ChopChop: Semantically Constraining the Code Output of Language Models

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Language models (LMs) can generate code, but cannot guarantee its correctness—producing outputs that often violate type safety, program invariants, or semantic equivalence. Constrained decoding offers a solution by restricting generation to programs that satisfy desired properties. Yet, existing methods are limited to shallow syntactic constraints or rely on brittle, ad hoc encodings of semantics over token sequences.

We present ChopChop, the first programmable framework for semantic constrained decoding, enabling LMs to generate code that provably satisfies rich semantic properties. ChopChop enables construction of constrained decoders that incorporate advanced formal methods by connecting token-level generation with reasoning over abstract program structures. It is the first system capable of constraining an LM to only generate programs that are provably equivalent to a supplied reference program. We also show that it can naturally implement existing applications, such as type-constrained decoding for TypeScript.

1 Introduction

Language models (LMs) have fundamentally transformed how we interact with code—generating functions, completing boilerplate, and even suggesting entire programs. Yet, despite their success, LMs offer no guarantees of correctness: they produce code that looks plausible but often violates critical syntactic or semantic properties.

Constrained decoding has emerged as a promising solution to this problem [31, 32, 10, 1, 23]. In constrained decoding, a language model generates a sequence one token at a time, but the next token is chosen not only for its likelihood, but also based on whether extending the current output with that token could ultimately produce a program that satisfies a user-defined constraint.

However, existing constrained decoding techniques are limited in scope. Early methods focused solely on syntactic correctness—e.g., enforcing that outputs conform to a context-free grammar (CFG). More recent techniques attempt to enforce richer constraints like type safety [23] or runtime properties [1], but do so via ad hoc treatments in which constraints are expressed on the level of raw text. These approaches are inherently brittle because they do not operate over the formal structure of programs—as abstract syntax trees. Moreover, they preclude integration with more advanced formal methods for reasoning about deep semantic properties—such as program equivalence or adherence to complex invariants—as such methods are fundamentally defined over abstract syntax.

In this paper, we ask:

Can we design a principled, programmable framework for constrained decoding that enforces deep semantic properties—over programs instead of token sequences?

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

Achieving this goal introduces two major challenges.

1. **Bridging the syntax-semantics gap.** Formal methods reason about semantic properties over abstract syntax trees (ASTs), while language models generate concrete syntax one token at a time. Enforcing semantic constraints during decoding thus requires translating between the evolving token prefix and the corresponding space of possible ASTs the prefix might generate.
2. **Dealing with partial programs.** Traditional program analyses operate on complete programs. But in constrained decoding, we must decide—incrementally, as each token is produced—whether a partial prefix could still yield a program that satisfies the desired constraint.

Our approach. We present CHOPCHOP, the first unified, programmable framework for semantic constrained decoding over the abstract syntax of programs. The key idea is to reduce constrained decoding to a problem in the program synthesis literature: *realizability*, the task of determining whether a (possibly infinite) space of programs contains one that satisfies a specified property.

As the LM emits tokens, CHOPCHOP constructs a symbolic representation (as a regular tree grammar) of the set of all ASTs that are syntactically valid completions of the current prefix. CHOPCHOP then analyzes this representation using a user-provided analysis—which is defined at the level of abstract syntax—and checks realizability with respect to the propety the analysis enforces. If there does not exist a valid program in this space, the proposed token is rejected and an alternative is tried. This pipeline ensures that every accepted token keeps the generation process on track toward satisfying the semantic constraints.

Applications. We demonstrate the generality of CHOPCHOP through two diverse applications:

- **Program Equivalence-Guided Decoding:** We constrain a language model to generate programs equivalent, modulo term rewriting, to a reference program. The analysis works by using a data structure known as an e-graph to efficiently reason about equivalence classes of ASTs.
- **Type-Safe Decoding:** We constrain a language model to emit only well-typed programs in a subset of TypeScript. The analysis is natural to define, as like a normal typechecker it operates over the level of abstract syntax.

2 Overview of ChopChop

We illustrate our approach with the following example task:

Generate a sum of odd integers whose total is even.

For instance, the expression `5 + 7` is a valid solution: all summands are odd and the total is even. If we prompt a *language model* (LM) with this task, there is no guarantee it will succeed. It might produce a sum with even numbers (`2 + 2`), an odd total (`1 + 1 + 1`), or nonsensical output (`banana`).

CHOPCHOP enables users to enforce such constraints (i.e., that the summands are odd and the sum is even) on the LM by providing two inputs:

1. A *parser definition* for translating strings generated by the language model to ASTs.
2. A set of *semantic pruners*, each representing a constraint over sets of possible ASTs, used to prune invalid programs. (see `odds` and `even_sum` in Figures 3a and 3b). A pruner is a function that takes a representation of a set of possible abstract syntax trees (ASTs) and returns the subset where ASTs that do not satisfy the constraint have been removed.

Given these inputs, CHOPCHOP interacts with the LM to guarantee that any generated program is syntactically valid and satisfies the provided semantic constraints.

```
int := [1-9][0-9]*          E ::= int        {E1.ast = Lit int.value}
                              | E + int      {E1.ast = Sum E2.ast int.ast}
```

Figure 1: A parser definition for the language of integer sums. The AST of an integer literal "`int`" is its value; that of "`int + E`" is a sum node `Sum int.ast` E_2`.ast`. Left-recursive grammars are handled by ChopChop.

2.1 Semantic Constrained Decoding as Realizability

A trivial way to ensure constraint satisfaction is to let the LM generate a full program, check whether it satisfies the constraints, and retry if it does not; this approach is called "rejection sampling" and is, in general, very inefficient (Section 3 presents settings in which some LMs never produce valid programs using this approach!) Ideally, instead, we would like to rule out "doomed" program prefixes as soon as the LM generates them: for example, if the LM generates the prefix `2 +`, there is no point continuing, since any completion will include the even number 2, violating our constraint.

Therefore, instead of producing full programs and verifying them afterwards, ChopChop follows the approach called *constrained decoding* [31, 32, 10, 1, 23], which restricts the LM's choices of next tokens *during generation*. For example, say the already generated prefix is `2`, and the LM's top two choices for the next token are `+` and `2`. ChopChop would disallow `+`, since it leads to the "doomed" prefix `2 +`, and instead `2` will be chosen, since it can still lead to a valid completion (e.g., `221 + 9`). We refer to this process as *semantic constrained decoding* (SemCD), because it prunes the LM's choices based on semantic constraints over ASTs as opposed to *syntactic constrained decoding* (SynCD), which enforces shallow syntactic properties of the token stream.

To constrain what tokens to allow, ChopChop incrementally constructs a *program space*—a symbolic representation of all possible ASTs that can be generated from the current token prefix using the user-provided parser; it then invokes the user-defined semantic pruners (in our example, `odds` and `even_sum`) to prune this space and remove semantically incorrect programs, and checks whether at least one valid completion remains in the resulting program space.

By drawing a connection to concepts used in program synthesis [12], this process can be formalized as an (approximate) *realizability checker*, $\mathsf{realizable}(\omega, \varphi)$, which determines whether the current token prefix ω can still be extended to a program whose AST satisfies the constraint. If the answer is negative, the LM proposes an alternative token and the process repeats until a realizable prefix is found. This *symbolic, programmable, semantics-aware pruning* enables LMs to generate only programs that satisfy rich semantic properties—without modifying the model, manually rewriting AST constraints to operate on the string-representation of programs, or relying on token-level heuristics.

2.2 Analyzing Prefixes of Programs

Although $\mathsf{realizable}(\omega, \varphi)$ takes a concrete token prefix as input, it fundamentally asks a semantic question: *Does there exist a program, consistent with the prefix ω, that satisfies the constraint φ?* Answering this question requires reasoning not about a single program, but about the (potentially infinite) space of ASTs that can be built by completing ω.

We tackle this problem by breaking it into four conceptual and algorithmic subgoals:

1. **Representation:** How can we finitely describe an infinite program space?
2. **Completion:** Given a prefix ω, how can we algorithmically construct the program space of all ASTs consistent with ω?
3. **Pruning:** Given a program space X, how can we compute or approximate the subset program space $\mathsf{X}' \subseteq \mathsf{X}$ of ASTs that satisfy a semantic constraint φ?
4. **Non-Emptiness:** Given a pruned space X', how can we check whether X' is non-empty?

Goals 3 and 4 may seem redundant: why not check directly whether any AST in X satisfies φ? Unfortunately, this problem is undecidable in general, even when checking φ is decidable for individual ASTs [16, 17]. Our decomposition reflects a trade-off: rather than requiring satisfiability of φ

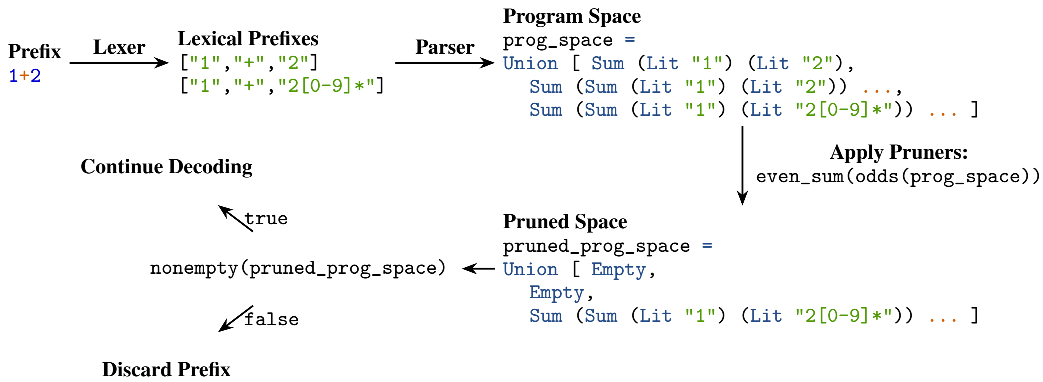


Figure 2: Flow of CHOPCHOP on prefix `1+2`. The prefix is lexed into possible lexical sequences, parsed into a symbolic program space, semantically pruned, and checked for nonemptiness to determine realizability. If realizble, the prefix may be extended. If unrealizable, the prefix is discarded.

to be decidable over program spaces, we instead rely on user-supplied approximate pruners (Goal 3) and implement a fixed, automated check for non-emptiness (Goal 4). This gives users control over the level of approximation, allowing them to express rich constraints while maintaining tractable reasoning over infinite program spaces.

In the rest of the section, we describe our approach following the overview given in Figure 2.

Goal 1: Representing Infinite Program Spaces The following datatype describes the abstract syntax used in our running example:

```
data Expr = Lit String       -- a numeric literal, e.g., Lit "5"
          | Sum Expr Expr    -- sum of two expressions
```

An element of the above datatype represents the abstract syntax tree for a single program. To represent *program spaces* (i.e., potentially infinite sets of programs), we lift the definition of `Expr`:

```
data ExprSpace = Empty                        -- empty program space
               | Union [ExprSpace]            -- union of multiple spaces
               | Lit Regex                    -- set of literals described by a regex
               | Sum ExprSpace ExprSpace      -- Sum operator applied to two subspaces
```

For example, the space `Sum left right` is formed via a cartesian product of all the ASTs in the spaces `left` and `right`, i.e. the set $\{\texttt{Sum}\ e_1\ e_2 \mid e_1 \in \texttt{left}, e_2 \in \texttt{right}\}$. Readers familiar with *version space algebra* [19, 28] will recognize this as a version space, where `Union` is the union node and `Sum` is a join node.

One subtlety is that program spaces can contain cycles. For example, the *infinite* space consisting of *all* sums of integers can be represented using the following recursive definition:

```
all = Union [Lit "[1-9][0-9]*", Sum all all]
```

Note that even though `all` is infinitely recursive, it has a *finite* representation in memory as a cyclic term—i.e., there is no reason to infinitely unroll the recursion. In the current implementation of CHOPCHOP, all program spaces are regular and represented using this finite, cyclic form. To manipulate such cyclic terms, we implement a solver inspired by CoCaml [14], which supports equational reasoning for terms with cycles. This allows us to perform computations over program spaces—e.g., applying transformations such as `odds` and `even_sum` (Figure 3)—without materializing infinite sets.

Goal 2: Computing the Program Space Consistent with a Prefix Given a concrete string prefix ω, our goal is to compute the program space that contains all ASTs that can be parsed from any completion of ω using the user-defined parser (Figure 1).

```haskell
odds :: ExprSpace -> ExprSpace
odds Empty              = Empty
odds (Union children) = Union (map odds children)
odds (Lit regex)       = Lit (regex `intersect` "[0-9]*[13579]") -- only odd
odds (Sum left right) = Sum (odds left) (odds right)
```

(a) odds pruner: retains only programs using odd literals.

```haskell
even_sum :: ExprSpace -> ExprSpace
even_sum Empty              = Empty
even_sum (Union children) = Union (map even_sum children)
even_sum (Lit regex)       = Lit (intersect regex "[0-9]*[02468]") -- only even
even_sum (Sum left right) = Union (Sum (even_sum left) (even_sum right))
                                   (Sum (odd_sum left) (odd_sum right))
```

```haskell
odd_sum :: ExprSpace -> ExprSpace
... -- Analogous to even_sum
```

(b) `even_sum` pruner: retains programs whose total evaluates to an even number.

Figure 3: Example semantic pruners.

We begin by lexing ω into a finite set of *lexical prefixes*. For example, take ω = 1+2. This has two valid lexical prefixes:

- `["1","+","2"]`: where the next character is non-numeric (e.g., the full string might be 1+2+3);
- `["1","+","2[0-9]*"]`: where the next character continues the numeric literal (e.g., 1+21).

Given a lexical prefix, we use the user-supplied parser to compute the corresponding program space. To this end, we use a *derivative-based parser*, following the approach of [21]. In our framework, a parser can be modeled as a cyclic object that encodes the future parses of a stream of lexemes[1]. Critically, parsers support a *derivative* operation, `derivative parser` ω that advances the parser state by consuming a sequence of lexemes ω, analogous to Brzozowski derivatives for regular expressions [6].

For each lexical prefix—e.g., `["1","+","2"]`—we start from the user-provided parser and apply successive derivatives for each lexeme in the prefix; this results in a parser that accepts exactly those programs that begin with the given lexical prefix. Finally, we convert each derived parser into a corresponding program space (essentially by discarding the information about concrete syntax), and combine the program spaces from different lexical prefixes using the `Union` constructor. For our example, the prefixes `["1","+","2"]` and `["1","+","2[0-9]*"]` together would induce the following program space `prog_space`:

```haskell
Union [ Sum (Lit "1") (Lit "2"),                 -- ["1","+","2"] followed by END
        Sum (Sum (Lit "1") (Lit "2"))        e,  -- ["1","+","2"] followed by + E
        Sum (Sum (Lit "1") (Lit "2[0-9]*")) e ] -- ["1","+","2[0-9]*"]
```

where `e = Union [Lit "[1-9][0-9]*", Sum (Lit "[1-9][0-9]*") e]` represents the space of programs derivable from the nonterminal E in Figure 1.

Goal 3: Pruning the Program Space to Satisfy Semantic Constraints To prune away semantically incorrect programs, a user supplies semantic pruners. A *semantic pruner* is a co-recursive function that takes a program space X and returns the sub-space $\mathsf{X}' \subseteq \mathsf{X}$ of ASTs that satisfy a semantic constraintPruners can be composed to enforce conjunctions of constraints, allowing users to modularly define and reuse semantic constraints across tasks.

For our running example, we supply two pruners, `odds` and `even_sum`, shown in Figure 3. To obtain the pruned space `pruned_prog_space`, we apply the two pruners in sequence:

```haskell
pruned_prog_space = even_sum(odds(prog_space))
```

To get a sense of how a pruner is applied to program space, consider the inner application:

[1]We use the term *lexeme* for programming-language tokens to avoid confusion with LM tokens.

```
odds (Union [Sum (Lit "1") (Lit "2"), ...]) => -- distribute over union
Union [odds (Sum (Lit "1") (Lit "2")), ...] => -- distribute over sum
Union [Sum (Lit (intersect "1" "[0-9]*[13579]"))
           (Lit (intersect "2" "[0-9]*[13579]")), ...] =>
Union [Sum (Lit "1") Empty, ...] =>
Union [Empty, ...]
```

In other words, `odds` will prune away the first two sub-spaces of the union in `prog_space`, since they contain even numbers. Note, however, that because the other two sub-spaces of the union (omitted under the ellipsis) are *cyclic terms*, applying the pruner to them does not necessarily reduce to a normal form; hence the task of checking emptiness for pruned sub-spaces is non-trivial.

Goal 4: Deciding Nonemptiness of the Pruned Space The final step in computing $\mathsf{realizable}(\omega, \varphi)$ is to check whether the pruned program space X' is non-empty. To this end, ChopChop implements the function `nonempty :: ExprSpace -> Bool`, which performs a fixpoint computation over a cyclic object representing X'. In our example, `nonempty` determines that the third sub-space of the union in `pruned_prog_space` is non-empty, and hence the whole union is non-empty.

Together, the four components—lexing, parser derivatives, user-defined pruners, and the nonemptiness check—enable us to compute $\mathsf{realizable}(\omega, \varphi)$ for our running example as:

```
nonempty (even_sum (odds (derivative parser ω)))
```

ChopChop unifies syntactic parsing and semantic constraint enforcement within a single symbolic framework. This pipeline is efficient, modular, and requires minimal effort from the user: they define a parser and supply composable pruners for each constraint. ChopChop handles the rest—automatically enforcing semantic constraints during generation without modifying the underlying LM.

3 Evaluation

We demonstrate the generality of our framework by instantiating it for two domains, described in more detail below: enforcing semantic equivalence for a basic functional language and enforcing type safety for TypeScript. Enforcing semantic equivalence via constrained decoding is a completely novel application made possible only by our technique. It is an example of a constraint that is fundamentally beyond the capability of existing approaches. To evaluate, we compare our semantic constrained decoders written in ChopChop to the following baselines:

- ***Unconstrained Decoding***: The LM generates code without any constraints.
- ***Grammar-Constrained Decoding (GCD)***: The LM must produce syntactically valid programs (enforced via a grammar), but no semantic restrictions are applied.

Procedure

We evaluate using a variety of models at different temperatures: a detailed description of our experimental setup is in Appendix 6.1. For each benchmark, model, decoder, and temperature, we run constrained decoding until either: (*i*) the END token is generated, or (*ii*) a fixed token budget (set to 400) is exhausted. or (*iii*) a 150 second timeout is reached. We only implement a naive sampling strategy where if a token is rejected, we backtrack by one token and re-sample with that token removed. Constrained decoders may fail if they use up their token or time budget without completing a valid program—especially if the LM repeatedly proposes unrealizable tokens that must be pruned. Unconstrained decoders may also fail to terminate if the model does not emit an END token withing the budget.

A run is considered **successful** if: (*i*) A complete program is emitted, and (*ii*) It satisfies the semantic constraints of the task (i.e., equivalence or type safety).

3.1 Equivalence-Guided Decoding

In this case study, the LM is given expressions in a basic functional language and is asked to refactor them into equivalent programs. The language consists of basic arithmetic operators, identifiers, integer constants, function applications, and let bindings. For example, consider the following program that computes the distance between two points.

```
sqrt (pow (x2 - x1) 2 + pow (y2 - y1) 2)
```

A valid output for the LM might be:

```
let dx = x2 - x1 in
let dy = y2 - y1 in
sqrt (pow dy 2 + pow dx 2)
```

To represent the program space of equivalent programs, CHOPCHOP uses an *e-graph* [33], a data structure for compactly representing spaces of equivalent programs. Equivalences are defined in terms of a list of *rewrite rules*. For example, the rule $x + y \rightarrow y + x$ encodes the commutativity of addition. Then, given an initial program, these rules are iteratively applied (up to a fixed budget) to matching subterms to generate a space of programs that can be proven equivalent to the original using the given rewrite rules. In the above example for instance, `pow (y2 - y1) 2) + sqrt (pow (x2 - x1) 2` would be an equivalent program stored in the e-graph. To build the e-graphs, CHOPCHOP uses the egglog library [34]. For our case study, we define a set of basic rewrite rules for arithmetic expressions (ones with addition, subtraction, multiplication, and division).

The set of terms an e-graph represents forms a regular tree language and can be represented as a finite tree automata [29]. This is useful because regular tree languages are closed under intersection: our decoder works by intersecting the prefix space at each step with the tree automata corresponding to the e-graph, then checking that the intersection is nonempty.

We created 10 benchmark tasks in the basic functional language, where the goal is to refactor a program into an equivalent one—e.g., factoring out subexpressions into `let` bindings. We use a fixed system prompt that specifies the grammar of the language and instructs the model to return only a refactored program with no explanation. We count a response as *correct* if the LM produces a complete program that is equivalent to the input program, with no post-processing. Unconstrained decoding often fails to produce just the output code, despite being explicitly instructed to do so. Therefore, we also evaluate a variant where the prompt wraps outputs in triple backticks (```) to encourage the model to delimit its code clearly. This avoids penalizing runs that fail only due to formatting. We refer to the two variants as *No Delimit* and *Delimit*, respectively.

3.2 Type-Safe Decoding

For our second instantiation of CHOPCHOP, we implemented a type-constrained decoder for a subset of typescript. To simplify the implementation, we restrict our attention to a syntactic subset of TypeScript that omits certain features such as strings, arrays, lambda abstractions, and property accesses. We source benchmarks from the TypeScript translations of the MBPP [4] tasks from the MultiPL-E dataset [7] and extracted the 74/809 tasks that can be solved in our language fragment. We provide context to the LM which instructs it to avoid language constructs outside our language fragment.

An example task is given below:

```
// Write a typescript function to find the next perfect square
// greater than a given number.
function next_Perfect_Square(N: number): number
```

We count a response as *correct* if the generated TypeScript program compiles.

3.3 Results

Effectiveness Table 1 reports the number of successful runs for CHOPCHOP and the two baselines—unconstrained decoding and grammar-constrained decoding—on all benchmarks.

Table 1: Successful generations for different decoding strategies across models and temperatures (higher is better). For equivalence-guided decoding we report the number of benchmarks for which an equivalent program was produced. For TypeScript we report the number of benchmarks on which compilable code was produced. Best results per column are **bolded**.

		DeepSeek-Coder-6.7b Temperature						CodeLlama-7B Temperature						CodeLlama-13B Temperature					
		0.01	0.3	0.5	0.7	1.0	Tot.	0.01	0.3	0.5	0.7	1.0	Tot.	0.01	0.3	0.5	0.7	1.0	Tot.
Equivalence No Delimit (10 programs)	Unconstrained	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Grammar	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Semantic	**7**	**7**	**8**	**8**	**3**	**33**	**8**	**9**	**9**	**9**	**8**	**43**	**10**	**8**	**8**	**10**	**6**	**42**
Equivalence Delimit (10 programs)	Unconstrained	0	0	0	0	0	0	3	3	2	3	1	12	4	2	2	3	2	13
	Grammar	0	0	0	0	1	1	3	4	6	4	1	18	4	5	2	2	0	13
	Semantic	**10**	**10**	**8**	**9**	**8**	**45**	**9**	**10**	**10**	**6**	**6**	**41**	**9**	**8**	**7**	**8**	**6**	**38**
TypeScript (74 programs)	Unconstrained	0	0	0	0	0	0	**72**	71	70	60	29	302	**71**	68	65	52	12	268
	Grammar	0	0	0	0	2	2	69	65	57	57	30	278	66	63	59	47	19	254
	Semantic	**52**	**49**	**54**	**46**	**31**	**232**	71	**73**	**71**	**67**	**49**	**331**	69	**69**	**66**	**59**	**42**	**305**

Table 2: Overhead of checking realizability in semantic constrained decoding (milliseconds/produced token).

Overhead in ms/token	DeepSeek-Coder-6.7b Temperature					CodeLlama-7B Temperature					CodeLlama-13B Temperature				
	0.01	0.3	0.5	0.7	1.0	0.01	0.3	0.5	0.7	1.0	0.01	0.3	0.5	0.7	1.0
Equiv No Delimit	225	221	199	233	216	159	77	74	77	161	58	68	67	61	142
Equiv Delimit	82	73	564	198	162	50	55	65	121	156	63	74	79	49	128
TypeScript	364	347	343	290	236	240	207	253	223	356	301	228	236	274	323

Across nearly all configurations, semantic constrained decoding delivers consistent and often dramatic improvements.

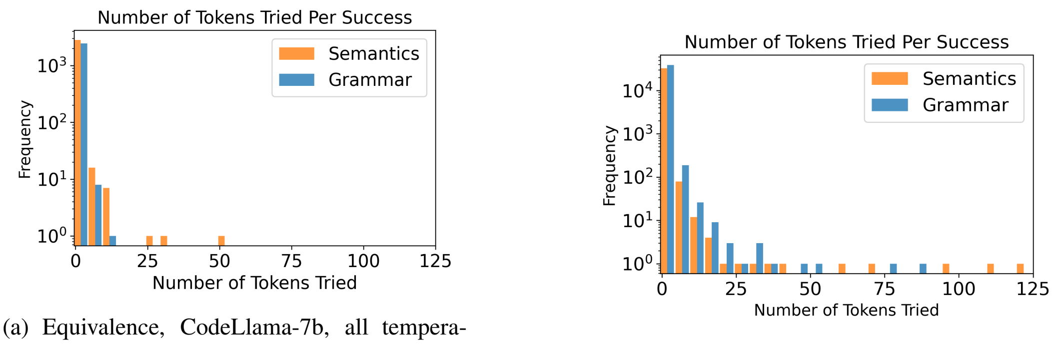
Unconstrained, most models perform very poorly on our equivalence benchmarks, with most failing to generate even a single semantically equivalent program. Several factors contribute to this: (*i*) We intentionally use small- and medium-sized models, which highlight the gains possible even for less capable models. (*ii*) The toy language used in the benchmarks is likely out-of-distribution for most pretrained LMs, which are tuned on real-world languages like Python and JavaScript. In particular, DeepSeek-Coder-6.7B frequently attempts to write Python, Lisp, or TypesScript code, and fails all benchmarks without semantic constraints as a result. (*iii*) Despite clear instructions, models frequently emit natural language explanations, markdown, or commentary—none of which are semantically valid outputs under our equivalence checker.

Overhead Table 2 shows the average overhead of semantic constrained decoding (in ms) per generated token. Overhead on decoding time range from tens to a few hundred milliseconds per token, which is a very small price to pay for assurance provided by semantic constrained decoding. As a reference, on our hardware, CodeLlama-13B takes on average 81 ms to produce a token when unconstrained.

To better illustrate the source of the overhead, Figure 4 plots the distribution of the number of tokens that are tried before finding a realizable one, for CodeLLama-7B. In general, even with semantic constrained decoding the first token tried is accepted most of the times, which is expected as typically LM only have high entropy for specific tokens that are particularly relevant to the final output.

4 Related Work

Constrained decoding. Constrained decoding techniques ensure the output of language models meets a given specification by throwing away invalid next tokens at each step. Grammar-constrained decoding, in which the constraint is given as a context-free grammar, has been well studied [30, 11, 32, 31, 25, 10]. A significant portion of CHOPCHOP's runtime is spent pruning tokens that fail simple syntactic validity. Integrating fast syntactic filtering tools such as LLGuidance [22] as a pre-processing step could greatly reduce this cost by eliminating invalid tokens early. Beyond syntax, more recent work has explored enforcing specific semantic constraints such as type safety [23]. Frameworks such as monitors [1] and completion engines [27] provide abstractions for providing



(a) Equivalence, CodeLlama-7b, all temperatures

(b) Typescript, CodeLlama-7b, all temperatures

Figure 4: Distribution of how many tokens were proven unrealizably by semantic constrained decoding to produce each individual token. The kth bar gives the number of successful tokens that were produced after trying between $5k$ and $5k + 4$ unsuccesful tokens by CodeLlama-7b.

more complex constraints by allowing a user to provide a monitor (written in a general purpose language) that performs the decoding. The main difference between our work and those techniques is that they require the user to write checkers over strings. By contrast, our approach operates at the level of abstract syntax, abstracting away the syntactic component and allowing the user to define pruners at the level of program spaces, thus enabling new applications such as equivalence-guided decoding.

Algebraic approaches to parsing. We build on a long line of work viewing parsers and grammars as algebraic, recursive structures [18, 20]. Might et al. [21] presents a functional approach to parsing based on applying Brzozowski derivatives to parser combinators. Zipper-based variants such as [8, 9] reduce redundant traversals in the basic version of PwD to improve efficiency. Integrating these techniques into our implementation could be another avenue to improve performance.

Regular coinduction. Our implementation relies on regular coinduction to represent and manipulate cyclic program spaces. CoCaml [14, 15] is a framework to unambiguously define and compute recursive functions over regular codata. Because some recursive functions admit more than one interpretation on codata, the CoCaml language allows users to define custom solvers implement their desired semantics. We do not use the CoCaml language directly. However, our Python backend handles the computation of corecursive functions which produce codata (e.g., our pruners) with a solver analogous to the corec solver presented in [14]. Our backend's solver for `@fixpoint`-annotated functions which compute concrete values over regular codata is analogous to the fixpoint solver presented in [14].

Unrealizability and pruning in synthesis. Our approach draws inspiration from the concept of unrealizability–the problem of determining whether no solution exists that satisfies a given specification [12]. Existing approaches to proving unrealizability [12, 13] typically focus on specific synthesis domains, leveraging domain insights to solve particular tasks. Pruning, for example based on types [24] or examples [3], to remove infeasible portions of the search space is a well-established technique in program synthesis. Our work provides a framework to adapt these general methods from traditional synthesis towards constraining the output of LLMs.

5 Conclusion

We introduced ChopChop, a new framework for semantic constrained decoding that allows one to impose semantic constraints directly on the abstract syntax trees representing programs (instead of their string syntax). ChopChop allows one to program constraints by providing *semantic pruners*—recursive program operating over finite representations of the infinitely many programs the LM can produce on a given prefix. This flexibility enables new applications—e.g., constraining the an LM to only output programs that are equivalent (up to rewrite rules) to a given input program.

References

[1] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). Association for Computing Machinery, New York, NY, USA, 1–11. `http://papers.nips.cc/paper_files/paper/2023/hash/662b1774ba8845fc1fa3d1fc0177ceeb-Abstract-Conference.html`

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 1–17.

[4] 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. arXiv:2108.07732 [cs.PL] `https://arxiv.org/abs/2108.07732`

[5] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag, Berlin, Heidelberg, 257–281. doi:`10.1007/978-3-662-44202-9_11`

[6] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. doi:`10.1145/321239.321249`

[7] 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. 2022. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. arXiv:2208.08227 [cs.LG] `https://arxiv.org/abs/2208.08227`

[8] Pierce Darragh and Michael D. Adams. 2020. Parsing with zippers (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 108 (Aug. 2020), 28 pages. doi:`10.1145/3408990`

[9] Romain Edelmann, Jad Hamza, and Viktor Kunčak. 2020. Zippy LL(1) parsing with derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1036–1051. doi:`10.1145/3385412.3385992`

[10] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore. `https://aclanthology.org/2023.emnlp-main.674`

[11] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. In *The 2023 Conference on Empirical Methods in Natural Language Processing*. `https://openreview.net/forum?id=KkHY1WGDII`

[12] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York, NY, USA, July 13-17, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*. Springer, Springer, 335–352. doi:`10.1007/978-3-030-25540-4_18`

[13] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1128–1142. doi:10.1145/3385412.3385979

[14] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. 2017. CoCaml: Functional Programming with Regular Coinductive Types. *Fundam. Informaticae* 150, 3-4 (2017), 347–377. doi:10.3233/FI-2017-1473

[15] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. 2013. Language Constructs for Non-Well-Founded Computation. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–80.

[16] Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2023. Unrealizability Logic. *Proc. ACM Program. Lang.* 7, POPL, Article 23 (Jan. 2023), 30 pages. doi:10.1145/3571216

[17] Jinwoo Kim, Shaan Nagy, Thomas Reps, and Loris D'Antoni. 2025. Semantics of Sets of Programs. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 110 (April 2025), 27 pages. doi:10.1145/3720515

[18] Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A typed, algebraic approach to parsing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 379–393. https://doi.org/10.1145/3314221.3314625

[19] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (2003), 111–156. doi:10.1023/A:1025671410623

[20] Hans Leiß. 1991. Towards Kleene Algebra with Recursion. In *Proceedings of the 5th Workshop on Computer Science Logic (CSL '91)*. Springer-Verlag, Berlin, Heidelberg, 242–256.

[21] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 189–195. doi:10.1145/2034773.2034801

[22] Michał Moskal, Hudson Cooper, Aaron Pham, Devise Lucato, Steph Wolski, and Ying Xiong. 2025. *guidance-ai/llguidance*. https://github.com/guidance-ai/llguidance

[23] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025. Type-Constrained Code Generation with Language Models. *Proc. ACM Program. Lang.* 9, PLDI, Article 171 (June 2025), 26 pages. doi:10.1145/3729274

[24] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *SIGPLAN Not.* 50, 6 (June 2015), 619–630. doi:10.1145/2813885.2738007

[25] Kanghee Park, Timothy Zhou, and Loris D'Antoni. 2025. Flexible and Efficient Grammar-Constrained Decoding. arXiv:2502.05111 [cs.CL] https://arxiv.org/abs/2502.05111

[26] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

[27] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. arXiv:2201.11227 [cs.LG] https://arxiv.org/abs/2201.11227

[28] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. doi:10.1145/2814270.2814310

[29] Dan Suciu, Yisu Remy Wang, and Yihong Zhang. 2025. Semantic foundations of equality saturation. *International Conference on Database Theory* (2025).

[30] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2025. SynCode: LLM Generation with Grammar Augmentation. *Transactions on Machine Learning Research* (2025). https://openreview.net/forum?id=HiUZtgAPoH

[31] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. arXiv:2305.19234 [cs.CL]

[32] Brandon T Willard and Rémi Louf. 2023. Efficient Guided Generation for Large Language Models. *arXiv e-prints* (2023), arXiv–2307.

[33] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. doi:10.1145/3434304

[34] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (June 2023), 25 pages. doi:10.1145/3591239

6 Appendix

6.1 Models, Parameters and Hardware

We run all experiments using the instruction-tuned versions (i.e. models that are trained to follow instructions in the prompt) of the following models: DeepSeek-Coder-6.7b, CodeLlama-7B, and CodeLlama-13B. Each model is evaluated at five different sampling temperatures: $0.01, 0.3, 0.5, 0.7, 1.0$. These ranges of small-to-medium models and low-to-high temperatures lets us explore a range of model capability. We run all experiments on a Supermicro SYS-4029GP-TRT with two Intel(R) Xeon(R) Gold 6230 CPUs, 384 GB RAM, a 4 TB SSD, and eight Nvidia Geforce RTX 2080Ti GPUs.

6.2 Lexing

We open with a brief background on maximal munch lexing, the most widely used lexing formalism.

Maximal Munch Lexing A maximal munch lexer [2] is instantiated by a collection of disjoint regular expressions, each of which corresponds to a different kind of lexeme. For example, we might give the regex `[1-9][0-9]*` to describe the set of strings representing integers, and the regexes `true` and `false` to describe the strings encoding keywords true and false, respectively. We call these regexes *lexeme classes*.

Given such a collection of disjoint regexes, a *lex* of a string ω is a partition $(\omega_1, \dots, \omega_n)$ of ω so that each ω_j matches one of the given regexes. These strings ω_j are called *lexemes*. The *maximal munch lex* of ω is the unique lex so that, for any other lex $(\omega_1, \dots, \omega_j, \omega'_{j+1}, \dots \omega'_m)$, we have $|\omega_j| \geq |\omega'_{j+1}|$.

Lexing a Partial Program Given a string ω that represents a partial program, our goal is to construct a representation of set of maximal munch lexes of all the strings that extend ω. This will be the output we pass to the parser. For example, the string `1 + 3` can be extended to `1 + 34`, whose maximal munch lex is `[1, +, 34]`, or `1 + 3 + 4`, whose maximal munch lex is `[1, +, 3, +, 4]`. To represent the set of maximal munch lexes of completions of ω, we will build a *partial lex* L like the following:

$$L = \{\texttt{[1, +, 3]}, \texttt{[1, +, 3[0-9]+]}\}$$

Each element of L is a *lexical prefix* – a sequence of lexemes that ends in a regex. A lexical prefix describes the set of prefixes that match it. So `[1, +, 3]` describes the lexes `[1, +, 3, +]`, `[1, +, 3, 5, 6]`, etc. And `[1, +, 3[0-9]+]` describes the lexes `[1, +, 31, +]`, `[1, +, 354, 6]`, etc. The partial lex L describes the set of lexes that extend any one of its lexical prefixes. Our goal will be to produce L from ω so that the lexes described by L are exactly the longset match lexes of completions of ω.

To build L incrementally, we introduce a richer representation of L that tells us how much of a regex has already been matched by ω. For example, if I had a "print" lexeme, then

$$L = \{\texttt{[print]}\}$$

could be the partial lex of both $\omega =$ "print" and $\omega =$ "pri". To resolve this, we add a @ annotation that tells us how much of a regex has been matched. Then, I can distinguish

`[print @]`

from

`[pri @ nt]`

Left of the @ is a string that has been explicitly matched by the end of ω, and right of the @ is a regex that has not been matched yet.

To advance an annotated regex by a character a, we define $D_a(\texttt{x @ y}) = \texttt{xa @ } D_a(\texttt{y})$, where $D_a(y)$ is the usual Brzozowski derivative of y with respect to a [6]. For example,

$$D_i(\texttt{pr @ int}) = \texttt{pri @ nt}$$

Algorithm 1 Partial Lexing (partial_lex)

procedure partial_lex($\omega \in \Sigma^*$)
  $\tilde{L} =$ compute_lexer_state(ω)
  $L' =$ remove_annotations($\tilde{L}$)
  $L =$ remove_ignorable_tokens(L')
  return L

@memoize
procedure compute_lexer_state($a_1 \ldots a_n \in \Sigma^*$)
  if $n = 0$ **then**
    return $\{()\}$
  else
    $\tilde{L} \leftarrow$ compute_lexer_state($a_1 \ldots a_{n-1}$)
    $\tilde{L} \leftarrow$ extend_lexer_state($\tilde{L}, a_n$)
    $\tilde{L} \leftarrow$ remove_nonmaximal_munch_lexes($\tilde{L}$)
    return $\tilde{L}$

procedure extend_lexer_state($\tilde{L} : LEXERSTATE, a \in \Sigma$)
  result $\leftarrow \emptyset$
  for each $l = [l_1, \cdots, l_m] \in \tilde{L}$ **do**
    if $m = 0$ **then**
      result $\leftarrow$ result $\cup \{(D_a(@\top_\tau)) \mid D_a(\top_\tau) \neq \bot_\tau\}$
    if $\epsilon \in y_m$ **then**
      result $\leftarrow$ result $\cup \{[l_1, \cdots, l_{m-1}, x_m@\epsilon, D_a(@\top_\tau)] \mid D_a(\top_\tau) \neq \bot_\tau\}$
    if $D_a(l_m) \neq \bot_\tau$ **then**
      result $\leftarrow$ result $\cup \{[l_1, \cdots, l_{m-1}, D_a(l_m)]\}$
  return result

procedure remove_nonmaximal_munch_lexes($\tilde{L} : LEXERSTATE$)
  for each $l = [l_1, \cdots, l_m] \in \tilde{L}$ **do**
    if $\exists l' = [l_1, \ldots, l_{k-1}, l'_k, \ldots, l'_{m'}] \in \tilde{L}. \ (|x_k| < |x'_k| \wedge \epsilon \in y'_k)$ **then**
      $\tilde{L}.pop(l)$
  return $\tilde{L}$

Figure 5: The function compute_lexer_state produces a lexer state $\tilde{L}$ for ω by iteratively extending partial lexes by the next character (extend_lexer_state) and discarding partial lexes which fail maximal munch by not having the largest possible tokens from left to right (remove_nonmaximal_munch_lexes). The outermost function partial_lex turns $\tilde{L}$ into L. Above, Σ represents an alphabet of characters. We memoize compute_lexer_state so that, when a prefix ω grows to $\omega\alpha$, computing compute_lexer_state($\omega\alpha$) will reuse the earlier computation of compute_lexer_state(ω).

Given ω, we will incrementally build a *lexer state*, a set of sequences of such annotated regexes `x @ y` like

$$\tilde{L} = \{(\texttt{pri @ nt})\}$$

that projects to the desired L when annotation symbols @ are removed.

The full algorithm to build L is given in Algorithm 1. It very closely mirrors the naïve approach to maximal munch lexing [2]. We iterate through ω character by character, constructing the lexer state $\tilde{L}$ incrementally. As we go, we discard the partial lexes from $\tilde{L}$ that fail maximal munch by not having the largest possible tokens from left to right. At the end of the algorithm, we remove the pointers and throw away "ignorable" tokens (e.g., whitespace and comments) to convert $\tilde{L}$ into L.

If there is a reserved whitespace character (e.g., ' ') and a class of lexemes that matches a single occurrence of that character (e.g., `\s+`), then the lexer state L that our algorithm produces describes exactly the set of maximal munch lexes of completions of ω, up to the presence/absence of ignorable tokens like whitespace.

6.3 TypeScript Typechecking

We present our typescript grammar in Figure 6. Typing rules for individual TypeScript programs are presented in Figures 7 and 8. We use a terse, inference-only typesystem for individual programs that we reuse when pruning sets of programs. When we write, e.g., $\Gamma \vdash e \implies \texttt{bool}$, we mean that e infers to a type that matches `bool`. Similarly, when we say $\Gamma \vdash s \implies \tau - ()$, we mean that s infers to a subtype of τ that does not contain the unit type. Note that our typing contexts Γ assign types and a designation of mutable/immutable to variables. A much cleaner type system is given in Bierman et al. [5], but this one suffices for our purposes.

A bidirectional typesystem is a typesystem that is written to allow for types to be computed deterministically and syntactically [26]. A bidirectional typesystem contains two kinds of judgments. The first kind of judgment, called checking, is written $\Gamma \vdash x \impliedby \tau$. When a check judgment appears in a hypothesis, it means that we assert that x must type to τ under Γ. The second kind of judgment is inference, written $\Gamma \vdash x \implies \tau$. In hypotheses, such judgments mean that we compute the type of x as τ. This allows us to use τ elsewhere in our hypotheses.

Typepruning of sets of programs is a bidirectional process. Let X be a `ProgramSpace` coterm. Our typepruning system includes two kinds of judgments: pruning judgments and inference judgements. Pruning judgments, written $\Gamma \vdash \mathsf{X} \to_\tau \mathsf{X}'$, mean that $\mathsf{X}' \subseteq \mathsf{X}$ contains all τ-typed terms in X under Γ. This is our analogue of "checking". The second kind of judgments we allow are standard type inference judgments, written $\Gamma \vdash \texttt{t} \impliedby \tau$. Our rules for inference judgments follow Figures 7 and 8. The bulk of the rules for type pruning are given in Figures 9 and 10; we omit a few redundant language features for brevity (loops, which are handled similar to conditionals, etc.).

Statement Grammar

$$
\begin{aligned}
\text{Statements} &\rightarrow \text{Statement} \\
&\quad \mid\ \text{Statement ; Statements} \\[1ex]
\text{Statement} &\rightarrow \text{Assignment ;} \\
&\quad \mid\ \text{Exp ;} \\
&\quad \mid\ \text{RETURN Exp ;} \\
&\quad \mid\ \text{Block} \\
&\quad \mid\ \text{FUNCTION VAR (Typed\_id}^{*}\text{)} \\
&\quad\quad :\ \text{Type Block} \\
&\quad \mid\ \text{FOR(Assignment; Exp;} \\
&\quad\quad \text{Reassignment) Block} \\
&\quad \mid\ \text{WHILE(Exp) Block} \\
&\quad \mid\ \text{IF(Exp) THEN Statement} \\
&\quad\quad \text{ELSE Statement} \\
&\quad \mid\ \text{IF(Exp) THEN Statement} \\[1ex]
\text{Assignment} &\rightarrow \text{LET Typed\_id} = \text{Exp} \\
&\quad \mid\ \text{CONST Typed\_id} = \text{Exp} \\
&\quad \mid\ \text{Reassignment.} \\[1ex]
\text{Reassignment} &\rightarrow \text{Typed\_id} = \text{Exp} \\
&\quad \mid\ \text{Typed\_id}\ ++ \\
&\quad \mid\ \text{Typed\_id} += 1 \\[1ex]
\text{Typed\_id} &\rightarrow \text{VAR}:\ \text{Type} \\[1ex]
\text{Type} &\rightarrow \text{INTTYPE} \\
&\quad \mid\ \text{BOOLTYPE} \\
&\quad \mid\ (\text{Typed\_id}^{*}) \Rightarrow\ \text{Type} \\[1ex]
\text{Block} &\rightarrow \{\} \\
&\quad \mid\ \{\ \text{Statements}\ \}
\end{aligned}
$$

Expression Grammar

$$
\begin{aligned}
\text{Exp} &\rightarrow \text{Form} \\
&\quad \mid\ \text{Form ? Exp : Exp.} \\[1ex]
\text{Form} &\rightarrow \text{Comp} \\
&\quad \mid\ \text{Comp \&\& Comp} \\
&\quad \mid\ \text{Comp} \mid\mid \text{Comp.} \\[1ex]
\text{Comp} &\rightarrow \text{Bin} \\
&\quad \mid\ \text{Bin} < \text{Bin} \\
&\quad \mid\ \text{Bin} == \text{Bin} \\
&\quad \vdots \\[1ex]
\text{Bin} &\rightarrow \text{App} \\
&\quad \mid\ \text{App} + \text{Bin} \\
&\quad \mid\ \text{App} - \text{Bin} \\
&\quad \vdots \\[1ex]
\text{App} &\rightarrow \text{Base\_exp} \\
&\quad \mid\ \text{Base\_exp ()} \\
&\quad \mid\ \text{Base\_exp } (A). \\[1ex]
\text{Base\_exp} &\rightarrow \text{INT} \\
&\quad \mid\ \text{VAR} \\
&\quad \mid\ \text{(Exp).} \\[1ex]
\text{Exps} &\rightarrow \text{Exp} \\
&\quad \mid\ \text{Exp , Exps.}
\end{aligned}
$$

Figure 6: Our Subset of TypeScript. The start nonterminal is Statements, in the upper left.

Inference Typing Rules for Expressions and Statements Expression Rules

$$\textsc{Int}\quad \frac{}{\Gamma \vdash 0 \implies \text{int}}$$

$$\textsc{True}\quad \frac{}{\Gamma \vdash \text{True} \implies \text{bool}}$$

$$\textsc{Var}\quad \frac{(x, \tau, \_) \in \Gamma}{\Gamma \vdash x \implies \tau}$$

$$\textsc{Sum}\quad \frac{\Gamma \vdash e1 \implies \text{int} \quad \Gamma \vdash e2 \implies \text{int}}{\Gamma \vdash e1 + e2 \implies \text{int}}$$

$$\textsc{Ternary Expression}\quad \frac{\Gamma \vdash e_1 \implies \text{bool} \quad \Gamma \vdash e_2 \implies \tau \quad \Gamma \vdash e_3 \implies \tau}{\Gamma \vdash (e_1 ? e_2 : e_3) \implies \tau}$$

$$\textsc{Function Application}\quad \frac{\Gamma \vdash f \implies \tau_1, \ldots, \tau_n \to \tau \quad \forall j < n.\, \Gamma \vdash x_j \implies \tau_j}{\Gamma \vdash f(x_1, \ldots, x_n) \implies \tau}$$

Figure 7: Selected Inference Rules for Typing Individual Expressions.

Inference Typing Rules for Individual Statements
Statement Rules

EXPRESSION STATEMENT

$$\frac{\Gamma \vdash e \implies \tau' \quad \Gamma \vdash \bar{s} \implies \tau}{\Gamma \vdash e; \bar{s} \implies \tau}$$

LET VARIABLE DECLARATION

$$\frac{\Gamma \vdash e \implies \top \quad \Gamma + (x, \tau, \text{mutable}) \vdash \bar{s} \implies \tau'}{\Gamma \vdash \text{let } x : \tau = e; \bar{s} \implies \tau'}$$

CONST VARIABLE DECLARATION

$$\frac{\Gamma \vdash e \implies \tau \quad \Gamma + (x, \tau, \text{immutable}) \vdash \bar{s} : \tau'}{\Gamma \vdash \text{const } x : \tau = e; \bar{s} : \tau'}$$

IF-THEN-ELSE – MAY NOT RETURN

$$\frac{\Gamma \vdash e \implies \text{bool} \quad \Gamma \vdash s_1 \implies \tau \quad \Gamma \vdash s_2 \implies \tau \quad \Gamma \vdash \bar{s} \implies \tau}{\Gamma \vdash \text{if } e \text{ then } s_1 \text{ else } s_2; \bar{s} \implies \tau}$$

IF-THEN-ELSE – DEFINITELY RETURNS

$$\frac{\Gamma \vdash e \implies \text{bool} \quad \Gamma \vdash s_1 \implies \tau - () \quad \Gamma \vdash s_2 \implies \tau - () \quad \Gamma \vdash \bar{s} \implies \top}{\Gamma \vdash \text{if } e \text{ then } s_1 \text{ else } s_2; \bar{s} \implies \tau}$$

WHILE

$$\frac{\Gamma \vdash e \implies \text{bool} \quad \Gamma \vdash s \implies \tau \quad \Gamma \vdash \bar{s} \implies \tau}{\Gamma \vdash \text{while}(e)\ s; \bar{s} \implies \tau}$$

NO-OP

$$\frac{}{\Gamma \vdash \cdot \implies ()}$$

Figure 8: Selected Typing Rules for Individual Statements. We give the typing rules of individual programs in our subset of TypeScript, eliding some trivial cases. Note that $\bar{s}$ refers to a (possibly empty) sequence of statements. We use $\cdot$ to denote the empty sequence of statements.

Set Builders

UNION

$$\frac{\Gamma \vdash \texttt{x1} \rightarrow_\tau \texttt{x1'} \quad \cdots \quad \Gamma \vdash \texttt{xn} \rightarrow_\tau \texttt{xn'}}{\Gamma \vdash \texttt{Union (x1 ... xn)} \rightarrow_\tau \texttt{Union (x1' ... xn')}}$$

EMPTY

$$\frac{}{\Gamma \vdash \texttt{Empty} \rightarrow_\tau \texttt{Empty}}$$

(a) Set Builders

Expressions

CONST

$$\frac{reg \not\subseteq id\_regex}{\Gamma \vdash \texttt{ConstS (reg)} \rightarrow_\tau \texttt{ConstS (intersection reg \textbackslash tau.regex)}}$$

WELL-TYPED COMPLETE VARIABLE

$$\frac{reg \subseteq id\_regex \quad reg.has\_only\_one\_member \quad \Gamma[reg] \leq \tau}{\Gamma \vdash \texttt{ConstS (reg)} \rightarrow_\tau \texttt{ConstS (reg))}}$$

ILL-TYPED COMPLETE VARIABLE

$$\frac{reg \subseteq id\_regex \quad reg.has\_only\_one\_member \quad \Gamma[reg] \not\leq \tau}{\Gamma \vdash \texttt{ConstS (reg)} \rightarrow_\tau \texttt{Empty})}$$

INCOMPLETE VARIABLE

$$\frac{reg \subseteq id\_regex \quad \neg reg.has\_only\_one\_member}{\Gamma \vdash \texttt{ConstS (reg)} \rightarrow_\tau \texttt{ConstS (} \{\texttt{x} \in \Gamma \mid \texttt{x} \in \texttt{reg} \wedge \Gamma[\texttt{x}] \leq \tau\}))}$$

(b) Selected Base Expressions. id_regex is the constant regex for identifiers. τ_regex is the regex describing constants of type τ – this regex may be empty.

SUM

$$\frac{\Gamma \vdash \texttt{a} \rightarrow_\tau \texttt{a'} \quad \Gamma \vdash \texttt{b} \rightarrow_\tau \texttt{b'}}{\Gamma \vdash \texttt{SumS (a b)} \rightarrow_\tau \texttt{SumS (a' b')}}$$

FUNCTION APPLICATION WITH COMPLETE FUNCTION

$$\frac{\texttt{(collapse f) == Some s} \quad \Gamma \vdash \texttt{s} \Longleftarrow \tau_1 \times \cdots \times \tau_n \rightarrow \tau \quad \Gamma \vdash \texttt{xs} \rightarrow_{\tau_1 \times \cdots \times \tau_n} \texttt{xs'}}{\Gamma \vdash \texttt{Apply (f xs)} \rightarrow_\tau \texttt{Apply (f xs')}}$$

FUNCTION APPLICATION WITH COMPLETE FUNCTION

$$\frac{\texttt{(collapse f) == Nothing} \quad \Gamma \vdash \texttt{f} \rightarrow_{\top \rightarrow \tau} \texttt{f'}}{\Gamma \vdash \texttt{Apply (f xs)} \rightarrow_\tau \texttt{Apply (f' xs)}}$$

(c) Compound Expressions

TERNARY

$$\frac{\Gamma \vdash \texttt{guard} \rightarrow_{\texttt{bool}} \texttt{guard'} \quad \Gamma \vdash \texttt{then} \rightarrow_\tau \texttt{then'} \quad \Gamma \vdash \texttt{else} \rightarrow_\tau \texttt{else'}}{\Gamma \vdash \texttt{TernaryOp (guard then else)} \rightarrow_\tau \texttt{TernaryOp (guard' then' else')}}$$

EXPRESSION SEQUENCE

$$\frac{\Gamma \vdash \texttt{x} \rightarrow_{\tau_1} \texttt{x'} \quad \Gamma \vdash \texttt{xs} \rightarrow_{\tau_2 \times \cdots \times \tau_n} \texttt{xs'}}{\Gamma \vdash \texttt{ExpSeq (x xs)} \rightarrow_{\tau_1 \times \cdots \times \tau_n} \texttt{ExpSeq (x' xs')}}$$

(d) Miscellaneous Expressions

Figure 9: Typepruning rules for expressions and set builders.

Statement Sequence

$$\frac{\Gamma \vdash \texttt{s} \rightarrow_{\tau-()} \texttt{s'} \quad \Gamma \vdash \texttt{ss} \rightarrow_{\top} \texttt{ss'} \quad \Gamma \vdash \texttt{s} \rightarrow_{()} \texttt{s''} \quad \Gamma \vdash \texttt{ss} \rightarrow_{\tau} \texttt{ss''}}{\Gamma \vdash \texttt{StatementSeq (s ss)} \rightarrow_{\tau} \texttt{Union (StatementSeq (s' ss') StatementSeq (s'' ss''))}}$$

Expression Statement – Void Type Constraint

$$\frac{\Gamma \vdash \texttt{e} \rightarrow_{\top} \texttt{e'} \quad () \leq \tau}{\Gamma \vdash \texttt{ExpStatement (e)} \rightarrow_{\tau} \texttt{ExpStatement (e')}}$$

Expression Statement – Nonvoid Type Constraint

$$\frac{() \not\leq \tau}{\Gamma \vdash \texttt{ExpStatement (e)} \rightarrow_{\tau} \texttt{Empty}}$$

Return

$$\frac{\Gamma \vdash \texttt{e} \rightarrow_{\tau} \texttt{e'}}{\Gamma \vdash \texttt{Return (e)} \rightarrow_{\tau} \texttt{Return (e')}}$$

While

$$\frac{\Gamma \vdash \texttt{b} \rightarrow_{\text{bool}} \texttt{b'} \quad \Gamma \vdash \texttt{ss} \rightarrow_{\tau} \texttt{ss}'}{\Gamma \vdash \texttt{While (b ss)} \rightarrow_{\tau} \texttt{While (b' ss')}}$$

(a) Statements

If-Then-Else

$$\frac{\Gamma \vdash \texttt{guard} \rightarrow_{\texttt{bool}} \texttt{guard'} \quad \Gamma \vdash \texttt{then} \rightarrow_{\tau} \texttt{then'} \quad \Gamma \vdash \texttt{else} \rightarrow_{\tau} \texttt{else'}}{\Gamma \vdash \texttt{Ite (guard then else)} \rightarrow_{\tau} \texttt{Ite (guard' then' else')}}$$

Function Declaration

$$\frac{\Gamma \vdash \texttt{guard} \rightarrow_{\texttt{bool}} \texttt{guard'} \quad \Gamma \vdash \texttt{then} \rightarrow_{\tau} \texttt{then'} \quad \Gamma \vdash \texttt{else} \rightarrow_{\tau} \texttt{else'}}{\Gamma \vdash \texttt{FunctionDecl (guard then else)} \rightarrow_{\tau} \texttt{TernaryOp (guard' then' else')}}$$

Let Binding Complete LHS

$$\frac{\begin{array}{c}\texttt{(collapse type) == Some t} \quad \texttt{(collapse var) == Some (ConstS v)} \quad v \notin \Gamma \\ \tau = \texttt{(parse\_type t)} \quad \Gamma + (\texttt{(get\_name var)}, \tau, \texttt{immutable}) \vdash \texttt{rhs} \rightarrow_{\tau} \texttt{rhs'}\end{array}}{\Gamma \vdash \texttt{Let (var type rhs)} \rightarrow_{\tau} \texttt{Let (var type rhs')}}$$

Let Binding Incomplete LHS

$$\frac{\texttt{(collapse type) == Nothing} \vee \texttt{(collapse var) == Nothing}}{\Gamma \vdash \texttt{Let (var type rhs)} \rightarrow_{\tau} \texttt{Let (var type rhs)}}$$

(b) If-then-else, Function Declaration, Let Bindings

Figure 10: Type Pruning Rules for Statements

6.3.1 Benchmarks and Additional Data

Context All TypeScript experiments used the following (somewhat dramatic) context in instruct mode:

~~~~
You are a very skilled coding assistant for the TypeScript programming language.
An very important automated service will ask you to write a typescript function.
The query begins with a comment describing the desired function behavior.
Then, the query gives a signature for the function you are supposed to write.
For example, a query might look like:

```
// Write a typescript function to add two numbers.
function sum(left_addend: number, right_addend: number): number
```

Your response should be a correct implementation of the function.
Start and end your solution with a codeblock using ```.
For example:

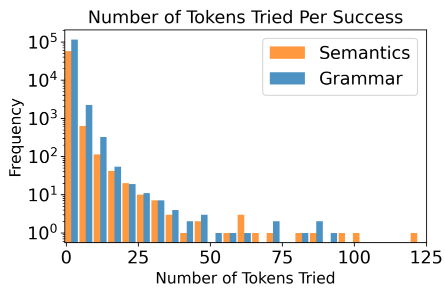
```
function sum(left_addend: number, right_addend: number): number {
    return left_addend + right_addend;
}
```

NEVER write the name of the language in your program.
Do NOT use arrays, strings, lambdas, or comments.
Do NOT write anything before or after your codeblock.
ONLY output code.
You MUST include type annotations.
Your program MUST COMPILE AS WRITTEN OR LIVES WILL BE LOST.
~~~~

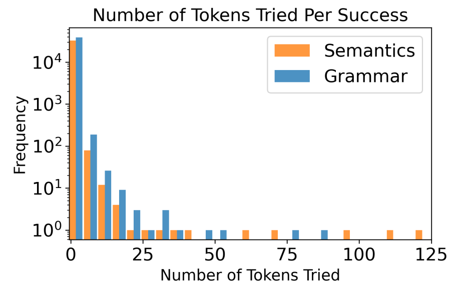
Benchmarks We ran our experiments on the following 74 benchmarks from the MBPP [4] benchmarks available in the MultiPL-E dataset [7]:

```
mbpp_80_tetrahedral_number
mbpp_392_get_max_sum
mbpp_171_perimeter_pentagon
mbpp_127_multiply_int
mbpp_435_last_Digit
mbpp_287_square_Sum
mbpp_606_radian_degree
mbpp_803_is_perfect_square
mbpp_731_lateralsurface_cone
mbpp_581_surface_Area
mbpp_135_hexagonal_num
mbpp_739_find_Index
mbpp_17_square_perimeter
mbpp_77_is_Diff
mbpp_126_sum
mbpp_266_lateralsurface_cube
mbpp_797_sum_in_range
mbpp_3_is_not_prime
mbpp_458_rectangle_area
mbpp_441_surfacearea_cube
mbpp_162_sum_series
mbpp_448_cal_sum
mbpp_738_geometric_sum
mbpp_239_get_total_number_of_sequences
```

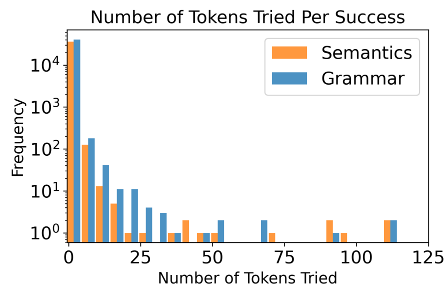
mbpp_59_is_octagonal
mbpp_638_wind_chill
mbpp_577_last_Digit_Factorial
mbpp_84_sequence
mbpp_724_power_base_sum
mbpp_641_is_nonagonal
mbpp_279_is_num_decagonal
mbpp_72_dif_Square
mbpp_781_count_divisors
mbpp_309_maximum
mbpp_295_sum_div
mbpp_14_find_Volume
mbpp_167_next_power_of_2
mbpp_600_is_Even
mbpp_742_area_tetrahedron
mbpp_432_median_trapezium
mbpp_234_volume_cube
mbpp_422_find_Average_Of_Cube
mbpp_292_find
mbpp_389_find_lucas
mbpp_227_min_of_three
mbpp_388_highest_Power_of_2
mbpp_271_even_Power_Sum
mbpp_67_bell_number
mbpp_274_even_binomial_Coeff_Sum
mbpp_86_centered_hexagonal_number
mbpp_574_surfacearea_cylinder
mbpp_430_parabola_directrix
mbpp_406_find_Parity
mbpp_605_prime_num
mbpp_264_dog_age
mbpp_770_odd_num_sum
mbpp_453_sumofFactors
mbpp_244_next_Perfect_Square
mbpp_93_power
mbpp_291_count_no_of_ways
mbpp_637_noprofit_noloss
mbpp_293_otherside_rightangle
mbpp_592_sum_Of_product
mbpp_256_count_Primes_nums
mbpp_479_first_Digit
mbpp_267_square_Sum
mbpp_58_opposite_Signs
mbpp_103_eulerian_num
mbpp_20_is_woodall
mbpp_96_divisor
mbpp_404_minimum
mbpp_752_jacobsthal_num
mbpp_765_is_polite
mbpp_801_test_three_equal



(a) TypeScript, DeepSeek-Coder-6.7b, all temperatures



(b) TypeScript, CodeLlama7B, all temperatures



(c) TypeScript, CodeLlama13B, all temperatures

Figure 11: Distribution of how many tokens were proven unrealizably by semantic constrained decoding to produce each individual token. The kth bar gives the number of successful tokens that were produced after trying between $5k$ and $5k + 4$ unsuccesful tokens by CodeLlama-7b.

Additional Results We include the total number of guesses required per token for each of our three generation modes in Figure 11.

6.4 Equivalence-Guided Decoding

6.4.1 Benchmarks and Additional Data.

Context. The equivalence benchmarks use the following context. The last line is removed for the NO-DELIMIT experiments.

````
You are a code refactoring assistant for a simple functional language.
The language consists of expressions which are either identifiers, integers,
basic arithmetic operations, function application, and let expressions.
The only binary operators are +, -, *, and /.
All other functions (for example, sqrt or pow) are named---
ONLY use names appearing in the original program.

As examples, syntactically valid programs would include:

```
let x = sqrt 42 in
let y = pow (f x) 2 in
y - 3
```

and

```
f x + g y
```
````

```
Your job is to refactor programs into *equivalent* ones which also
have clear, readable style using let bindings when helpful.
Never introduce new features not in the language.
Never include comments or explanations.
ONLY output code, then IMMEDIATELY stop.
Never redefine variables in the original program
or that have already been defined.

Start and end your solution with a codeblock using ```.
```

Benchmarks. We show the 10 benchmark programs we used below.

```
1.  fetch_document (authorize_user_for_document (
    authenticate_user current_user web_request) document_id)

2.  sqrt (pow (x1 - x2) 2 + pow (y1 - y2) 2)

3.  pow 10 (-15) * (66743 * m1 * m2) / (pow r 2)

4.  add_watermark (apply_filter (
   crop_image original_image selection) filter_type) watermark_image

5.  start + (end - start) * scale

6.  (sum (filter positive xs)) / (length (filter positive xs))

7.  power / 1000 * hours * price_per_kwh

8.  (-b + sqrt ((pow b 2) - 4 * a * c)) / (2 * a)

9.  map toUpper (filter isAlpha s)

10. sqrt ((pow (a - ((a+b+c)/3)) 2) +
    (pow (b - ((a+b+c)/3)) 2) + (pow (c - ((a+b+c)/3)) 2)) / 3
```

Egglog file. We show the Egglog file defining the rewrites for the initial e-graph below. It encodes basic arithmetic rules.

```
(datatype Math
  (Num i64)
  (Str String)
  (Var String)
  (Add Math Math)
  (Sub Math Math)
  (Neg Math)
  (Pow Math Math)
  (Sqrt Math)
  (Mul Math Math)
  (Div Math Math)
  (App Math Math))

(rewrite (Add a b)
         (Add b a))

(rewrite (Add (Num a) (Num b))
         (Num (+ a b)))

(rewrite (Add (Add a b) c)
```

```
          (Add a (Add b c)))

(rewrite (Neg a)
          (Sub (Num 0) a))

(rewrite (Sub (Num 0) a)
          (Neg a))

(rewrite (Sub a b)
          (Add a (Mul (Num -1) b)))

(rewrite (Sub (Num a) (Num b))
          (Num (- a b)))

(rewrite (Mul a b)
          (Mul b a))

(rewrite (Mul (Num a) (Num b))
          (Num (* a b)))

(rewrite (Mul (Mul a b) c)
          (Mul a (Mul b c)))

(rewrite (Mul a (Add b c))
          (Add (Mul a b) (Mul a c)))

(rewrite (Div a b)
          (Mul a (Div (Num 1) b)))

(rewrite (Mul a (Div (Num 1) b))
          (Div a b))

(rewrite (Div (Num 1) (Mul b c))
          (Mul (Div (Num 1) b) (Div (Num 1) c)))

(rewrite (Mul (Div (Num 1) b) (Div (Num 1) c))
          (Div (Num 1) (Mul b c)))
```