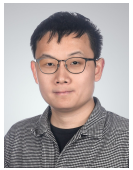





Type-Constrained Code Generation with Language Models

Niels Mündler*, Jingxuan He*, Hao Wang, Koushik Sen, Dawn Song, Martin Vechev

ACM PLDI 2025

SRILAB ETH zürich Berkeley UNIVERSITY OF CALIFORNIA

AI is Redefining Software Development

Completion

GitHub Copilot

JET BRAINS

tabnine

Chat

Gemini

Claude

Agent

SWE-agent

OpenHands

DEVIN

Microsoft has over a million paying Github Copilot users: CEO Nadella

Microsoft

Written by **Tiernan Ray**

Oct. 25, 2023 at 5:52 a.m. PT

AI means everyone can now be a programmer, Nvidia chief says

By **Reuters**

May 29, 2023 3:40 PM PDT

NVIDIA.

Over 25% of Google's code is now written by AI—and CEO Sundar Pichai says it's just the start

Google

BY **GREG MCKENNA**

October 30, 2024 at 11:14 AM EDT

ANTHROP\C

37.2% of queries sent to Claude were in the "computer and mathematical" category, which in large part covers software engineering roles.

Feb 10, 2025

SOTA LLMs generate semantic errors and unsafe code

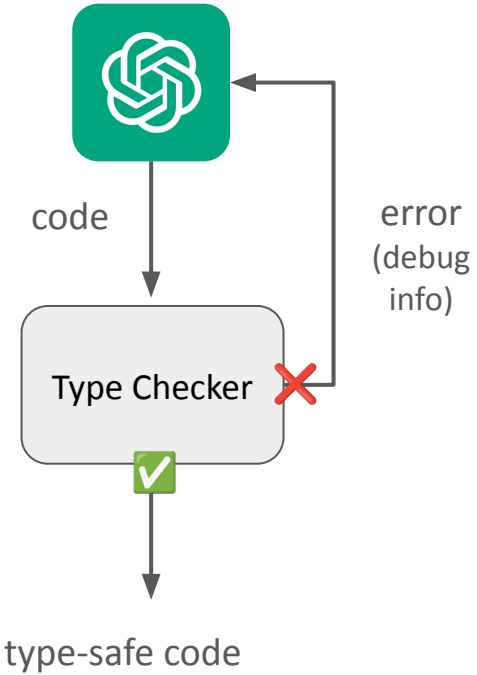
Our Work

Leveraging Type Systems for safe LLM code generation

Direct approach: Feedback loop with type checker

Treat type checker as a black box:

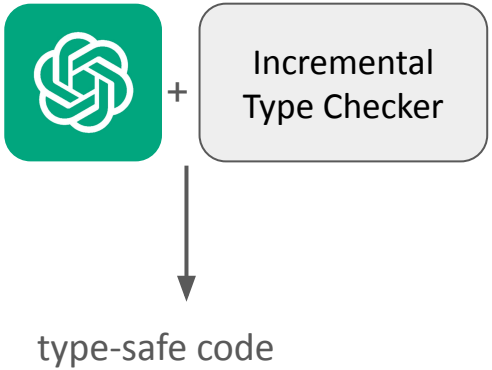
- No control during LLM decoding
- LLM needs to figure out a well-typed alternative from the error messages
- LLM passes: unpredictable, can require many iterations!



Our work: Constraining LLMs' next-token decoding with types

Potential Benefits:

- Full control during decoding
- Generated code is guaranteed to be type safe
- LLM passes: predictable, one pass



Wanted: new **incremental** type checkers for LLM decoding

Background: LLM generates code left-to-right

“Write a function to check if a string represents an integer”

```
function
```

Background: LLM generates code left-to-right

“Write a function to check if a string represents an integer”

```
function is
```

Background: LLM generates code left-to-right

“Write a function to check if a string represents an integer”

```
function is_int
```

Background: LLM generates code left-to-right

“Write a function to check if a string represents an integer”

Background: LLM generates code left-to-right

“Write a function to check if a string represents an integer”

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

Background: LLM generates code left-to-right

"Write a function to check if a string represents an integer"

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

Background: LLM generates code left-to-right

"Write a function to check if a string represents an integer"

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

Background: LLM generates code left-to-right

"Write a function to check if a string represents an integer"

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

num is of type number
parseInt expects type string

Generated by
CodeLlama 32B

Background: LLM generates code left-to-right

"Write a function to check if a string represents an integer"

```
function is_int(text: string): boolean {
  const num = Number(text);
  return !isNaN(num) &&
    parseInt(num, 10)
}
```

num is of type number
parseInt expects type string

Generated by CodeLlama 32B

Background: LLM generates code left-to-right

“Write a function to check if a string represents an integer”

```
function is_int(text: string): boolean {
  const num = Number(text);
  return !isNaN(num) &&
    parseInt(num.toString()) === num
}
```

There is a valid completion!

Incremental type checker

“Can current output be completed to be well-typed?”

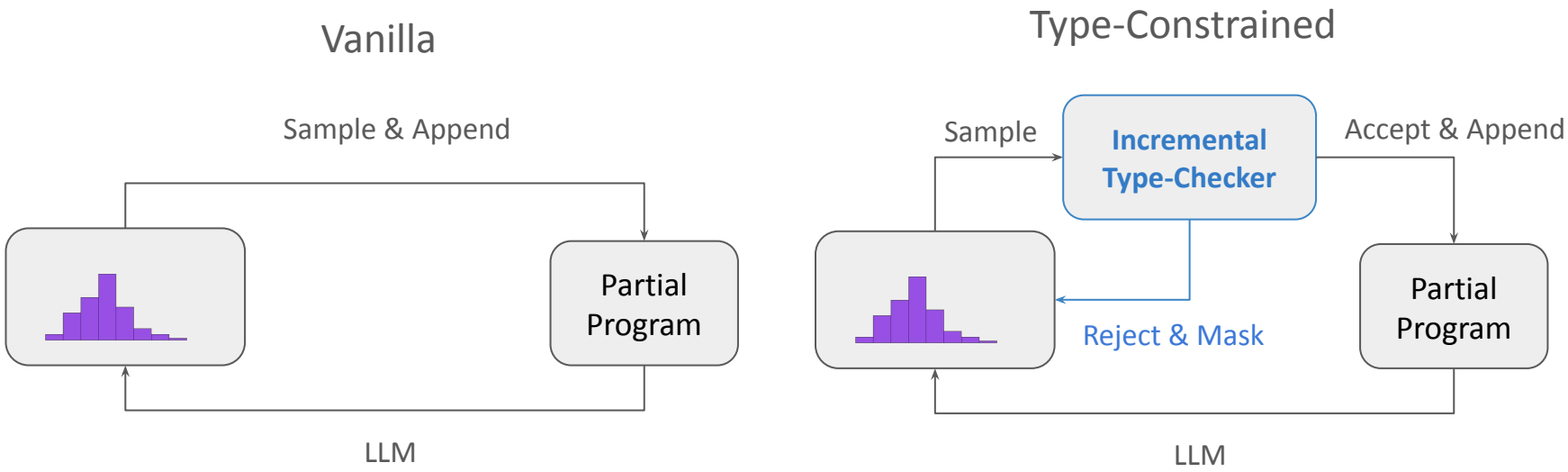
```
function is_int(text: string): boolean {
  const num = Number(text);
  return !isNaN(num) &&
    parseInt(num
```

ok!

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

not ok!

Vanilla vs. Type-Constrained LLM code generation



Building an Incremental Type checker

Key ingredient: Incremental type-checking automaton

Non-deterministic automaton that only accepts well-typed programs

- Abstract syntax tree
- Augmented with typing environment

Ensure *prefix property* on automaton

- Every state leads to an accepting state
- If we are in state ⟶ current parsed output is prefix
- If in no state ⟶ current parsed output is not prefix

Applying our method to a subset of TypeScript

Generic simply typed language

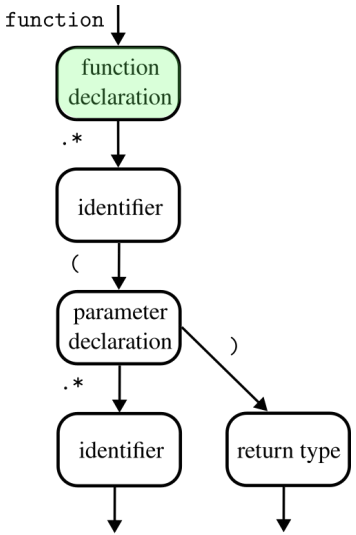
l ::=	Literal	p ::= x : T	Typed Identifier
`\d+`	Numeric Literal		
`"\w*"`	String Literal	T ::=	Type
`true` | `false`	Boolean Literal	`number`	Numeric Type
		`string`	String Type
x ::= `\w+`	Identifier	`boolean`	Boolean Type
		$(\overline{p})$ => T	Function Type
e ::= e_0 | e_1	Expression		
e_0 ::=	Base Expression	s ::=	Statement
l	Literal	`let` x : T `;`	Variable Declaration
x	Identifier	e `;`	Expression Statement
$(\overline{p})$ => e	Function Expression	`return` e `;`	Return Statement
(e)	Grouped Expression	`{` $\overline{s}$ `}`	Statement Block
e_1 ::=	Extension Expression	`function` x $(\overline{p})$: T `{` $\overline{s}$ `}`	Function Definition
$e \odot e$	Binary Operator	`if` (e) s `else` s	If-Then-Else Statement
$e(\overline{e})$	Function Call		
$e.n$	Member Access	M ::= $\overline{s}$	Program

TypeScript extensions

- Mutable vs Immutable Variables
- Arrays
- Loops
- More Operators and Types
- Operator Precedence
- Global Identifiers
- Imports
- Polymorphic Built-Ins
- …

Incremental type checking

Automaton



Typing Environment

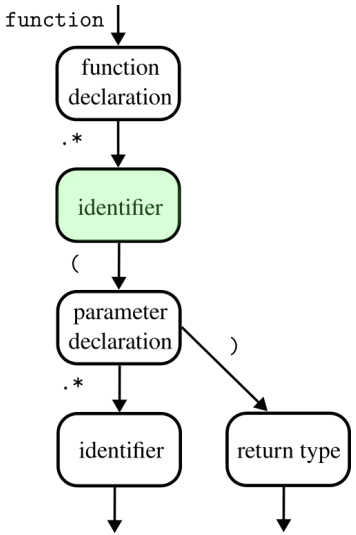
Identifier	Type
parseInt	(string) => number

Code

```
function
```

Incremental type checking

Automaton



Typing Environment

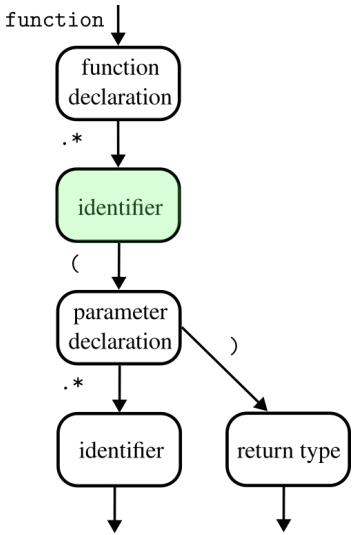
Identifier	Type
parseInt	(string) => number

Code

```
function is
```

Incremental type checking

Automaton



Typing Environment

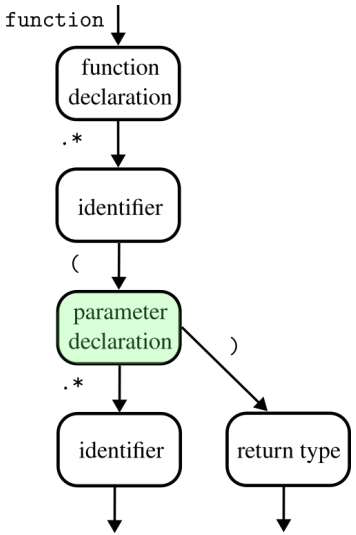
Identifier	Type
parseInt	(string) => number

Code

```
function is_int
```

Incremental type checking

Automaton



Typing Environment

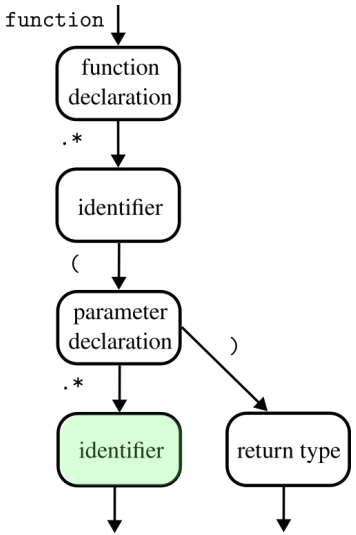
Identifier	Type
parseInt	(string) => number

Code

```
function is_int(text
```

Incremental type checking

Automaton



Typing Environment

Identifier	Type
parseInt	(string) => number

Code

```
function is_int(text
```

Incremental type checking

Automaton

Typing Environment

Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean

Code

```
function is_int(text: string): boolean {
```

Incremental type checking

Automaton

Typing Environment

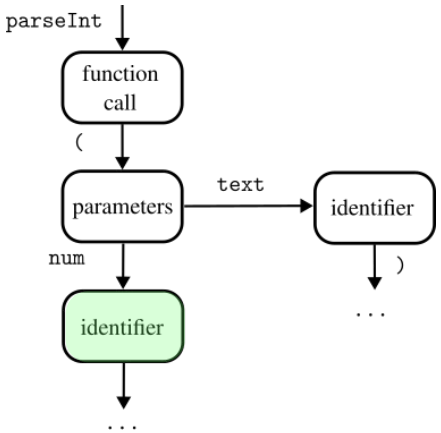
Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

Code

```
function is_int(text: string): boolean {
  const num = Number(text);
```

Incremental type checking

Automaton



Typing Environment

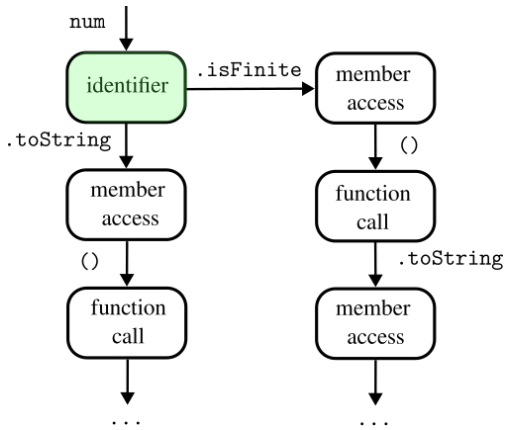
Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

Code

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

Incremental type checking

Automaton



Typing Environment

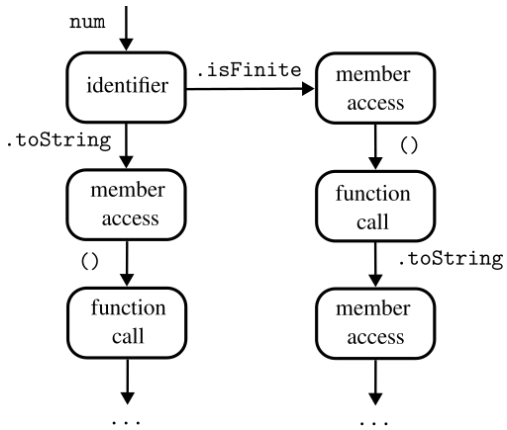
Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

Code

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

Incremental type checking

Automaton



Typing Environment

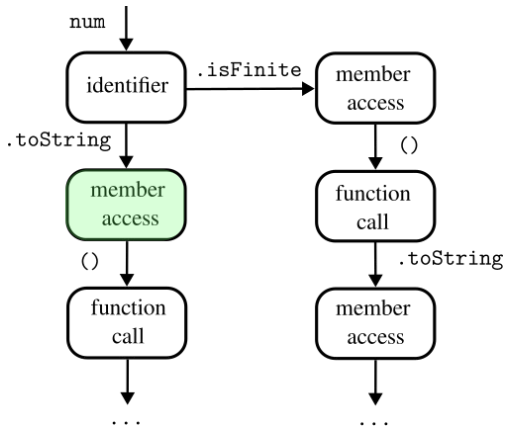
Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

Code

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

Incremental type checking

Automaton



Typing Environment

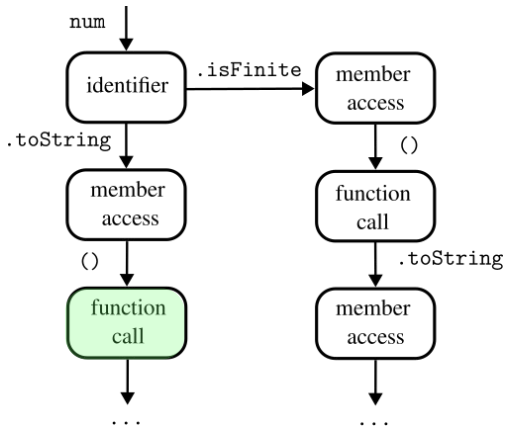
Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

Code

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

Incremental type checking

Automaton



Typing Environment

Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

Code

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

Incremental type checking

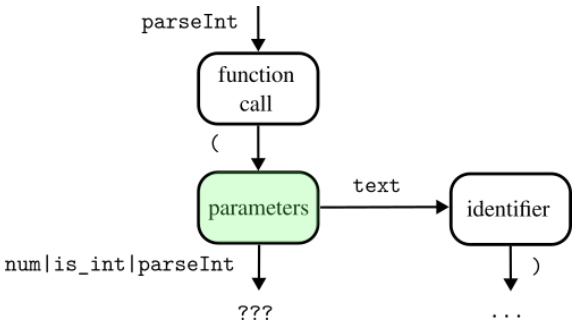
Automaton | Typing Environment | Code

Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

```
function is_int(text: string): boolean {
  const num = Number(text);
  return !isNaN(num) &&
    parseInt(num.toString()) === num
}
```

Rewind: Addressing type inhabitation

Automaton



Typing Environment

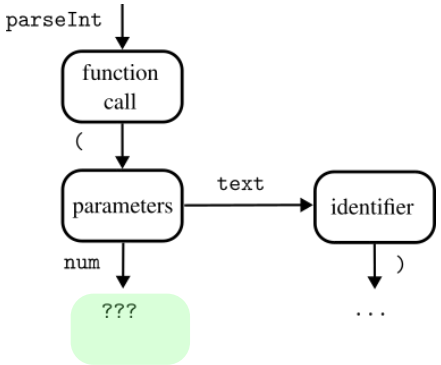
Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

Code

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

Rewind: Addressing type inhabitation

Automaton



Typing Environment

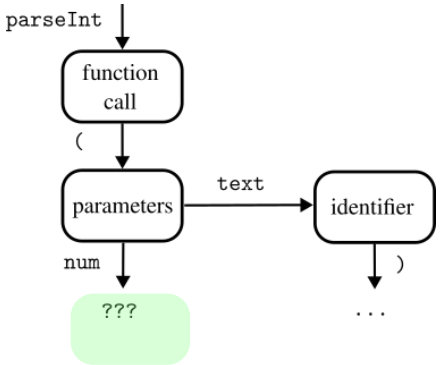
Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

Code

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

Rewind: Addressing type inhabitation

Automaton



Typing Environment

Identifier	Type
parseInt	(string) => number
text	string
is_int	(string) => boolean
num	number

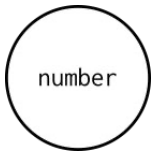
Code

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

Check if we *can inhabit* the goal type with a completion of the current partial expression

Type reachability search

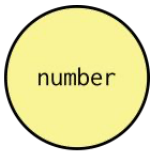
Can we append operators to turn type number into string?




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

Type reachability search

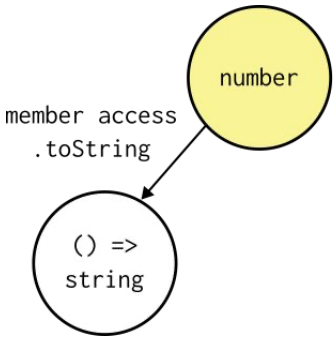
Can we append operators to turn type number into string?




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

Type reachability search

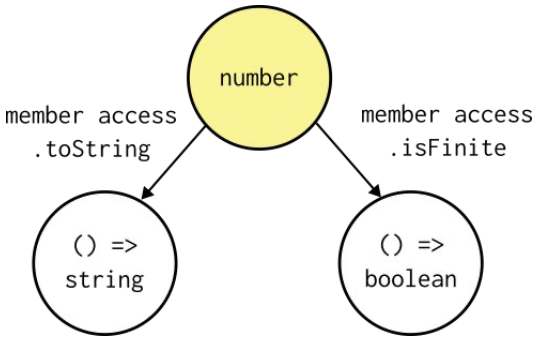
Can we append operators to turn type number into string?




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

Type reachability search

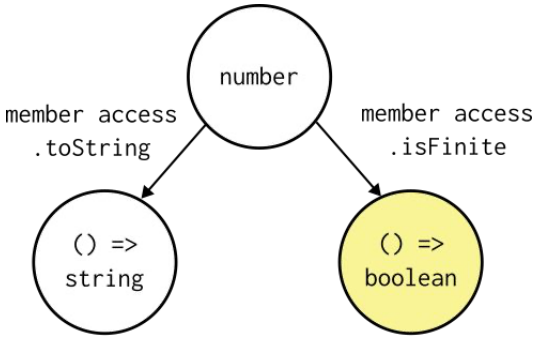
Can we append operators to turn type number into string?




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

Type reachability search

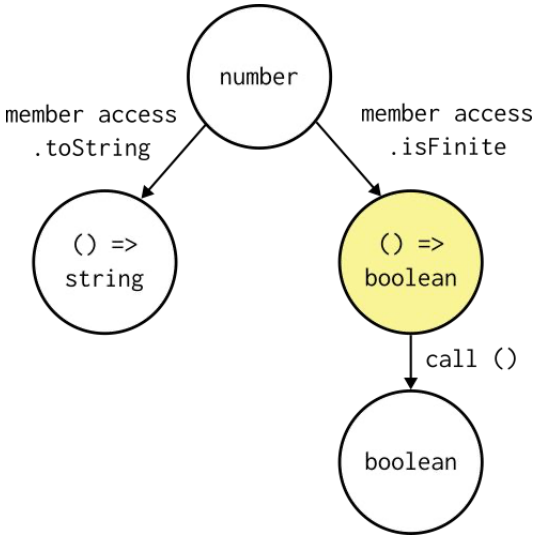
Can we append operators to turn type number into string?




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

Type reachability search

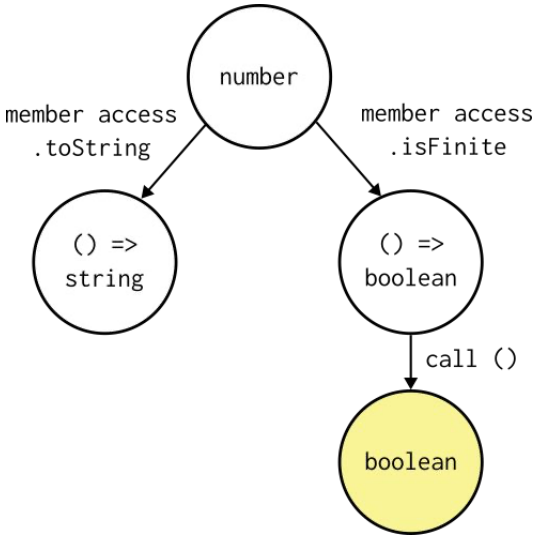
Can we append operators to turn type number into string?




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

Type reachability search

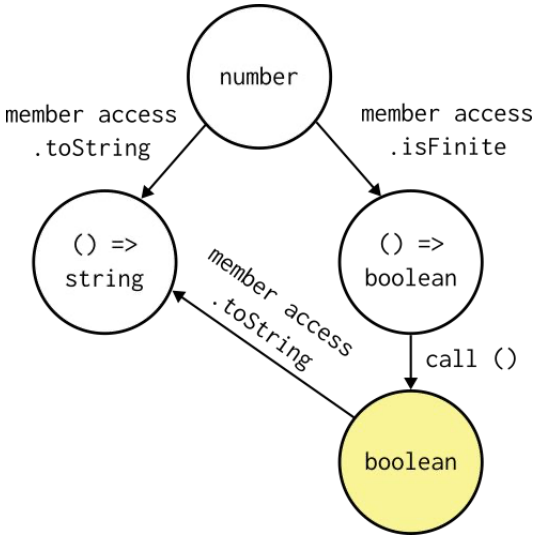
Can we append operators to turn type number into string?




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

Type reachability search

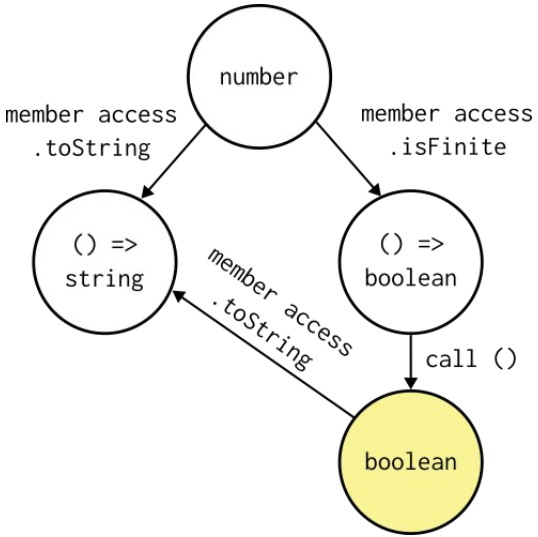
Can we append operators to turn type number into string?




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

Type reachability search

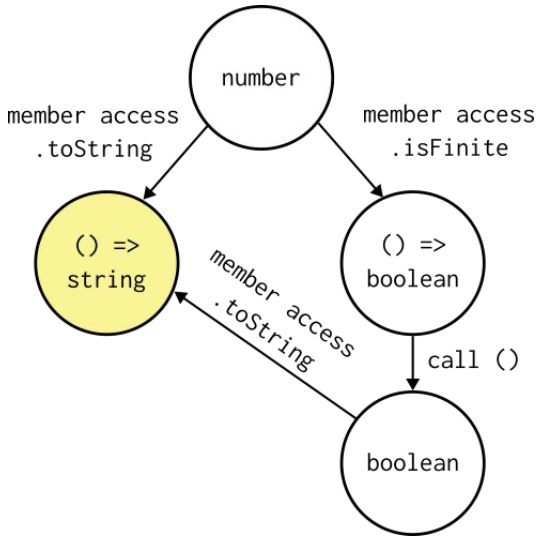
Can we append operators to turn type number into string?




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

Type reachability search

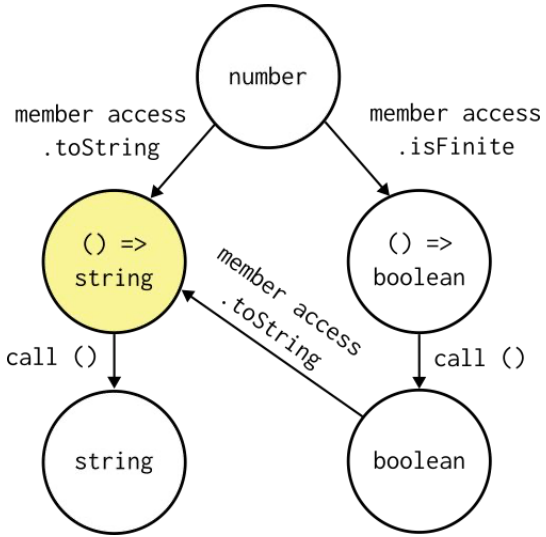
Can we append operators to turn type number into string?




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

Type reachability search

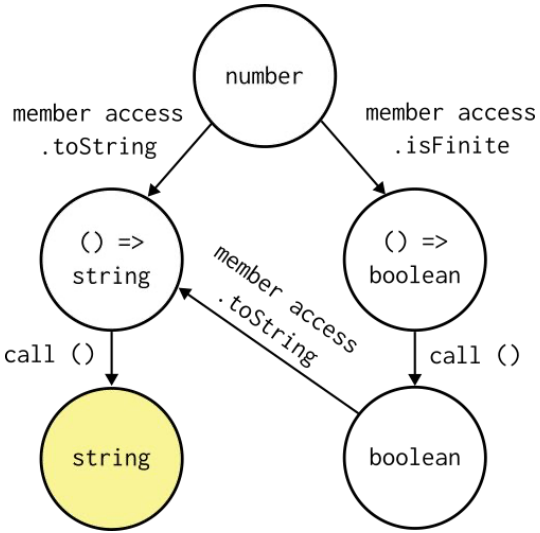
Can we append operators to turn type number into string?




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

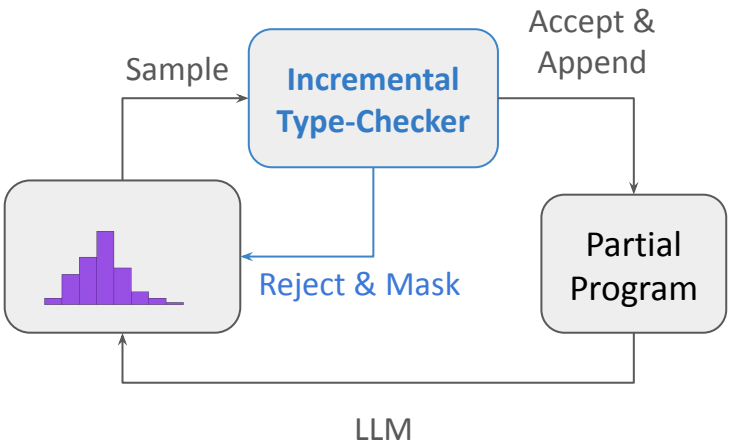
Type reachability search

Can we append operators to turn type number into string?




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

Type-Constrained LLM code generation: benefits



One forward LLM inference pass, no backtracking

Guaranteed type safety on termination

Fully leverages type system during decoding

Requires manually coming up with the automata from the language and its type system definitions

Experimental Evaluation

Drastic reduction of compiler errors

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	103	**44**$_{\downarrow 57.3\%}$	177	**80**$_{\downarrow 54.8\%}$	194	**103**$_{\downarrow 46.9\%}$
	Gemma 2 9B	45	**13**$_{\downarrow 71.1\%}$	75	**16**$_{\downarrow 78.7\%}$	113	**52**$_{\downarrow 54.0\%}$
	Gemma 2 27B	15	**2**$_{\downarrow 86.7\%}$	20	**3**$_{\downarrow 85.0\%}$	45	**22**$_{\downarrow 51.1\%}$
	DS Coder 33B	26	**5**$_{\downarrow 80.8\%}$	18	**7**$_{\downarrow 61.1\%}$	36	**15**$_{\downarrow 58.3\%}$
	CodeLlama 34B	86	**28**$_{\downarrow 67.4\%}$	158	**59**$_{\downarrow 62.7\%}$	153	**48**$_{\downarrow 68.6\%}$
	Qwen2.5 32B	17	**2**$_{\downarrow 88.2\%}$	24	**5**$_{\downarrow 79.2\%}$	36	**13**$_{\downarrow 63.9\%}$
MBPP	Gemma 2 2B	67	**27**$_{\downarrow 59.7\%}$	126	**79**$_{\downarrow 37.3\%}$	194	**108**$_{\downarrow 44.3\%}$
	Gemma 2 9B	30	**10**$_{\downarrow 66.7\%}$	67	**33**$_{\downarrow 50.7\%}$	129	**63**$_{\downarrow 51.2\%}$
	Gemma 2 27B	20	**7**$_{\downarrow 65.0\%}$	37	**22**$_{\downarrow 40.5\%}$	71	**32**$_{\downarrow 54.9\%}$
	DS Coder 33B	32	**19**$_{\downarrow 40.6\%}$	29	**13**$_{\downarrow 55.2\%}$	90	**43**$_{\downarrow 52.2\%}$
	CodeLlama 34B	80	**41**$_{\downarrow 48.8\%}$	126	**54**$_{\downarrow 57.1\%}$	157	**76**$_{\downarrow 51.6\%}$
	Qwen2.5 32B	19	**13**$_{\downarrow 31.6\%}$	22	**16**$_{\downarrow 27.3\%}$	55	**29**$_{\downarrow 47.3\%}$

Drastic reduction of compiler errors

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	103	**44**$_{\downarrow 57.3\%}$	177	**80**$_{\downarrow 54.8\%}$	194	**103**$_{\downarrow 46.9\%}$
	Gemma 2 9B	45	**13**$_{\downarrow 71.1\%}$	75	**16**$_{\downarrow 78.7\%}$	113	**52**$_{\downarrow 54.0\%}$
	Gemma 2 27B	15	**2**$_{\downarrow 86.7\%}$	20	**3**$_{\downarrow 85.0\%}$	45	**22**$_{\downarrow 51.1\%}$
	DS Coder 33B	26	**5**$_{\downarrow 80.8\%}$	18	**7**$_{\downarrow 61.1\%}$	36	**15**$_{\downarrow 58.3\%}$
	CodeLlama 34B	86	**28**$_{\downarrow 67.4\%}$	158	**59**$_{\downarrow 62.7\%}$	153	**48**$_{\downarrow 68.6\%}$
	Qwen2.5 32B	17	**2**$_{\downarrow 88.2\%}$	24	**5**$_{\downarrow 79.2\%}$	36	**13**$_{\downarrow 63.9\%}$
MBPP	Gemma 2 2B	67	**27**$_{\downarrow 59.7\%}$	126	**79**$_{\downarrow 37.3\%}$	194	**108**$_{\downarrow 44.3\%}$
	Gemma 2 9B	30	**10**$_{\downarrow 66.7\%}$	67	**33**$_{\downarrow 50.7\%}$	129	**63**$_{\downarrow 51.2\%}$
	Gemma 2 27B	20	**7**$_{\downarrow 65.0\%}$	37	**22**$_{\downarrow 40.5\%}$	71	**32**$_{\downarrow 54.9\%}$
	DS Coder 33B	32	**19**$_{\downarrow 40.6\%}$	29	**13**$_{\downarrow 55.2\%}$	90	**43**$_{\downarrow 52.2\%}$
	CodeLlama 34B	80	**41**$_{\downarrow 48.8\%}$	126	**54**$_{\downarrow 57.1\%}$	157	**76**$_{\downarrow 51.6\%}$
	Qwen2.5 32B	19	**13**$_{\downarrow 31.6\%}$	22	**16**$_{\downarrow 27.3\%}$	55	**29**$_{\downarrow 47.3\%}$

Drastic reduction of compiler errors

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	103	**44**$_{\downarrow 57.3\%}$	177	**80**$_{\downarrow 54.8\%}$	194	**103**$_{\downarrow 46.9\%}$
	Gemma 2 9B	45	**13**$_{\downarrow 71.1\%}$	75	**16**$_{\downarrow 78.7\%}$	113	**52**$_{\downarrow 54.0\%}$
	Gemma 2 27B	15	**2**$_{\downarrow 86.7\%}$	20	**3**$_{\downarrow 85.0\%}$	45	**22**$_{\downarrow 51.1\%}$
	DS Coder 33B	26	**5**$_{\downarrow 80.8\%}$	18	**7**$_{\downarrow 61.1\%}$	36	**15**$_{\downarrow 58.3\%}$
	CodeLlama 34B	86	**28**$_{\downarrow 67.4\%}$	158	**59**$_{\downarrow 62.7\%}$	153	**48**$_{\downarrow 68.6\%}$
	Qwen2.5 32B	17	**2**$_{\downarrow 88.2\%}$	24	**5**$_{\downarrow 79.2\%}$	36	**13**$_{\downarrow 63.9\%}$
MBPP	Gemma 2 2B	67	**27**$_{\downarrow 59.7\%}$	126	**79**$_{\downarrow 37.3\%}$	194	**108**$_{\downarrow 44.3\%}$
	Gemma 2 9B	30	**10**$_{\downarrow 66.7\%}$	67	**33**$_{\downarrow 50.7\%}$	129	**63**$_{\downarrow 51.2\%}$
	Gemma 2 27B	20	**7**$_{\downarrow 65.0\%}$	37	**22**$_{\downarrow 40.5\%}$	71	**32**$_{\downarrow 54.9\%}$
	DS Coder 33B	32	**19**$_{\downarrow 40.6\%}$	29	**13**$_{\downarrow 55.2\%}$	90	**43**$_{\downarrow 52.2\%}$
	CodeLlama 34B	80	**41**$_{\downarrow 48.8\%}$	126	**54**$_{\downarrow 57.1\%}$	157	**76**$_{\downarrow 51.6\%}$
	Qwen2.5 32B	19	**13**$_{\downarrow 31.6\%}$	22	**16**$_{\downarrow 27.3\%}$	55	**29**$_{\downarrow 47.3\%}$

Drastic reduction of compiler errors

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	103	**44**$_{\downarrow 57.3\%}$	177	**80**$_{\downarrow 54.8\%}$	194	**103**$_{\downarrow 46.9\%}$
	Gemma 2 9B	45	**13**$_{\downarrow 71.1\%}$	75	**16**$_{\downarrow 78.7\%}$	113	**52**$_{\downarrow 54.0\%}$
	Gemma 2 27B	15	**2**$_{\downarrow 86.7\%}$	20	**3**$_{\downarrow 85.0\%}$	45	**22**$_{\downarrow 51.1\%}$
	DS Coder 33B	26	**5**$_{\downarrow 80.8\%}$	18	**7**$_{\downarrow 61.1\%}$	36	**15**$_{\downarrow 58.3\%}$
	CodeLlama 34B	86	**28**$_{\downarrow 67.4\%}$	158	**59**$_{\downarrow 62.7\%}$	153	**48**$_{\downarrow 68.6\%}$
	Qwen2.5 32B	17	**2**$_{\downarrow 88.2\%}$	24	**5**$_{\downarrow 79.2\%}$	36	**13**$_{\downarrow 63.9\%}$
MBPP	Gemma 2 2B	67	**27**$_{\downarrow 59.7\%}$	126	**79**$_{\downarrow 37.3\%}$	194	**108**$_{\downarrow 44.3\%}$
	Gemma 2 9B	30	**10**$_{\downarrow 66.7\%}$	67	**33**$_{\downarrow 50.7\%}$	129	**63**$_{\downarrow 51.2\%}$
	Gemma 2 27B	20	**7**$_{\downarrow 65.0\%}$	37	**22**$_{\downarrow 40.5\%}$	71	**32**$_{\downarrow 54.9\%}$
	DS Coder 33B	32	**19**$_{\downarrow 40.6\%}$	29	**13**$_{\downarrow 55.2\%}$	90	**43**$_{\downarrow 52.2\%}$
	CodeLlama 34B	80	**41**$_{\downarrow 48.8\%}$	126	**54**$_{\downarrow 57.1\%}$	157	**76**$_{\downarrow 51.6\%}$
	Qwen2.5 32B	19	**13**$_{\downarrow 31.6\%}$	22	**16**$_{\downarrow 27.3\%}$	55	**29**$_{\downarrow 47.3\%}$

Drastic reduction of compiler errors

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	103	**44**$_{\downarrow 57.3\%}$	177	**80**$_{\downarrow 54.8\%}$	194	**103**$_{\downarrow 46.9\%}$
	Gemma 2 9B	45	**13**$_{\downarrow 71.1\%}$	75	**16**$_{\downarrow 78.7\%}$	113	**52**$_{\downarrow 54.0\%}$
	Gemma 2 27B	15	**2**$_{\downarrow 86.7\%}$	20	**3**$_{\downarrow 85.0\%}$	45	**22**$_{\downarrow 51.1\%}$
	DS Coder 33B	26	**5**$_{\downarrow 80.8\%}$	18	**7**$_{\downarrow 61.1\%}$	36	**15**$_{\downarrow 58.3\%}$
	CodeLlama 34B	86	**28**$_{\downarrow 67.4\%}$	158	**59**$_{\downarrow 62.7\%}$	153	**48**$_{\downarrow 68.6\%}$
	Qwen2.5 32B	17	**2**$_{\downarrow 88.2\%}$	24	**5**$_{\downarrow 79.2\%}$	36	**13**$_{\downarrow 63.9\%}$
MBPP	Gemma 2 2B	67	**27**$_{\downarrow 59.7\%}$	126	**79**$_{\downarrow 37.3\%}$	194	**108**$_{\downarrow 44.3\%}$
	Gemma 2 9B	30	**10**$_{\downarrow 66.7\%}$	67	**33**$_{\downarrow 50.7\%}$	129	**63**$_{\downarrow 51.2\%}$
	Gemma 2 27B	20	**7**$_{\downarrow 65.0\%}$	37	**22**$_{\downarrow 40.5\%}$	71	**32**$_{\downarrow 54.9\%}$
	DS Coder 33B	32	**19**$_{\downarrow 40.6\%}$	29	**13**$_{\downarrow 55.2\%}$	90	**43**$_{\downarrow 52.2\%}$
	CodeLlama 34B	80	**41**$_{\downarrow 48.8\%}$	126	**54**$_{\downarrow 57.1\%}$	157	**76**$_{\downarrow 51.6\%}$
	Qwen2.5 32B	19	**13**$_{\downarrow 31.6\%}$	22	**16**$_{\downarrow 27.3\%}$	55	**29**$_{\downarrow 47.3\%}$

Drastic reduction of compiler errors

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	103	**44**$_{\downarrow 57.3\%}$	177	**80**$_{\downarrow 54.8\%}$	194	**103**$_{\downarrow 46.9\%}$
	Gemma 2 9B	45	**13**$_{\downarrow 71.1\%}$	75	**16**$_{\downarrow 78.7\%}$	113	**52**$_{\downarrow 54.0\%}$
	Gemma 2 27B	15	**2**$_{\downarrow 86.7\%}$	20	**3**$_{\downarrow 85.0\%}$	45	**22**$_{\downarrow 51.1\%}$
	DS Coder 33B	26	**5**$_{\downarrow 80.8\%}$	18	**7**$_{\downarrow 61.1\%}$	36	**15**$_{\downarrow 58.3\%}$
	CodeLlama 34B	86	**28**$_{\downarrow 67.4\%}$	158	**59**$_{\downarrow 62.7\%}$	153	**48**$_{\downarrow 68.6\%}$
	Qwen2.5 32B	17	**2**$_{\downarrow 88.2\%}$	24	**5**$_{\downarrow 79.2\%}$	36	**13**$_{\downarrow 63.9\%}$
MBPP	Gemma 2 2B	67	**27**$_{\downarrow 59.7\%}$	126	**79**$_{\downarrow 37.3\%}$	194	**108**$_{\downarrow 44.3\%}$
	Gemma 2 9B	30	**10**$_{\downarrow 66.7\%}$	67	**33**$_{\downarrow 50.7\%}$	129	**63**$_{\downarrow 51.2\%}$
	Gemma 2 27B	20	**7**$_{\downarrow 65.0\%}$	37	**22**$_{\downarrow 40.5\%}$	71	**32**$_{\downarrow 54.9\%}$
	DS Coder 33B	32	**19**$_{\downarrow 40.6\%}$	29	**13**$_{\downarrow 55.2\%}$	90	**43**$_{\downarrow 52.2\%}$
	CodeLlama 34B	80	**41**$_{\downarrow 48.8\%}$	126	**54**$_{\downarrow 57.1\%}$	157	**76**$_{\downarrow 51.6\%}$
	Qwen2.5 32B	19	**13**$_{\downarrow 31.6\%}$	22	**16**$_{\downarrow 27.3\%}$	55	**29**$_{\downarrow 47.3\%}$

Drastic reduction of compiler errors

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	103	**44**$_{\downarrow 57.3\%}$	177	**80**$_{\downarrow 54.8\%}$	194	**103**$_{\downarrow 46.9\%}$
	Gemma 2 9B	45	**13**$_{\downarrow 71.1\%}$	75	**16**$_{\downarrow 78.7\%}$	113	**52**$_{\downarrow 54.0\%}$
	Gemma 2 27B	15	**2**$_{\downarrow 86.7\%}$	20	**3**$_{\downarrow 85.0\%}$	45	**22**$_{\downarrow 51.1\%}$
	DS Coder 33B	26	**5**$_{\downarrow 80.8\%}$	18	**7**$_{\downarrow 61.1\%}$	36	**15**$_{\downarrow 58.3\%}$
	CodeLlama 34B	86	**28**$_{\downarrow 67.4\%}$	158	**59**$_{\downarrow 62.7\%}$	153	**48**$_{\downarrow 68.6\%}$
	Qwen2.5 32B	17	**2**$_{\downarrow 88.2\%}$	24	**5**$_{\downarrow 79.2\%}$	36	**13**$_{\downarrow 63.9\%}$
MBPP	Gemma 2 2B	67	**27**$_{\downarrow 59.7\%}$	126	**79**$_{\downarrow 37.3\%}$	194	**108**$_{\downarrow 44.3\%}$
	Gemma 2 9B	30	**10**$_{\downarrow 66.7\%}$	67	**33**$_{\downarrow 50.7\%}$	129	**63**$_{\downarrow 51.2\%}$
	Gemma 2 27B	20	**7**$_{\downarrow 65.0\%}$	37	**22**$_{\downarrow 40.5\%}$	71	**32**$_{\downarrow 54.9\%}$
	DS Coder 33B	32	**19**$_{\downarrow 40.6\%}$	29	**13**$_{\downarrow 55.2\%}$	90	**43**$_{\downarrow 52.2\%}$
	CodeLlama 34B	80	**41**$_{\downarrow 48.8\%}$	126	**54**$_{\downarrow 57.1\%}$	157	**76**$_{\downarrow 51.6\%}$
	Qwen2.5 32B	19	**13**$_{\downarrow 31.6\%}$	22	**16**$_{\downarrow 27.3\%}$	55	**29**$_{\downarrow 47.3\%}$

Improvement of functional correctness

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	29.1	**30.2**$_{\uparrow 3.8\%}$	50.2	**53.9**$_{\uparrow 7.5\%}$	11.6	**20.9**$_{\uparrow 79.4\%}$
	Gemma 2 9B	56.6	**58.3**$_{\uparrow 3.1\%}$	73.7	**78.3**$_{\uparrow 6.2\%}$	24.0	**34.9**$_{\uparrow 45.7\%}$
	Gemma 2 27B	69.5	**71.2**$_{\uparrow 2.5\%}$	86.6	**87.7**$_{\uparrow 1.3\%}$	38.4	**41.1**$_{\uparrow 7.1\%}$
	DS Coder 33B	68.9	**71.1**$_{\uparrow 3.2\%}$	88.7	**90.1**$_{\uparrow 1.6\%}$	47.6	**50.7**$_{\uparrow 6.5\%}$
	CodeLlama 34B	41.0	**43.4**$_{\uparrow 5.7\%}$	58.6	**63.5**$_{\uparrow 8.3\%}$	17.5	**27.4**$_{\uparrow 56.9\%}$
	Qwen2.5 32B	79.6	**81.8**$_{\uparrow 2.8\%}$	92.1	**93.9**$_{\uparrow 1.9\%}$	65.4	**71.2**$_{\uparrow 8.9\%}$
MBPP	Gemma 2 2B	40.4	**42.4**$_{\uparrow 5.2\%}$	52.3	**56.0**$_{\uparrow 7.0\%}$	12.1	**22.6**$_{\uparrow 86.7\%}$
	Gemma 2 9B	65.4	**67.4**$_{\uparrow 3.2\%}$	71.4	**75.8**$_{\uparrow 6.2\%}$	24.2	**31.9**$_{\uparrow 31.7\%}$
	Gemma 2 27B	70.6	**72.1**$_{\uparrow 2.2\%}$	83.1	**84.4**$_{\uparrow 1.6\%}$	39.1	**45.2**$_{\uparrow 15.5\%}$
	DS Coder 33B	65.4	**67.2**$_{\uparrow 2.8\%}$	85.9	**89.1**$_{\uparrow 3.6\%}$	35.1	**43.1**$_{\uparrow 23.0\%}$
	CodeLlama 34B	42.2	**45.6**$_{\uparrow 8.0\%}$	55.7	**63.3**$_{\uparrow 13.6\%}$	15.7	**26.6**$_{\uparrow 69.2\%}$
	Qwen2.5 32B	76.3	**76.6**$_{\uparrow 0.3\%}$	89.6	**90.4**$_{\uparrow 0.9\%}$	48.0	**54.0**$_{\uparrow 12.6\%}$

Future work

Extend to more TypeScript features

Construct incremental type-checker for other languages

Characterize the *kind* of type systems that *can* be incrementally verified

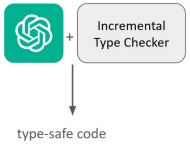
Conclusion

SOTA LLMs generate semantic errors and unsafe code

Our work: Type-augmented LLM decoders

Potential Benefits:

- Full control during decoding
- Generated code is guaranteed to be type safe
- Inference-time costs: predictable, one pass

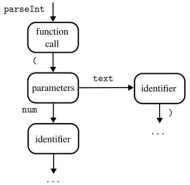


Wanted: new incremental type checkers for LLM decoding

Type-constrained code generation in LLMs: key ingredient

An automata tracked during LLM decoding that only accepts type safe programs

Technically, the automata is an AST augmented with type relevant context




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

Type-Constrained Code Generation with Language Models 24

Drastic reduction of compiler errors

		Synthesis		Translation		Repair	
	Model	Vanilla	Types	Vanilla	Types	Vanilla	Types
HumanEval	Gemma 2 2B	103	**44** $\downarrow_{57.3\%}$	177	**80** $\downarrow_{54.8\%}$	194	**103** $\downarrow_{46.9\%}$
	Gemma 2 9B	45	**13** $\downarrow_{71.1\%}$	75	**16** $\downarrow_{78.7\%}$	113	**52** $\downarrow_{54.0\%}$
	Gemma 2 27B	15	**2** $\downarrow_{86.7\%}$	20	**3** $\downarrow_{85.0\%}$	45	**22** $\downarrow_{51.1\%}$
	DS Coder 33B	26	**5** $\downarrow_{80.8\%}$	18	**7** $\downarrow_{61.1\%}$	36	**15** $\downarrow_{58.3\%}$
	CodeLlama 34B	86	**28** $\downarrow_{67.4\%}$	158	**59** $\downarrow_{62.7\%}$	153	**48** $\downarrow_{68.6\%}$
	Qwen2.5 32B	17	**2** $\downarrow_{88.2\%}$	24	**5** $\downarrow_{79.2\%}$	36	**13** $\downarrow_{63.9\%}$
MBPP	Gemma 2 2B	67	**27** $\downarrow_{59.7\%}$	126	**79** $\downarrow_{37.3\%}$	194	**108** $\downarrow_{44.3\%}$
	Gemma 2 9B	30	**10** $\downarrow_{66.7\%}$	67	**33** $\downarrow_{50.7\%}$	129	**63** $\downarrow_{51.2\%}$
	Gemma 2 27B	20	**7** $\downarrow_{65.0\%}$	37	**22** $\downarrow_{40.5\%}$	71	**32** $\downarrow_{54.9\%}$
	DS Coder 33B	32	**19** $\downarrow_{40.6\%}$	29	**13** $\downarrow_{55.2\%}$	90	**43** $\downarrow_{52.2\%}$
	CodeLlama 34B	80	**41** $\downarrow_{48.8\%}$	126	**54** $\downarrow_{57.1\%}$	157	**76** $\downarrow_{51.6\%}$
	Qwen2.5 32B	19	**13** $\downarrow_{31.6\%}$	22	**16** $\downarrow_{27.3\%}$	55	**29** $\downarrow_{47.3\%}$

https://github.com/eth-sri/type-constrained-code-generation