Correctness-Guaranteed Code Generation via Constrained Decoding

Lingxiao Li, Salar Rahili, Yiwei Zhao

Netflix Research

Structured Generation

Goal: Guide LLM to generate output that can be parsed programmatically

Typically means generating a JSON conforming to a given schema

- Applications: data extraction/generation, agentic tool use

Question: Can we extend structured generation to code?

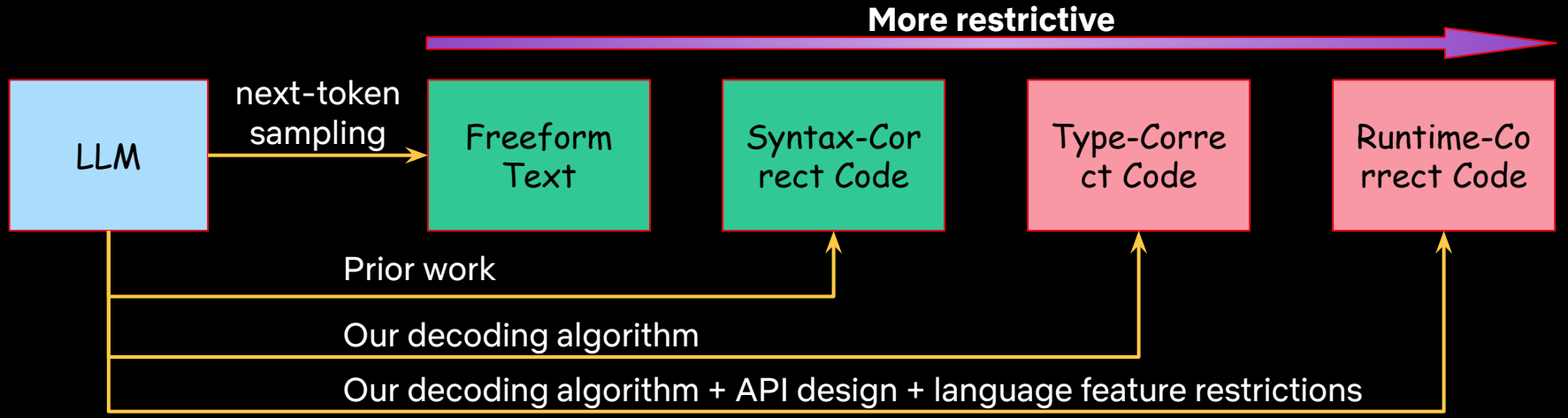
```json
{
  "characterName": "Elara Meadowlight",
  "class": "Archmage",
  "level": 87,
  "location": "Whispering Woods",
  "attributes": {
    "strength": 4,
    "wisdom": 18,
    "agility": 7
  },
  "inventory": [
    "Gnarled Oak Staff",
    "Pouch of Enchanted Seeds",
    "Tome of Ancient Runes"
  ]
}
```

JSON conforming to a schema

```
local Do: (Actor) -> boolean = function(user)
    return user.WithEnemySelected(GetRange(user), function(target)
        local damage_percent: number = CalculateDamagePercent(user);
        local damage: number = damage_percent * user.attack_power;

        local health_sacrifice_percent: number = CalculateHealthSacrificePercent(user);
        local health_sacrifice: number = health_sacrifice_percent * user.health;
        user.UpdateHealth(-health_sacrifice);
        damage = damage + health_sacrifice;

        g_game.ResolveHit(target, damage, user);

        local bleed_damage: number = CalculateBleedDamage(user);
        g_effects.bleed.Apply(target, {
            bleed_damage = bleed_damage,
        }, 5 + user.GetTalentLevel());

        return true;
    end);
end;
```

Type-correct code conforming to an API and a type system

Level of Code Correctness



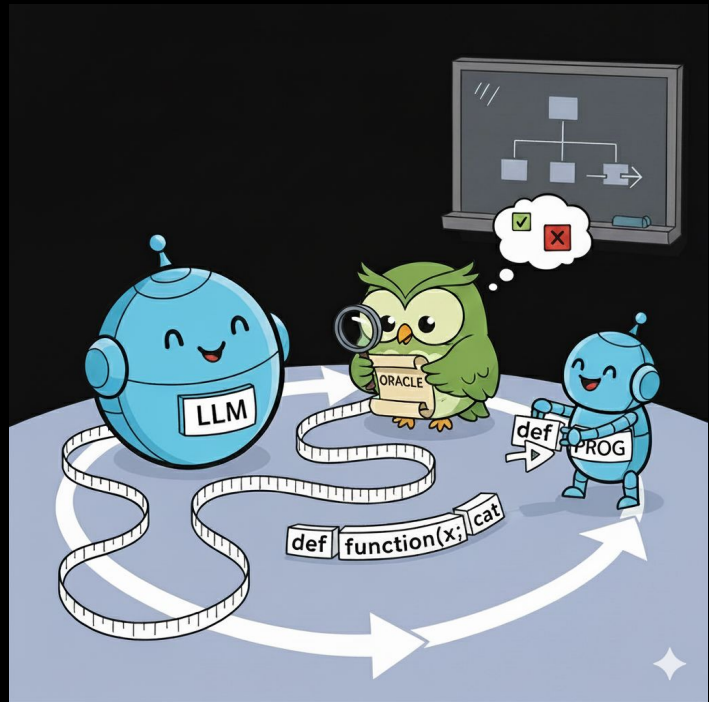
But wait: The Halting problem says runtime correctness is impossible!

- Need to restrict language features ⇒ resulting language still powerful (albeit not Turing-complete)

Constrained Decoding of LLMs

```
while not oracle.finished():
  prog = ""
  for t ~ LLM.next_token_prob(prog):
    if oracle.is_valid(t):
      oracle.accept(t)
     prog += t
     break
```

Question: How to build an oracle, a **prefix parser**, that incrementally guides generation towards type-correct code?



sLua

sLua: strongly typed, simplified version of Lua

- Type annotation
- Restricted memory access

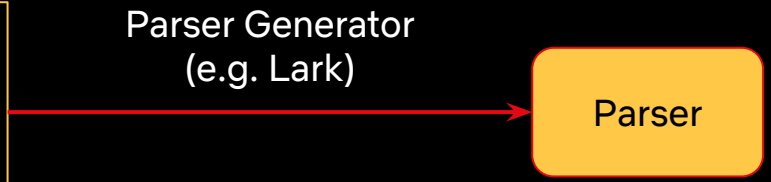
Goal: Build an oracle that only accepts **type-correct** prefix sLua programs.

```
local GetPoisonCount: (Actor) → number = function(user)
    return 2 + user.GetTalentLevel()
end

local range: number = 5
local duration: number = 5

NewTalent({
    name = "Catalyst",
    GetRange = function(user) return range end,
    GetCooldown = function(user) return 12 end,
    Do = function(user)
        return user.WithEnemySelected(range, function(target)
            local poisoned: boolean = Poisoned.WhenExists(
                target,
                function(param)
                    param.data.poison = param.data.poison * 2
                end)
            if not poisoned then
                Poisoned.Apply(target, {
                    poison = GetPoisonCount(user)
                }, duration)
            end
        end)
    end,
})
```

Background: CFG and Parsers

A **Context-Free Grammar (CFG)** consists of recursive rules and terminals (string literals or regexes).

```
expr: term (("+"|"-") term)*
term: factor (("*"|"/") factor)*
factor: NUMBER | "(" expr ")"
NUMBER: [1-9][0-9]*
```

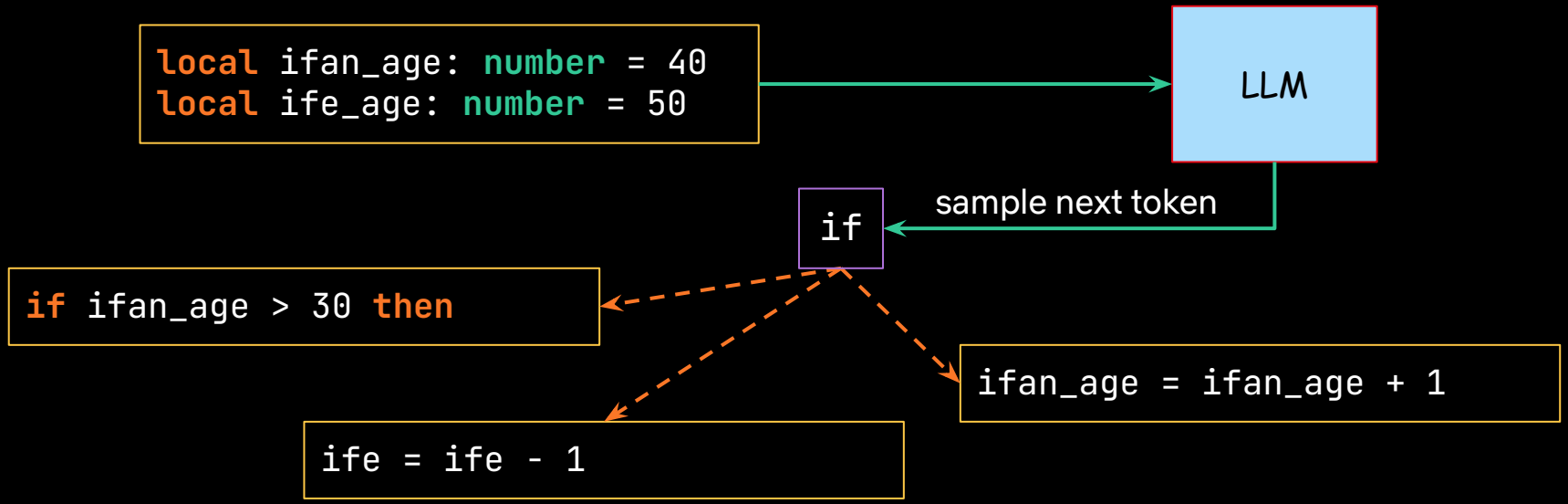


Assume the generated parser supports:

`Parser.feed(t)`: Advance parser state with terminal `t`

`Parser.accepts()`: Return the set of terminals that can be accepted next

Ambiguities of Prefix Programs



Parsing state after `if` is ambiguous! Yet we need to decide whether to accept right now

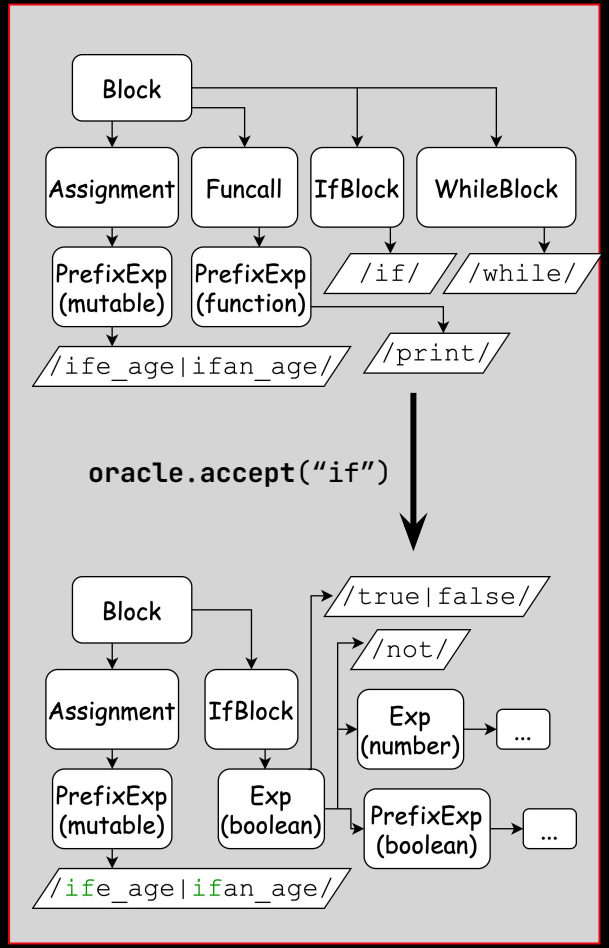
Right decision: accept if ≥ 1 valid continuation

Tree of Parsers: Overview

Main idea: Track **all** imminent possibilities

- Maintain a **tree of parsers** where each path from the root represents a possible parsing state
  - Each non-leaf node stores an environment (variable scopes) and a parser
  - Each leaf node is a finite state machine

Note: presented here is an improved version of our COLM paper



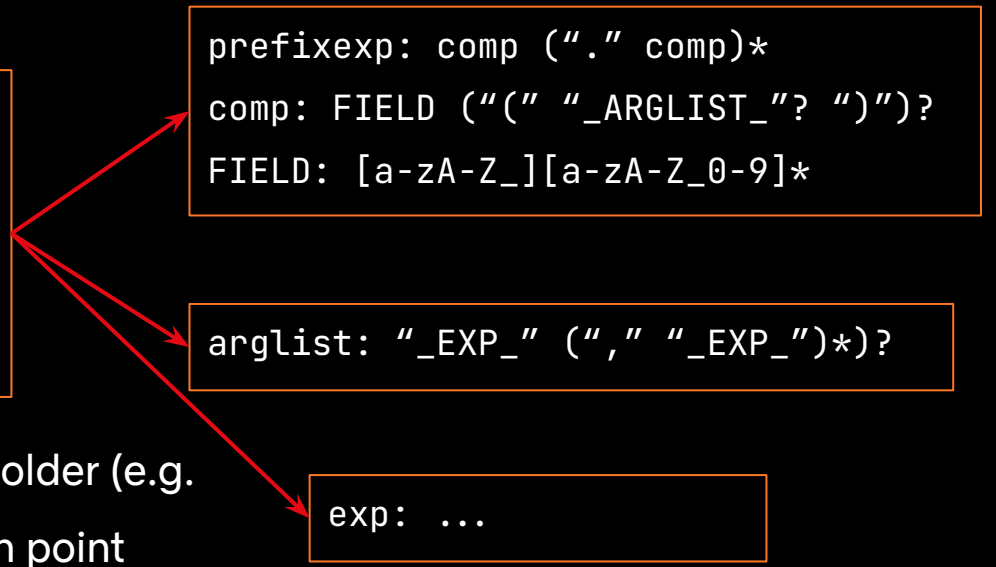
Decompose the Full CFG

The CFG of the target language is too permissive, so we first decompose it into smaller CFGs to have finer control.

Consider prefix expressions, e.g.,

```
a.foo().b.c().d.bar()
```

```
prefixexp: comp (“.” comp)*
comp: FIELD (“(” arglist? “)”)?
arglist: exp (“,” exp)*)?
exp: ...
FIELD: [a-zA-Z_][a-zA-Z_0-9]*
```



Decomposed CFGs contain special placeholder (e.g. `_ARGLIST_`) terminals, indicating recursion point

Core Tree Operations

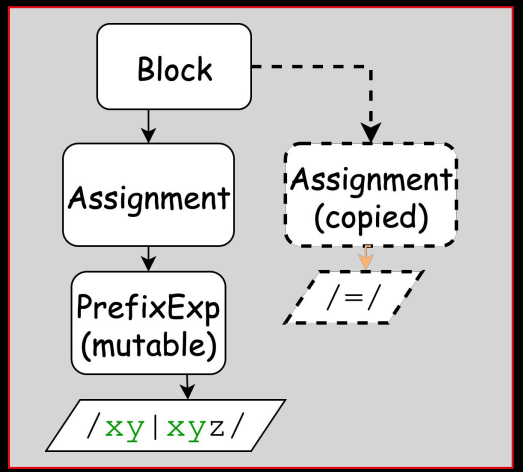
For a node `x` in the tree,

Prune: Delete branches that failed to accept the previous code segment

Spawn: Whenever a `_PLACEHOLDER_` appears in `x`'s `parser.accepts()`, add a child node of the proper type to `x`

Split: Whenever a child of `x` can end, add an updated copy of `x` to `x`'s parent

Split is essential: For instance, to distinguish assignment statement `xy = ...` vs `xyz = ...`, after parsing `xy`



Efficient splitting with **persistent data structures**.

Implement Per-Node Parsing Logic

The generated parser already takes care of syntax, so we only need to write context-sensitive logic.

Consider a prefix expression of target type T,

```
a.foo().b.c(arg1, arg2).d.bar()
```

We should

- Only allow field (or function call) that can eventually lead to T
  - Can be achieved by tracing the graph of types
- Only allow `.` if the preceding type is a table
- Pass the correct argument types when creating arglist nodes
- …

Only ~1k lines of Python code for implementing ToP for sLua!

Linear-Time Parsing of sLua

Linear-time parsing: ToP runs in **provably linear time** w.r.t. code length.

Intuition: Most branches in ToP will be pruned after a accepting few new characters.

Our (unoptimized) implementation: 2~3x as slow as unconstrained generation (Qwen Coder + vLLM)

From Type Correctness to Runtime Correctness

Goal: How can we restrict the sLua language features and the API (i.e. global variables/functions) to ensure **type correctness ⇒ runtime correctness**?

Observation: Most runtime errors are due to invalid memory access.

- Let's just forbid null pointers and dynamic data structure in the generated code!

Almost there, but non-termination can occur:

- infinite recursion ⇒ function definition follows strict lexical scoping
- infinite loop ⇒ cap maximum number of iterations

Workarounds Using Callbacks

Idea: Hide memory access inside API implementation and pass in callbacks.

Eliminate null references:

```
local actor = level.GetActorAt(coord)
if actor then
    actor.UpdateHealth(-5)
end
```



```
level.WithActorAt(coord, function(actor)
    actor.UpdateHealth(-5)
end)
```

Callback for range access:

```
level.ProjectBall(target_coord, radius, function(actor)
    if actor.faction ~= user.faction then
        actor.TakeDamage(5)
    end
end)
```

Alternatives: functional/declarative programming, dataflow paradigm

Application: Personalized Gameplay

Dungeon Crawler Infinite: A roguelike game with generated talents (abilities) and effects (buffs/debuffs) as sLua code:

- Talent script: Contains a `Do` function
- Effect script: Tracks state, implements hook functions (e.g. `OnTurnEnd`, `OnDamageTaken`)

Game scripting API exposed 8 compound types and 55 functions.

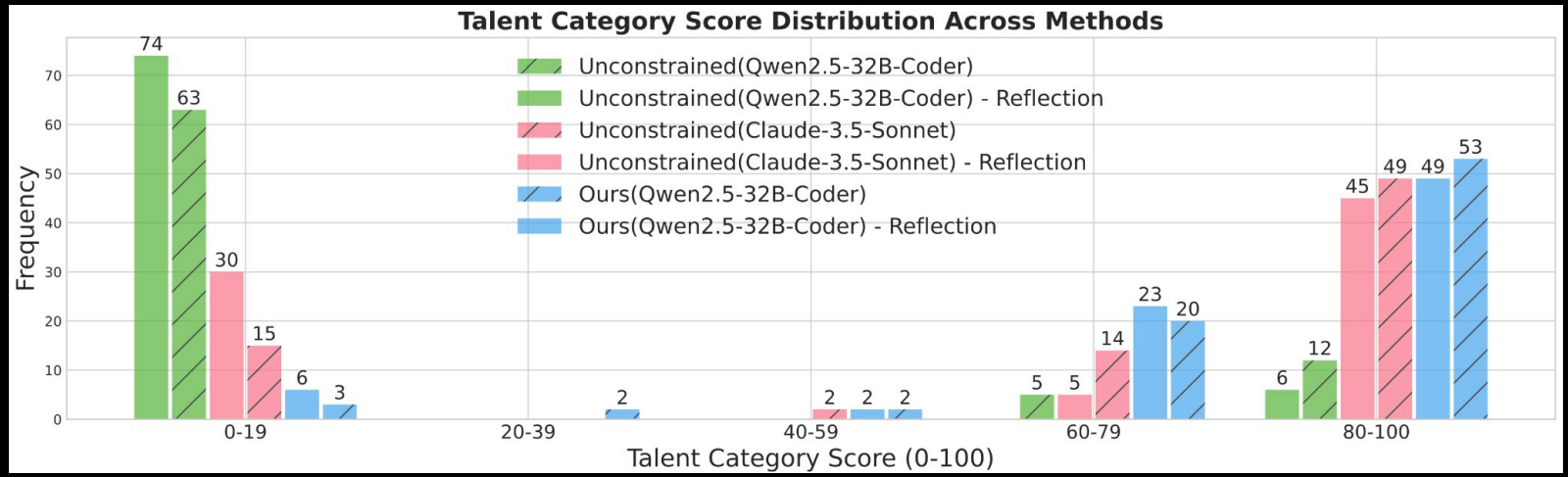
Generated sLua code using ToP is **provably runtime error-free**!

```
NewTalent({
  name = "Catalyst",
  GetRange = function(user) return range end,
  GetCooldown = function(user) return 12 end,
  Do = function(user)
    return user.WithEnemySelected(
      range,
      function(target)
        local poisoned: boolean = Poisoned.WhenExists(
          target,
          function(param)
            param.data.poison = param.data.poison * 2
          end)
        if not poisoned then
          Poisoned.Apply(target, {
            poison = GetPoisonCount(user)
          }, duration)
        end
      end)
    end,
})
```

Evaluation

Task: Generate a category of 4 synergistic talents and effects.

Result: Open-source LLM + our decoding algorithm can rival the best closed-source model in terms of correctness!



Can we use our algorithm to improve the model itself?

– Yes! Our method can serve as a token-level reward signal

Would the proposed method distort distribution just like other constrained decoding methods?

- It would. Recent techniques such as SMC could mitigate this

Thanks!