
\end{aligned}
$$

Now we have most of the required tools to define the recursive automata. The grouped expression is just concatenation, $A_{(e)} \coloneqq C_{(e)}$. The main functionality of the array expression is based on $C_{[\overline{e}]}$, however, the automaton is actually a union of three distinct automata:

$$
A_{\text{array}} \coloneqq \bigcup \begin{cases} C_{[]} \\ C_{[e]} \\ C_{[e} \oplus C_{\overline{e}} \downarrow T \oplus C_{]} \text{ for } e.\text{typ} = T \end{cases}
$$

This way, the automaton can constrain the type of subsequent expressions in the same array. The prefix property is not violated: The first two cases are always valid, in the third case, type T can clearly be expressed (as demonstrated in the first parsed expression), hence further occurrences are possible (the type environment can not be modified within expressions).

The arithmetic operator type constrains its states to those with valid operators, i.e.

$$
A_{\circ e} \coloneqq \bigcup_{\exists T.\ \text{lhs.typ} \circ S = T} \theta_{\circ} \oplus A_e \downarrow S
$$

The computed member access also depends on the parsed `lhs`:

$$
A_{[e]} \coloneqq \begin{cases} \theta_{[} \oplus A_e \downarrow number \oplus \theta_{]} & \text{if } \exists M.\text{lhs.typ} = M[] \\ A_{\emptyset} & \text{otherwise} \end{cases}
$$

.

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

$$
A_{(\overline{e})} \coloneqq \begin{cases} \theta_{(} \oplus A_{\overline{e}} \downarrow \overline{P}.typ \oplus \theta_{)} & \text{if lhs.}typ = (\overline{P}) \Rightarrow \bullet \\ A_{\emptyset} & \text{otherwise} \end{cases}
$$

Finally, member access depends on the attributes of the types in the lhs, $A_{.n} = A_x^{\Gamma(\text{lhs.typ})}$, where $\Gamma(Y)$ is the type-environment induced by all attributes of type Y. These definitions use several concepts that need to be properly introduced, and are so in the following paragraphs.

Tracking Type Environments We introduce A_x^{Γ}, the automaton for parsing identifiers, where $\Gamma : \mathcal{V} \mapsto \mathcal{T}$ (V and T are the set of valid variable names and types respectively, need to define them somewhere). It is defined as $A_x^{\Gamma} = \bigcup\{\theta_v \mid v \in \mathcal{D}(\Gamma)\}$, i.e. the union of all terminal automata for the respective variables in the type environment. The definition set $\mathcal{D}(\Gamma)$ is finite. To enforce that only defined identifiers are used, we use the attribute Γ that is passed down to all recursively invoked automata unless otherwise denoted (and has no effect if the automata is not A_x). This introduces the first non-syntactic language feature that our automaton can correctly parse in an incremental fashion. Note that we need not ensure that $\mathcal{D}(\Gamma) \neq \emptyset$ as we would for normal concatenation automata, because $I_E \neq \emptyset$ - we can always express a type as a literal expression.

Restricting Types We can further restrict an expression to a specific type, for example when we want to describe an expression that has number type for computed member access. Similarly to passing down the type environment, we pass the restricting type T by generating the automata $A \downarrow T$. Generally, this process needs to be performed carefully so as not to violate the prefix automaton property. Concretely, restricting types may turn accepting states into non-accepting states when the respectively represented expression does not match the expected type. To prevent thus accidentally constructing a non-prefix automaton we need to determine whether an automaton has any accepting states ahead of time. For this we define $\neg\emptyset(A, T)$, which determines whether $L(A \downarrow T) \neq \emptyset$.

For variables, we adjust to track the type of terminals by passing it as a parameter to the terminal itself, which has no effect on the terminal automaton per se: $A^{\Gamma}{}_{x} := \bigcup\{\theta_x^T \mid \Gamma \vdash x : T\}$. For the base automata and operations $\neg\emptyset$ is further defined:

$$
\begin{aligned}
\neg\emptyset(L_{number}, T) &:= \text{number} \leq T \\
\neg\emptyset(L_{string}, T) &:= \text{string} \leq T \\
\neg\emptyset(L_{boolean}, T) &:= \text{boolean} \leq T \\
\neg\emptyset(\theta_x, T) &:= true \\
\neg\emptyset(\theta_x^G, T) &:= G \leq T \\
\neg\emptyset(A \oplus B, T) &:= \neg\emptyset(A, T) \wedge \neg\emptyset(B, T) \\
\neg\emptyset(A \cup B, T) &:= \neg\emptyset(A, T) \vee \neg\emptyset(B, T)
\end{aligned}
$$

We can further automatically derive $A \downarrow T$ for these cases (where $A_\emptyset$ is an automaton with no states, accepting the empty language):

$$
\begin{aligned}
L_{number} \downarrow T &:= L_{number} \text{ if number } \leq T \text{ else } A_\emptyset \\
L_{string} \downarrow T &:= L_{string} \text{ if string } \leq T \text{ else } A_\emptyset \\
L_{boolean} \downarrow T &:= L_{boolean} \text{ if boolean } \leq T \text{ else } A_\emptyset \\
\theta_x \downarrow T &:= \theta_x \\
\theta_x^G \downarrow T &:= \theta_x^G \text{ if } G \leq T \text{ else } A_\emptyset \\
A \oplus B \downarrow T &:= A \downarrow T \oplus B \downarrow T \text{ if } \neg\emptyset(A, T) \wedge \neg\emptyset(B, T) \text{ else } A_\emptyset \\
A \cup B \downarrow T &:= A \downarrow T \cup B \downarrow T \text{ if } \neg\emptyset(A, T) \vee \neg\emptyset(B, T) \text{ else } A_\emptyset
\end{aligned}
$$

Effectively, this propagates the type requirement down to the base cases and makes the whole automaton reject pre-emptively if it can be determined that no completion will be possible, thus preserving the prefix automata property.

However, for expressions, it is more difficult - even if the intermediate expression is not of the desired type, we need to take into account types reachable by extension. For this, we use the type reachability algorithm.

We define $\neg\emptyset(A_e \downarrow T)$, i.e. whether the currently parsed expression has a valid completion that will fit into type T. Generally, the rule is to use DERIVABLE and the type reachability algorithm to determine whether a suitable type can be reached, i.e. $\neg\emptyset(A_e \downarrow T) = \text{REACHABLE}(\text{DERIVABLE}(e), T)$ for expression (sub) automaton A_e. The parameters for each expression depend on specifics to the operation, i.e. for arithmetic operations, they define which operators are applicable to the current type.

C.4 Complete reachability algorithm

The formal definitions for depth and root types can be found below:

$$
\text{DEPTH}(T) := \begin{cases} \text{DEPTH}(X) + 1 & \text{if } T = (\overline{p}) \Rightarrow X \\ 0 & \text{otherwise} \end{cases}
$$

Algorithm 2 Type Reachability Algorithm

Input: Current Type T, Goal Type G, Maximum depth d, Root types r
Output: Whether G can be reached by adding tokens to T

function REACHABLE(T, G, d, r)
  if T in progress **then return** false
  elif $T = G$ **then return** true
  mark T in progress
  $next := \{\}$
  for operator $\square$ from OP, CALL, MEMBER **do**
    $next := next \cup \text{TYPES}(T, \square)$ ▷ Collect types reachable through extension
  for T' **in** $next$ **do**
    if $\text{DEPTH}(T') > d \wedge \text{ROOT-TYPES}(T') - r = \emptyset$ **then** ▷ Prevent recursing too deeply
      continue
    if $\text{REACHABLE}(T', G, \max(d, \text{DEPTH}(T')), r \cup \text{ROOT-TYPES}(T'), N)$ **then** ▷ Recurse
      return true
  return false ▷ No suitable extension was found

$$\text{ROOT-TYPES}(T) := \begin{cases} \text{ROOT-TYPES}(X) & \text{if } S = (\overline{p}) \Rightarrow X \\ \{T\} & \text{otherwise} \end{cases}$$

The search algorithm described in §3.4, REACHABLE, is presented in Algorithm 2. The maximum depth d is initialized by default to $\max(\text{DEPTH}(T), \text{DEPTH}(G))$, and root types r are initialized to $\text{ROOT-TYPES}(T)$. All applicable operators for a given type include member access, computed member access, and function calls. When invoking the reachability algorithm with a set of types $\mathcal{T}$, it returns whether any type T within the set reaches G, i.e., $\text{REACHABLE}(\mathcal{T}, G) = \bigvee_{T \in \mathcal{T}} \text{REACHABLE}(T, G)$.

C.5 STATEMENTS

We define the remaining automata to capture the complete language from §3.1.

The single statement automaton is another co-recursive definition, since substatements (i.e. ITE) can themselves contain statements. $A_s := A_{e;} \cup A_{decl} \cup A_{block} \cup A_{ITE} \cup A_{fun} \cup A_{ret}$. The expression statement automaton and block automaton are simply defined as $A_{e;} := C_{e;}$ and $A_{block} := C_{\{\overline{s}\}}$. The declaration automaton $A_{decl} := C_{let x:T;}$ 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: $A_{ITE} := C_{\text{if}(e)\ s \text{ else } s}$.

The statements automaton $A_{\overline{s}} := A_s^+$, is based on the 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:

$$A_{ret} = \begin{cases} C_{\text{return } e \downarrow T} & \text{if } A_{ret}.R = T \\ A_{\emptyset} & \text{otherwise} \end{cases}$$

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

1. $X.R = T$, 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.
2. X.mustReturn = true, for the outermost statement block automaton. It is set to false for deeper nested statement blocks and as soon as a parsed statement has X.returned set to true - i.e. one of the main body statements returned in every branch.

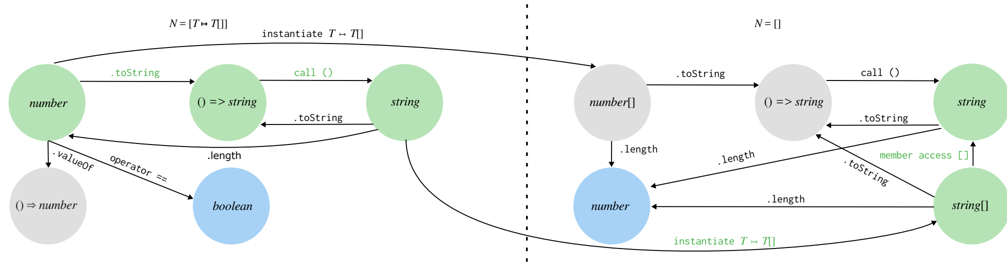

Figure 11: A longer example search through the graph for type reachability, starting from $t = number$ with the goal $string$ and $N = [T \mapsto T[]]$. This search could be the result of parsing, e.g., `let x : string; x = [1`. Nodes and paths along the final path are marked green, explored nodes marked blue. Note how the $string$ node on the left is not a valid reached node, since $N \neq []$. Node $() \Rightarrow string$ is explored even though it has larger depth ($(1, 0) > (0, 0)$) because it provides access to a new root type $string$.

3. X.returned $=$ false, per default in every statement, except a) in return automata, b) inside a multi-statement automaton where the previous statement has $returned = true$ and c) in ITE-automata where both branching statements have $returned = true$

As long as a multi-statement automaton X has $X.returned = false$ and $X.mustReturn = true$, it can not accept but instead forces the generation of another statement. Since we can always express the requested type (i.e. through literals) and can always simply generate a return statement to fulfill this requirement, the prefix automaton property is not violated.

Handling identifiers Generally, identifiers are passed on through all state transitions, matching the implicit inference rules for statement concatenation in Fig. 9, where the type environment of consecutive statements needs to be compatible. For example, the transition function for $A_{\overline{s}}$ is an instantiation of the standard plus-rule as follows:

$$f_{\overline{s}}(S^\Gamma, t) := \begin{cases} f_s(S^\Gamma, t) & \text{if } S^\Gamma \notin Q \\ f_s(S^\Gamma, t) \cup f_s(I^\Gamma, t) & \text{if } S^\Gamma \in Q \end{cases}$$

However, matching the type rules of statement block automaton A^Γ_{block} initialized in a context with environment Γ and which discards any updates to Γ inside the block, the rules for blocks deviate from standard automaton $C_{\{\overline{s}\}}$. Informally, the original type environment is copied into the sub-expression and restored when closing the braces of the block statement in automaton $\theta_{\}}$, formally $I^\Gamma_{block} = \{S^\Gamma \mid S \in I^\Gamma_B\}$ and

$$f^\Gamma_{block}(S^{\Gamma_b}, t) := \begin{cases} f_B(S^{\Gamma_b}, t) & \text{if } S^{\Gamma_b} \in \mathcal{S}_B \backslash \mathcal{S}_{\}} \\ \{S^\Gamma_{next} \mid S_{next} \in f_B(S^{\Gamma_b}, t)\} & \text{if } S^{\Gamma_b} \in \mathcal{S}_{\}} \end{cases}$$

Similarly, the types after exiting the function body are initialized with the function type and parameters added, and the surrounding type environment restored after exiting the body.

C.6 Arrays

Array types require modification to the reachability algorithm. Concretely, they require tracking both nestedness in terms of higher-order-function-ness and array-nestedness, because we can always map any array type to a deeper nested array type using `map`. The depth of an expression is then a tuple of two integers and we can skip exploring some type if its depth is larger in either dimension.

The root types of array types are defined as ROOT-TYPES$(T[]) =$ ROOT-TYPES(T).

Moreover, array types require the introduction of the nesting parameter to the reachability function The design of the reachability algorithm as search, in general, improves performance as opposed to an exhaustive enumeration of reachable types, however makes it a bit more involved to consider, e.g., array expressions `[e ....` What types can be reached from such an expression? Intuitively, for any type T reachable from `e`, we can reach $T[]$ from `[e]` - and then again all types reachable from $T[]$. In the search, we implement this by adding the *nested* parameter to the search, which acts as a layer in the search graph. From any type that can be reached, we can move one layer up, or equivalently remove the first element f from the nested parameter by instantiating the pattern $f(T)$ with the currently reached type T. We start in the deepest layer based on the current expression and can not return `true` without traversing through all layers. Such a search is visualized in Fig. 11. The addition to Algorithm 2 consists of a single line after Line 2, dubbed Line 2.5, "**elif** $f, \dots N' := N$ **then return** $\text{REACHABLE}(f(T), G, d, r, N')$". This line ensures that the goal type is not considered found as long as the nested parameter is not empty.

The nested parameter for the invocation in restricted expressions is implicitly defined by the recursive use of REACHABLE inside the DERIVABLE function. We add cases for partial array expressions and array member access as shown in Table 3. For started array expressions, e.g. $[\ e$, the derivability algorithm becomes impractical to compute since it would require enumerating all reachable types of the subexpression. We circumvent this issue by adding an element in front of the nested parameter N.

Table 3: Extension to the derivability function for array expressions.

x	DERIVABLE(x)
$[\ e$	$\{T[] \mid \text{REACHABLE}(\text{DERIVABLE}(e), T)\ \}$
$[\ e, \overline{e}$	$\{T[] \mid \Gamma \vdash e : T\}$
$e\ [$	$\{E \mid \Gamma \vdash e : E[]\}$

Lemma 6. *For some type transformation $f : \mathcal{T} \mapsto \mathcal{T}$, start type S, goal type G, and nested list N the following holds:* $\text{REACHABLE}(\{f(T) \mid \text{REACHABLE}(S, T)\ \}, G, N) = \text{REACHABLE}(S, G, N = [f] + N)$

Proof. As we can see in Algorithm 2 it is not possible for the reachability to terminate until the first element of N is removed and its instantiation executed: After adding the additional line 2.5, Line 3 can not be reached without removing the head element of N and recursing into it, and the recursion in Line 12 carries over the current value of N. Moreover, once line 2.5 is executed, the current type T is by definition reachable from S and thus reachable(S, T). In the recursive call, the same process is performed as though starting the reachability from $f(T)$. □

Lemma 6 can be used for transformations such as the array wrapping $T \mapsto T[]$ and type parameter instantiation for polymorphic types in the TypeScript standard library such as `map`. In the case of $[e$ (pictured in Fig. 11), we instantiate as usual and then use Lemma 6 to simplify the computation.

$$\begin{aligned}
\text{REACHABLE}(\text{DERIVABLE}([e), &G, d, r, N) \\
&= \text{reachable}(\text{DERIVABLE}([e), G, d, r, N) \\
&= \text{REACHABLE}(\{T[] \mid \text{REACHABLE}(\text{DERIVABLE}(e), T)\}, G, d, r, N) \\
&= \text{REACHABLE}(\text{DERIVABLE}(e), G, d, r, [T \mapsto T[]] + N)
\end{aligned}$$

D Extension to TypeScript

To enable type-aware constraining for the TypeScript language, we now introduce and shortly discuss the implementation of additional features beyond L_B.

Union Types Union types naturally arise in the return types of i) if-then-else blocks, ii) logical operators, and iii) the ternary operator. For instance, the expression `x ? 0 : ""` yields a return type of $number \mid string$. To handle them in the reachability algorithm, we define the root types of union types as $\text{ROOT-TYPES}(T \mid S) := \text{ROOT-TYPES}(T) \cup \text{ROOT-TYPES}(S)$, and consider the maximum depth of each element. Union types also allow type expressions to be extensible, i.e., allowing to append $\mid T$ to some declared type. We adapt the type expression automaton as was done for A_e. However, since there are no type restrictions on type annotations, no reachability algorithm is required.

Furthermore, the ability to declare variables with union types necessitates the introduction of assignment compatibility, which differs from type equality. Specifically, when restricting to a type $T \mid S$, expressions of either type T or S are valid to inhabit. To formalize this relationship, we define compatibility between type assignments using the operator $\geq$, following the rules established by Bierman et al. (2014). Consequently, we adjust the mechanisms of type reachability and type constraints in automata to use compatibility rather than strict equality.

Array Types To accommodate array types, the DEPTH function is extended to a two-tuple, which includes the function order and the dimensionality of the array. The root types of an array are the root types of its elements. The search does not explore when either element of this tuple exceeds the current depth. Additionally, DERIVABLE is expanded to handle partial expressions. The complete extension of the reachability and derivability algorithms is detailed in App. C.6.

Loops TypeScript supports various loop constructs, including `for`, `while`, `do-while`, and `for..in` loops. They have standard implementations, with restricted variable scopes of body and head. The `for..in` loops uniquely constrain the right-hand side of the `..in` operator to any array type. To accommodate this, we introduce the generic array $\bullet[]$, which is assignment-compatible with any array type and restricts the expression to the right-hand side of `in`.

Operator Precedence Determining the admissibility of operators depends heavily on the operator precedence. For instance, writing `"abc" + 1 .length` is invalid because operator precedence dictates that `1.length` is evaluated first due to the stronger binding of the member access operator `.`. Similarly, `1 + true ? 1 : 2` is invalid, since the ternary `?` operator has a weaker binding than the `+` operator, causing `1 + true` to be evaluated first. In terms of the syntax tree, the resulting restrictions are best visualized as preventing operators from "escaping" the current node, ensuring they bind only within the current node in the tree. This is enforced by setting upper and lower bounds on the operator precedence for admissible operators.

Global Variables and Imports In TypeScript, many variables are initialized in the global scope. These are incorporated by modifying the type environment of the main language automaton. Variables such as `Math` introduce additional types, which we handle individually and consider as root types. We also model the import of the `Crypto` library using `require`.

Returning Void Functions may be annotated with a return type of void, the default if no return type is specified. Void functions are not required to return, and may include `return;` statements without expressions. However, return statements with values are forbidden.

Type Casts If a type is a union type with a *falsy* type, which always evaluates to false, it is typecast by comparisons. For instance, in the code `if(x) A else B`, if `x` is of type $number \mid undefined$, $\Gamma$is updated such that in `A`, `x` is of type $number$, and in `B`, it is of type $undefined$. We recursively extract such type casts from comparative statements and update the type environment for in corresponding control flow.

Polymorphic Standard Library Functions TypeScript code makes frequent use of polymorphic types in members of built-in types. For example, the member function `map` of array types `T[]` has the type $((T) \Rightarrow P) \Rightarrow P[]$ where `P` is a type parameter. We support such polymorphisms by the DERIVABLE function to the instantiated version. In the `map` example, deriving type $(T) \Rightarrow P$ in the parameter will result in overall derivable type $P[]$, i.e. the reached type has to match the pattern of the first parameter of the map function and will allow deriving an array of its return type.

Array Type Inference TypeScript allows empty-array initialization, deriving the element type through parameters to polymorphic operations such as `push`, which appends an element to the array. The corresponding expression changes the type of the receiver array, even inside statement blocks. We carefully trace such cases and propagate respective type-environment influences correctly.

Automatic Semicolon Insertion We implement a close approximation to automatic semicolon insertion in the parsing algorithm, following the rules defined in the ECMA Script (Ecma International, 2016). Concretely we implement rules 1 and 3 in the parser by (i) after reaching any rejected state by encountering a closing brace or a newline, re-trying parsing by injecting a semicolon and (ii) before processing a newline after the last word in the input is `continue`, `return`, `break` or `throw`. Rule 2 requires no additional implementation.

Other features introduced for extensive support of TypeScript are *optional function parameters*, *rest parameters* and *tuple types*, which similarly require simple adaptions to the automatons and derivability function.

D.1 Proof of Soundness of the Type Search Algorithm

It follows the proof of Lemma 3. The core idea is to show a) if expression e inhabits type T, the types discovered by search over operator signatures are exactly the inhabitable types by syntactically and semantically valid extensions to e and b) the restricted graph search of the reachability algorithm discovers only types reachable from T via search over operator signatures.

Concretely our type graph is spanned by edges from type T to type R, if it is valid to apply operator $\circ$ to T with signature $T \circ S : R$. Note that we use this to model all operations on expressions, i.e., OP, CALL and MEMBER, where the respective signature for calls is $T()\overline{P} : R$ and members is $T. : R$ with more or 0 operands respectively.

For a), the search over operator signatures is an overestimation, and thus whenever an extension $\circ e'$ exists with $\Gamma \vdash e' : S$ and return type $\Gamma \vdash e \circ e' \vdash R$, we discover R by taking the edge from T to R spanned through $T \circ S : R$. For the reverse direction, we observe that we can express every type of L_B as a literal. Therefore can express all additionally required operands for every operator. For every operator $\circ$ with signature $T \circ S : R$, we can extend e using by $\circ e'$ where e' is a literal of type S to express R.

For b) we prove $\text{REACHABLE}(T, G) \implies$ there exists a path on the type graph between T and G. The reachability algorithm implements a depth-first search over the type graph (Even, 2011), by taking edges according to the admissible operators as described before. The only modification is a restriction, on which edges are taken based on computed depth and root types. Therefore the search is sound, returning only true if there is a sequence of edges in the type graph between T and G.

E More experimental details

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.

E.1 Hyperparameters and Compute

We run all models and evaluations on NVIDIA A100 GPUs. We use temperature sampling to incrementally generate completions with inference temperature for all runs set to 1. To ensure reproducibility, the four runs of §4 are executed with fixed seeds (0, 1, 2, and 3).

E.2 Details for Methods

To compute Syntax, we determine the subset of programs that are syntactically incorrect using the Oxidation toolchain (oxc project, 2024). We compute an upper bound on the performance of syntactic constraining and assume that every instance with syntax error would compile successfully under syntactic constraining. Due to the massive size and complexity of the TypeScript compiler, featuring over 427,105 lines of code in 698 files (Microsoft, 2024), it is improbable that our extension covers all features of TypeScript. To provide a realistic evaluation in Types, we emulate a type-aware constraining that supports the entire TypeScript feature set. We sample constrained and unconstrained in parallel, using the same seed, and report the unconstrained output if it successfully compiles and otherwise the constrained version.

E.3 Excluded MBPP Instances

We exclude the following MBPP instances as the auto-translation of MultiPL-E produced misleading type annotation for the function arguments:

- `mbpp_143_find_lists`
- `mbpp_222_check_type`
- `mbpp_240_replace_list`
- `mbpp_262_split_two_parts`
- `mbpp_265_list_split`
- `mbpp_284_check_element`
- `mbpp_390_add_string`
- `mbpp_398_sum_of_digits`
- `mbpp_405_check_tuplex`
- `mbpp_418_Find_Max`
- `mbpp_425_count_element_in_list`
- `mbpp_431_common_element`
- `mbpp_446_count_Occurrence`
- `mbpp_457_Find_Min`
- `mbpp_563_extract_values`
- `mbpp_580_extract_even`
- `mbpp_612_merge`
- `mbpp_725_extract_quotation`
- `mbpp_730_consecutive_duplicates`
- `mbpp_744_check_none`
- `mbpp_778_pack_consecutive_duplicates`
- `mbpp_791_remove_nested`

E.4 Extracting Executable Code

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.

We first try to extract the corresponding fenced typescript code block (`` ```typescript...``` ``). If the block is not closed, we consider all code until the end of the generation as part of the code block. 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. 12.

E.5 Speeding Up Decoding

Constrained decoding, sketched in Algorithm 1, is usually defined by computing a complete mask for the model vocabulary at every step and using it to mask out invalid tokens (Poesia et al., 2022). For non-zero temperatures, this approach is equivalent to sampling repeatedly and rejecting invalid tokens. We choose this alternative implementation as a default, as token masks have to be computed during sampling, and can not be deferred to pre-inference computation as done in previous work (Ugare et al., 2024; Beurer-Kellner et al., 2024).

However, for the case that the constraints are very tight and the model suggested token is not among the first k sampled tokens, we utilize the speed up that can be obtained by computing the mask in bulk for the entire vocabulary, using the trie data structure, as described by Poesia et al. (2022). The resulting refined algorithm is presented in Algorithm 3 Since the computation of the mask and the computation of logits are independent, even further optimization can run them in parallel and run the individual checking loop between Lines 6 and 9 until the mask has finished computing.

```
```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 12: 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

Algorithm 3 Fast Constrained Decoding

Input: Completion Engine CE_L, LLM M, Prompt x, k
Output: Completion o such that $o \in L$

1: $o := []$
2: CE_L.init(M.vocabulary, x)
3: **loop**
4: $\quad \boldsymbol{v} := M(x + o)$ ▷ compute logits
5: $\quad valid := false$
6: $\quad$ **for** k times **do**
7: $\quad\quad t := sample(\boldsymbol{v})$ ▷ e.g., argmax or sample with temperature
8: $\quad\quad valid := CE_L$.check(t)
9: $\quad\quad$ **if** $valid$ **then break**
10: $\quad$ **if** not $valid$ **then**
11: $\quad\quad \boldsymbol{m} := CE_L$.mask() ▷ compute mask
12: $\quad\quad \boldsymbol{v}' := \boldsymbol{v} \odot \boldsymbol{m}$ ▷ apply mask
13: $\quad\quad t := sample(\boldsymbol{v})$ ▷ e.g., argmax or sample with temperature
14: $\quad$ **if** $t = EOS$ **then break**
15: $\quad o$.append(t)
16: $\quad CE_L$.update(t)
17: **return** o

Table 4: Instances with compiler errors in Standard, Syntax, and our Types constraining on MBPP. Type-aware constraining reduces errors by 62.4% in Synthesis of MBPP, compared to only 5.7% through ideal syntax-only constraining. For Translation and Repair we observe 59.6% and 57.8% improvement due to type-aware constraining respectively.

	Synthesis			Translation			Repair		
Model	Standard	Syntax	Types	Standard	Syntax	Types	Standard	Syntax	Types
Gemma 2 2B	69	$67_{\downarrow 2.9\%}$	$\mathbf{25}_{\downarrow 63.8\%}$	132	$114_{\downarrow 13.6\%}$	$\mathbf{77}_{\downarrow 41.7\%}$	230	$219_{\downarrow 4.8\%}$	$\mathbf{109}_{\downarrow 52.6\%}$
Gemma 2 9B	32	$30_{\downarrow 6.2\%}$	$\mathbf{9}_{\downarrow 71.9\%}$	70	$64_{\downarrow 8.6\%}$	$\mathbf{28}_{\downarrow 60.0\%}$	179	$171_{\downarrow 4.5\%}$	$\mathbf{66}_{\downarrow 63.1\%}$
Gemma 2 27B	23	$22_{\downarrow 4.3\%}$	$\mathbf{5}_{\downarrow 78.3\%}$	43	$42_{\downarrow 2.3\%}$	$\mathbf{18}_{\downarrow 58.1\%}$	126	$109_{\downarrow 13.5\%}$	$\mathbf{48}_{\downarrow 61.9\%}$
DeepSeek C. 33B	32	$31_{\downarrow 3.1\%}$	$\mathbf{17}_{\downarrow 46.9\%}$	38	$35_{\downarrow 7.9\%}$	$\mathbf{12}_{\downarrow 68.4\%}$	156	$152_{\downarrow 2.5\%}$	$\mathbf{69}_{\downarrow 56.1\%}$
CodeLlama 34B	85	$76_{\downarrow 10.6\%}$	$\mathbf{42}_{\downarrow 50.6\%}$	132	$115_{\downarrow 12.9\%}$	$\mathbf{51}_{\downarrow 60.6\%}$	193	$180_{\downarrow 6.7\%}$	$\mathbf{89}_{\downarrow 53.9\%}$
Qwen2.5 32B	70	$65_{\downarrow 7.1\%}$	$\mathbf{25}_{\downarrow 62.9\%}$	35	$31_{\downarrow 11.4\%}$	$\mathbf{11}_{\downarrow 68.6\%}$	133	$124_{\downarrow 6.8\%}$	$\mathbf{54}_{\downarrow 59.4\%}$

Table 5: Pass@1 (in %) of unconstrained (Standard) and type-aware constrained (Types) generated code for the tasks Synthesis, Repair, and Translation on MBPP.

	Synthesis		Translation		Repair	
Model	Standard	Types	Standard	Types	Standard	Types
Gemma 2 2B	39.4	**41.8**	51.4	**57.3**	14.1	**29.3**
Gemma 2 9B	64.9	**67.7**	71.7	**78.3**	22.8	**37.3**
Gemma 2 27B	70.1	**72.3**	80.7	**84.8**	33.1	**46.3**
DeepSeek Coder 33B	65.8	**68.2**	83.4	**88.3**	25.7	**43.4**
CodeLlama 34B	40.8	**44.6**	54.1	**64.1**	19.3	**32.5**
Qwen2.5 32B	65.5	**74.5**	84.5	**89.4**	35.0	**52.7**

F Experimental Evaluation on MBPP

We run the same experiment described in §4 on the TypeScript translation of the dataset MBPP, provided in MultiPL-E Austin et al. (2021); Cassano et al. (2023). Due to the larger size of MBPP, we run the experiment only once with seed 0 and report the aggregated results in Table 4. For Repair, we collect all non-compiling programs from unconstrained synthesis for all models, resulting in 311 non-compiling programs.

F.1 Experimental Results

Reduction of Compilation Errors In our evaluation on MBPP, shown in Table 4, on average only 7.2% of compiler errors in MBPP are due to syntactic errors (Syntax). Contrasting this, type-aware constraining more than halves compilation errors, reducing them by 59.9%. We observe that models of all sizes and model families benefit similarly from the constraining, with a minimum reduction of 41.7%, even though the total amount of compiler errors varies strongly, e.g., between 5 (Gemma 2 27B) and 42 (CodeLlama 34B) compiler errors in Synthesis.

Improving Functional Correctness We compute the pass@1 performance of the generated code on the test cases provided in the translated datasets of MultiPL-E (Cassano et al., 2023) and present the results in Table 2. We find that our constraining impressively increases success rate by 6.7%, 9.3%, and 66.5% in the synthesis, translation, and repair tasks respectively. The impressive performance on repair tasks on MBPP and HumanEval is likely due to the propensity of LLMs to repeat compiler errors in presented solutions, already observed by prior work (Yang et al., 2024; Mündler et al., 2024), which is resolved by the effective guidance of type-aware constraining.

G Case Study Full Outputs

In this section, we provide the full outputs of the examples shown in the case study in Figs. 13–15. Details about the functionality for each instance are provided in the respective captions.

H Complete Prompts

In this section, we provide the complete prompts to the LLMs for the settings Synthesis, Translation, and Repair. The prompts are templates that were instantiated with the `prompt` field from the MultiPL-E dataset (Cassano et al., 2023). Since the models were instruction tuned, there were various formats for the chat template, we therefore provide the prompts on a high level. These prompts were then formatted using the chat template. The system prompt has been prepended to the first user prompt where no system prompts were available. The user prompt is the comment in front of the problem, with the comment `//` removed. Sample test cases are preserved when present. The model was prompted to complete the last assistant response, i.e., the beginning of the prompt was fixed for the model. The prompts for synthesis and translation are presented in Figs. 16 and 17

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. 18. Compiler errors contain line numbers for localization, which is why we annotate the output with line numbers.

```
function string_to_md5(text: string): string | undefined {
  if (text.length === 0) {
    return undefined;
  }
    const hash = crypto.createHash("md5")
                      .update(text).digest("hex");
  return hash;
}
```

(a) The code generated by the unconstrained Gemma 2 2B. The model tries to calculate the MD5 checksum of the input string using the library `crypto`. However, it does not import it and uses the symbol `crypto` directly, which does not exist yet.

```
function string_to_md5(text: string): string | undefined {
  if (text.length === 0) {
    return undefined;
  }
    const hash = require('crypto').createHash('md5');
  hash.update(text);
  return hash.digest('hex');
}
```

(b) The code generated by Gemma 2 2B with type-aware constraints. The constraint guides the model and help it generate the correct `require` statement that imports the library before use.

Figure 13: Comparison between the function generated by unconstrained Gemma 2 2B (Gemma Team, 2024) and the code generated with the guidance of type-aware constraints on the problem `string_to_md5` (HumanEval #162, TypeScript Translation, 2023). This problem requires a function that calculates the MD5 value of a given string.

```
function fibfib(n: number): number {
    if (n === 0) return 0;
    if (n === 1) return 0;
    if (n === 2) return 1;

    let fibfibNMinus3 = 0;
    let fibfibNMinus2 = 0;
    let fibfibNMinus1 = 1;
    let  fibfibN ;

    for (let i = 3; i <= n; i++) {
      fibfibN = fibfibNMinus1 + fibfibNMinus2 + fibfibNMinus3;
      fibfibNMinus3 = fibfibNMinus2;
      fibfibNMinus2 = fibfibNMinus1;
      fibfibNMinus1 = fibfibN;
    }

    return fibfibN;
}
```

(a) The code generated by the unconstrained Deepseek Coder 33B Instruct. The model uses an uninitialized `fibfibN` to save the n-th of the fibfib sequence along with the three variables `fibfibNMinus1, fibfibNMinus2, fibfibNMinus3` to store $a_{n-1}, a_{n-2}, a_{n-3}$ in the sequence. The code is functionally correct. However, since the program does not initialize `fibfibN`, the compiler infers it to be of type `number | undefined`, which is inconsistent with the return type `number`.

```
function fibfib(n: number): number {
    if (n === 0) return 0;
    if (n === 1) return 0;
    if (n === 2) return 1;

    let fibfibNMinus3 = 0;
    let fibfibNMinus2 = 0;
    let fibfibNMinus1 = 1;
    let fibfibN  = fibfibMinus1 + fibfibMinus2 + fibfibMinus3 ;

    for (let i = 3; i <= n; i++) {
      fibfibN = fibfibNMinus1 + fibfibNMinus2 + fibfibNMinus3;
      fibfibNMinus3 = fibfibNMinus2;
      fibfibNMinus2 = fibfibNMinus1;
      fibfibNMinus1 = fibfibN;
    }

    return fibfibN;
}
```

(b) The code generated by Deepseek Coder 33B Instruct with type-aware constraints. The constraint rules out the possibility of any variable that lacks both type annotation and initilization. Thus, the model generates a correct initialization that helps the compiler to deduce that `fibfibN` is indeed of the type `number`, which aligns with the return type of the function.

Figure 14: 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-aware constraints on the problem `fibfib` (HumanEval #63, TypeScript Translation, 2023). The problem asks to calculate a variant of the Fibonacci sequence called fibfib such that it satisfies $a_n = a_{n-1} + a_{n-2} + a_{n-3}$ rather than the $a_n = a_{n-1} + a_{n-2}$ for Fibonacci sequence.

```
function unique_digits(x: number[]): number[] {
  return x.filter(num => {
    const numStr = String(num);
    return !numStr.some(digit => parseInt(digit) % 2 === 0);
  }).sort((a, b) => a - b);
  }
```

(a) The code generated by the unconstrained Qwen2.5 32B Instruct. The code first filters out the digits containing even digits and then sorts the remaining array. The idea of the code is correct. However, the model hallucinates that the string `numStr` can already be viewed as a list and uses the method `some` directly, which the TypeScript compiler does not recognize. Thus, the code fails compilation with an undefined method.

```
function unique_digits(x: number[]): number[] {
  return x.filter(num => {
    const numStr = String(num);
    return !numStr.split('').some(digit => parseInt(digit) % 2 === 0);
  }).sort((a, b) => a - b);
}
```

(b) The code generated by Qwen2.5 32B Instruct with type-aware constraints. With the constraint, the model recognizes that `some` does not exist for string and uses `split` to convert the string into a list of characters first, solving the problem.

Figure 15: Comparison between the function generated by the unconstrained Qwen2.5 32B Instruct (Hui et al., 2024) and the code generated with the guidance of type-aware constraints on the problem `unique_digits` (HumanEval #104, TypeScript Translation, 2023). The problem asks to calculate a sorted array from a given array that only contains the numbers in the original array without any even digits.

```
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:
  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 16: The full prompt for the synthesis setting. Text in green is based on the problem instance, in this case HumanEval #0.

```
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.

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 17: 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.

````
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 18: 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.
````