Making The Most Out Of Large Language Models For Program Synthesis

(when you want correct results)

Elizabeth Polgreen
University of Edinburgh
SYNT 2024

Is formal synthesis dead yet?

Two methods for making the most out of LLMs in synthesis:

- Solving formal synthesis by guiding enumerative synthesis with LLMs
- Generating syntactically correct models for verification, via LLMs, synthetic programming elicitation and Max-SMT solvers

Guiding Enumerative Synthesis with Large Language Models

Yixuan Li

Julian Parsert

Elizabeth Polgreen

yixuan.li.cs@ed.ac.uk

CAV, Saturday 27th 4pm

Can LLMs solve formal synthesis problems?

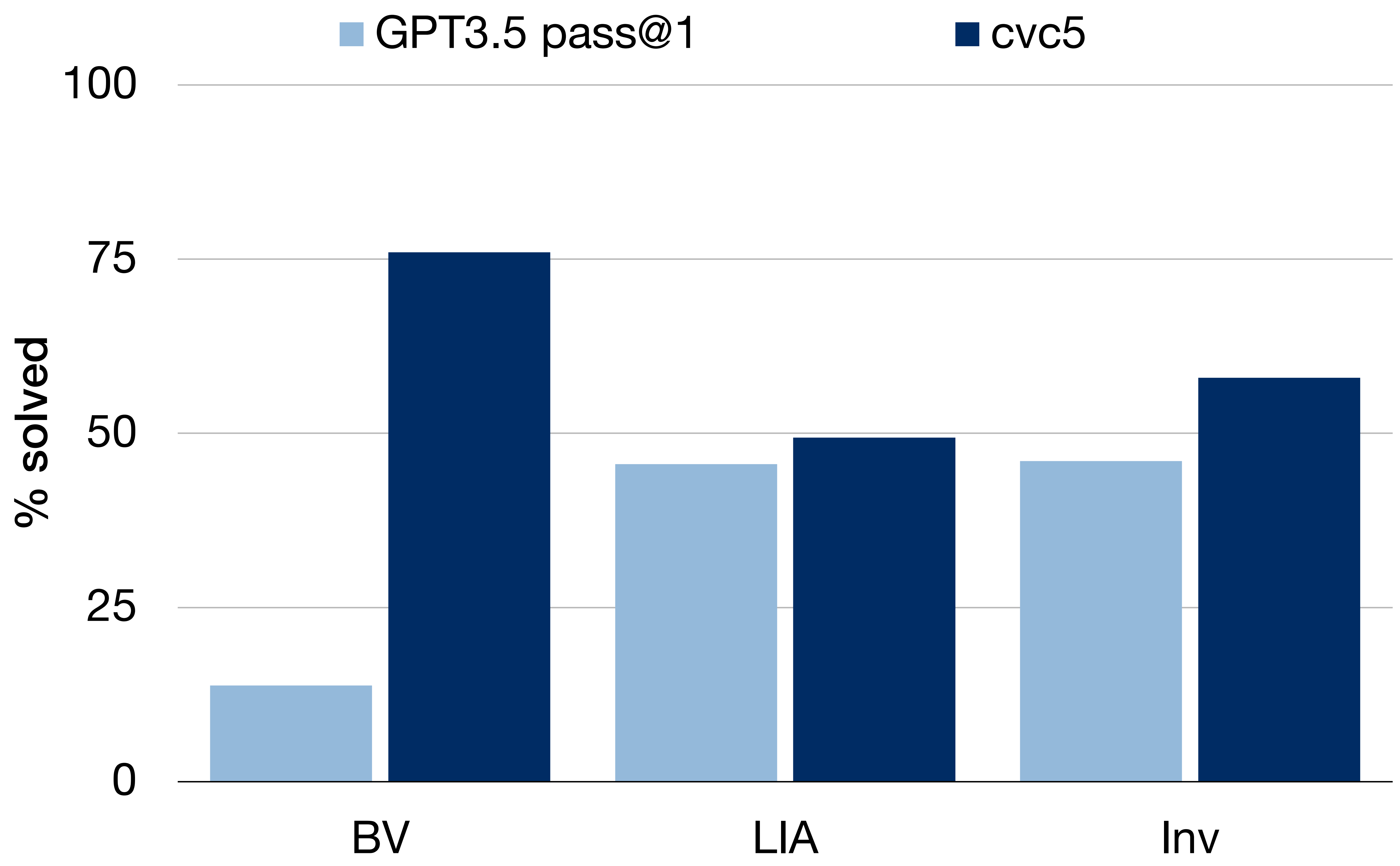
Formal Program Synthesis

$$\exists P \forall x. \sigma(P, x)$$

Does there exist a function P such that, for all possible inputs x, the specification σ will evaluate to true for P and x.

σ is a quantifier free formula in a background theory, e.g., Linear Integer Arithmetic

Can LLMs solve formal synthesis problems?

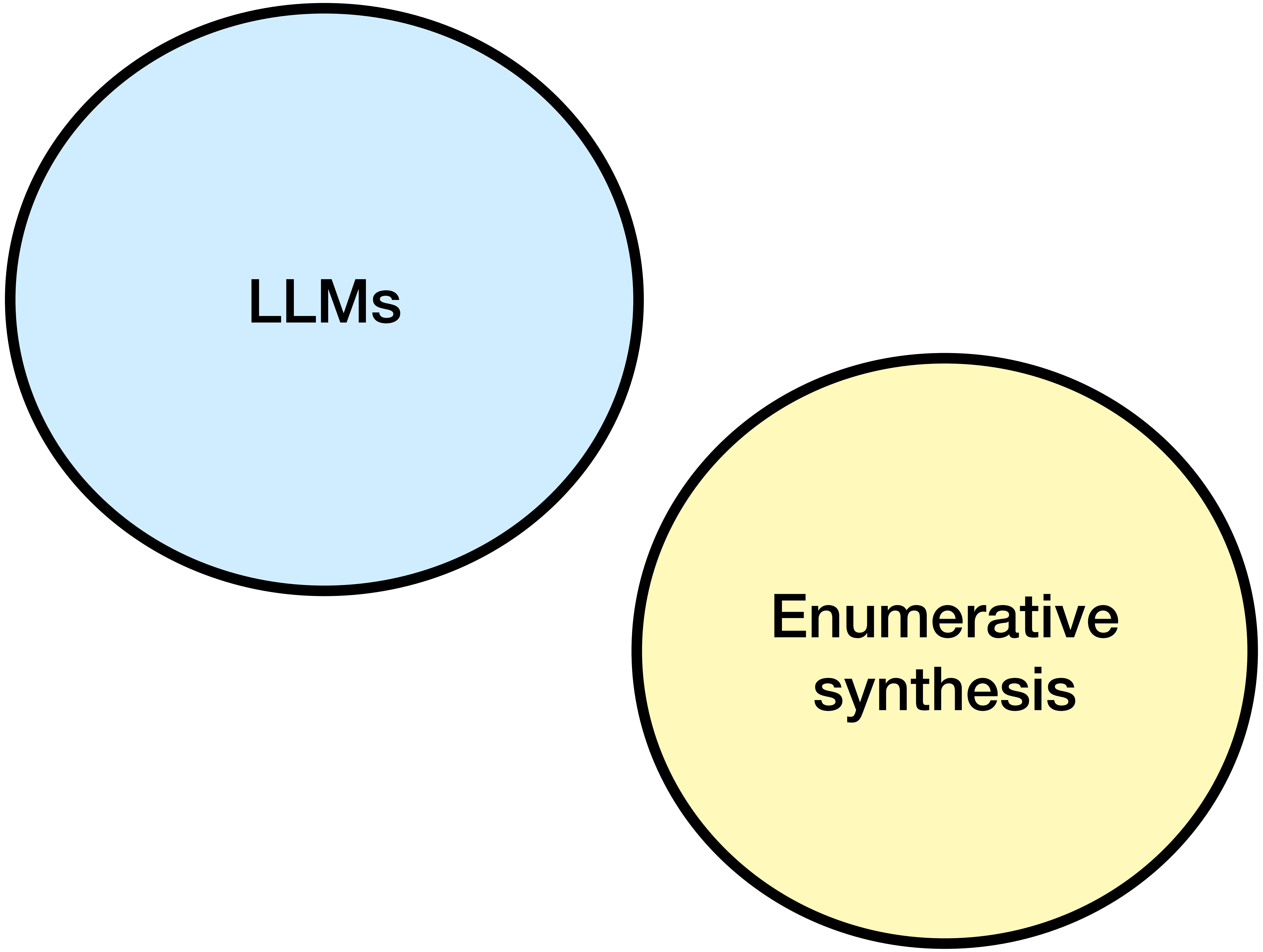


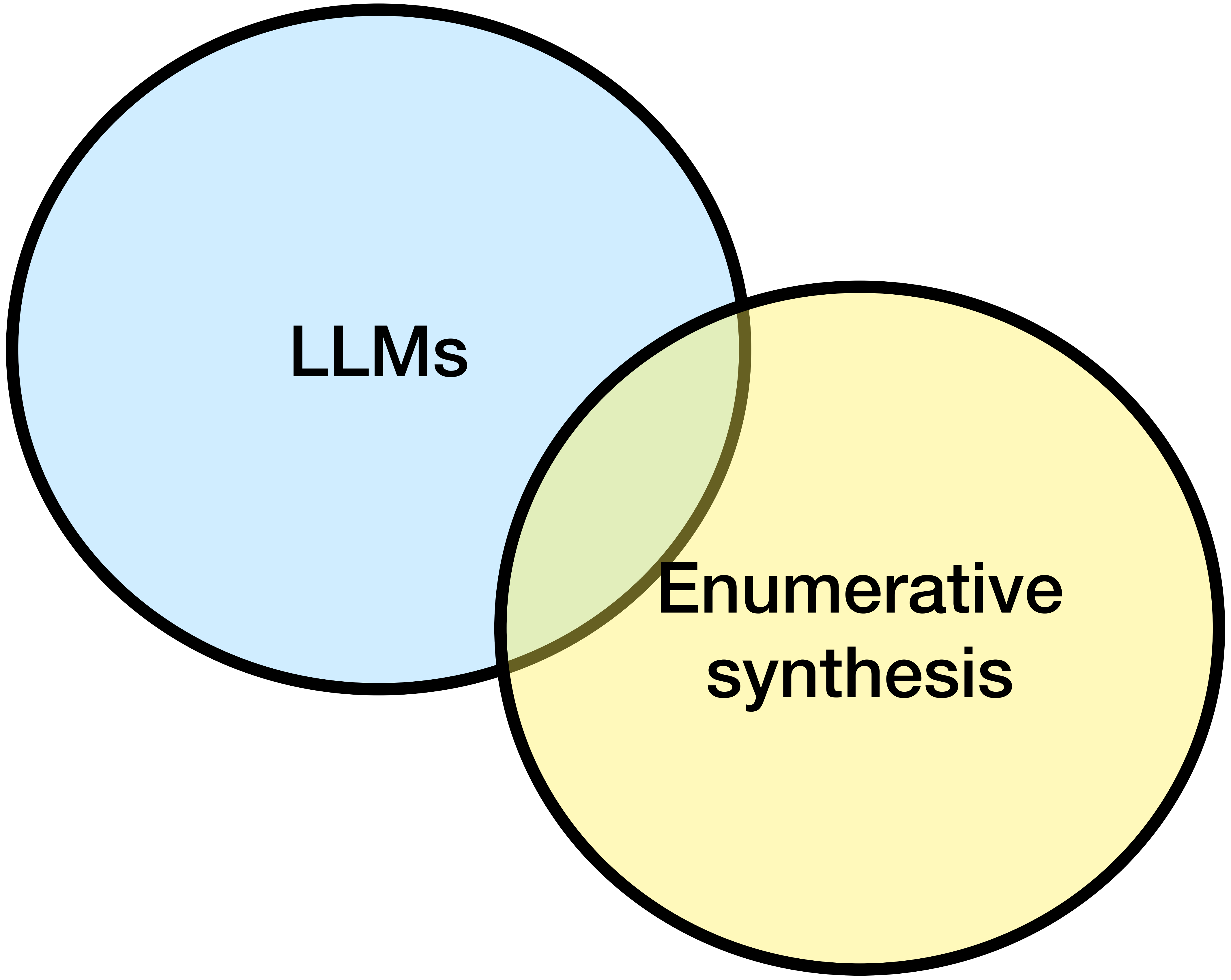
These are the results after some effort prompt engineering. Initial results were much worse.

Single function benchmarks from the SyGuS competition, with function names removed and the full grammar from the logic permitted and a timeout of 180s

Can LLMs solve formal synthesis problems?

Sort of.. but we think we can do better

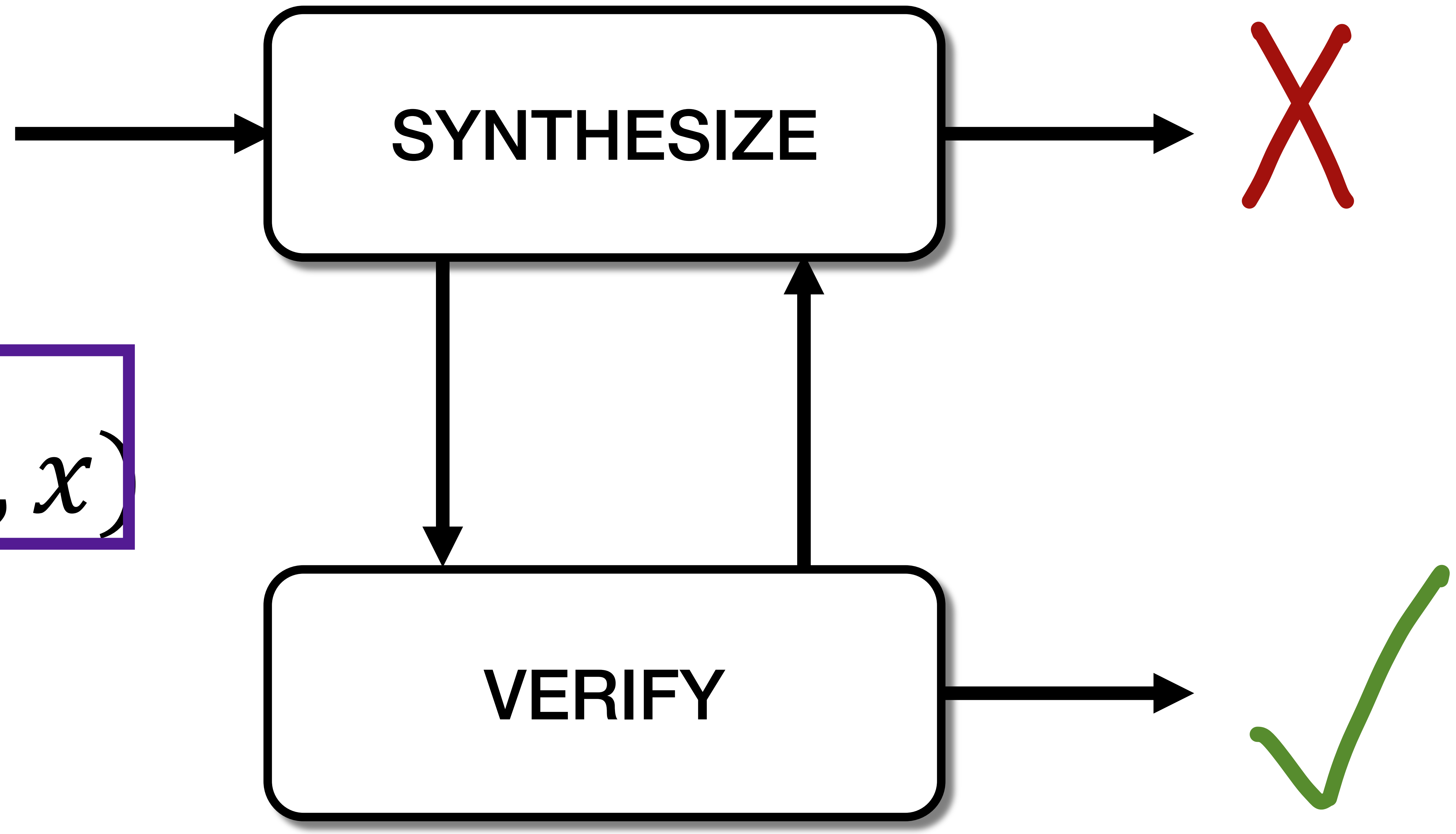

LLMs
Enumerative synthesis


LLMs
Enumerative
synthesis

Algorithms for formal synthesis

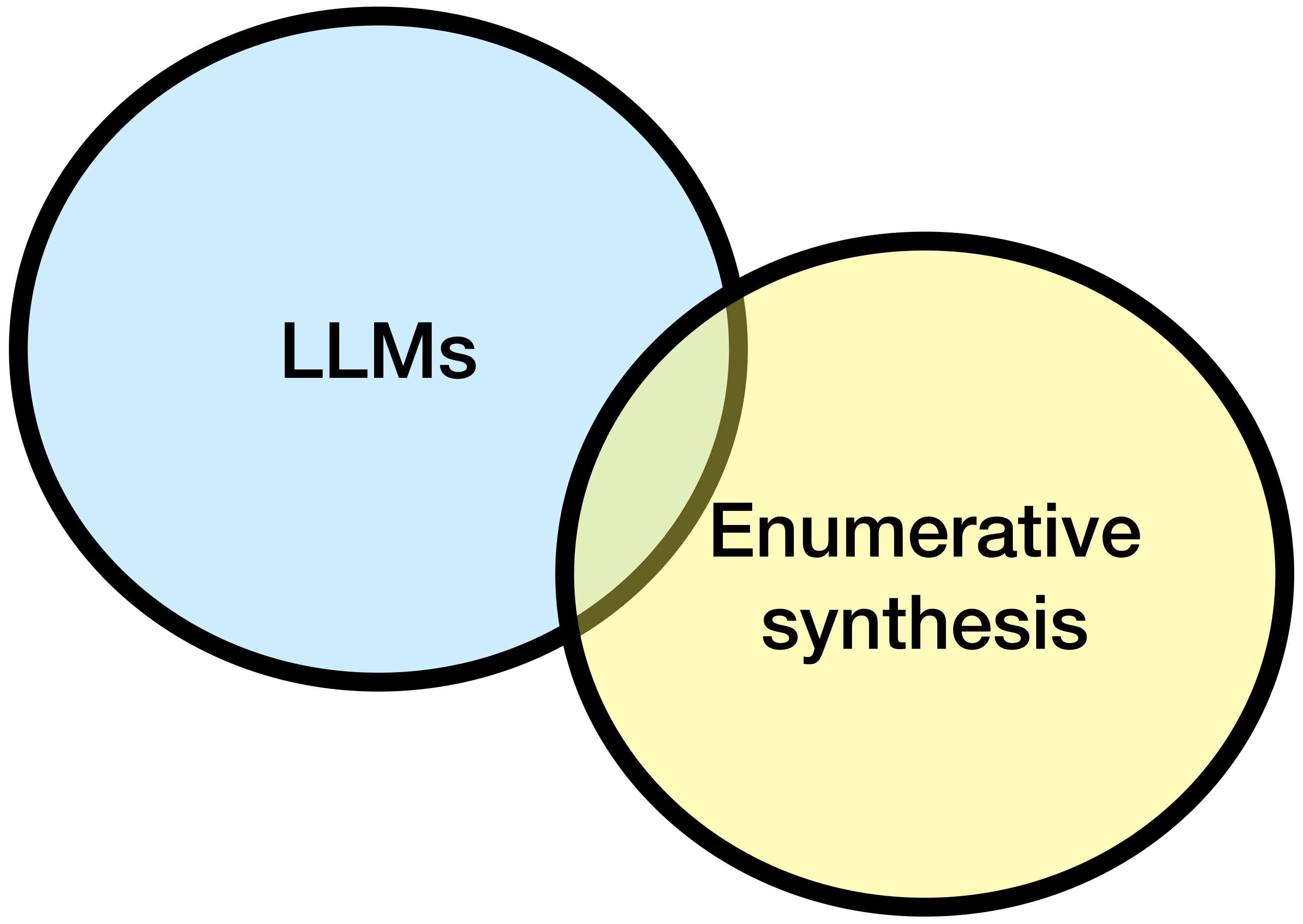
Counterexample Guided Inductive Synthesis

$$\boxed{\exists P \forall x.\, \sigma(P, x)}$$



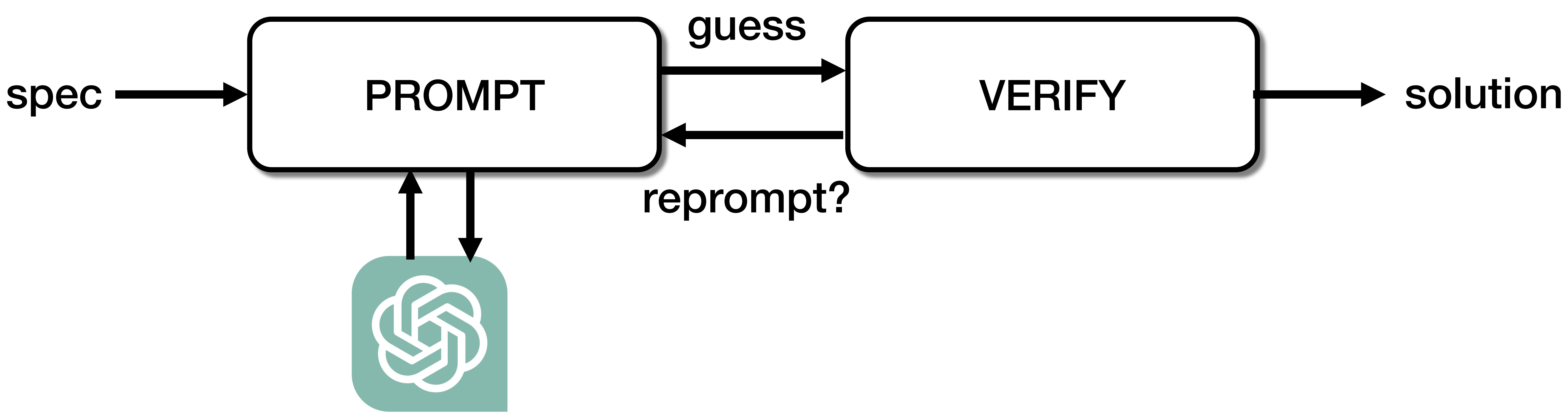
Combinatorial sketching for finite programs - Solar Lezama et al

Three approaches:



1. Prompt and verify
2. Use the LLM as pre-trained syntactic guidance
3. Use the LLM as an integrated syntactic oracle

Approach 1: prompt and verify



The right prompts?

- This is not a prompt engineering paper!
- We took some prompt engineering techniques from the literature, and tested them on a small sample of benchmarks.
- The space is huge, a much bigger search could be done.

The right prompts?

Role[1]

```
(set-logic LIA)
(declare-var vr0 Int)
(declare-var vr1 Int)
(declare-var vr2 Int)
(constraint (>= (fn vr0 vr1 vr2) vr0))
(constraint (>= (fn vr0 vr1 vr2) vr1))
(constraint (>= (fn vr0 vr1 vr2) vr2))
(constraint (or (= vr0 (fn vr0 vr1 vr2)) (or (= vr1 (fn vr0 vr1 vr2)) (=
     vr2 (fn vr0 vr1 vr2)))))
(check-synth)
You are a good synthesizer. Do you know what "(define-fun fn ((vr0 Int) (
     vr1 Int) (vr2 Int)) Int" is doing?
Write only one Lisp-like method "defun fn" without any built-in methods or
      arrays.
Requirements:
1. No built-in functions.
2. Never violate the SMT-LIB constraints above.
3. Pay attention to the define functions.
4. Ensure the response contains one and only one function.
5. Do not include any iterations, BitVec, or Int notations in the function
      body.
Write it correctly, or I will lose my job and 100 grandmothers will die.
     Don't disappoint me.
Write only one Lisp-like method "defun fn" that never violates the SMT-LIB
      constraints above.
```

[1] Better Zero-Shot Reasoning with Role-Play Prompting – Kong et al

The right prompts?

Role[1]

```
(set-logic LIA)
(declare-var vr0 Int)
(declare-var vr1 Int)
(declare-var vr2 Int)
(constraint (>= (fn vr0 vr1 vr2) vr0))
(constraint (>= (fn vr0 vr1 vr2) vr1))
(constraint (>= (fn vr0 vr1 vr2) vr2))
(constraint (or (= vr0 (fn vr0 vr1 vr2)) (or (= vr1 (fn vr0 vr1 vr2)) (=
     vr2 (fn vr0 vr1 vr2)))))
(check-synth)
You are a good synthesizer. Do you know what "(define-fun fn ((vr0 Int) (
     vr1 Int) (vr2 Int)) Int" is doing?
Write only one Lisp-like method "defun fn" without any built-in methods or
      arrays.
Requirements:
1. No built-in functions.
2. Never violate the SMT-LIB constraints above.
3. Pay attention to the define functions.
4. Ensure the response contains one and only one function.
5. Do not include any iterations, BitVec, or Int notations in the function
      body.
Write it correctly, or I will lose my job and 100 grandmothers will die.
     Don't disappoint me.
Write only one Lisp-like method "defun fn" that never violates the SMT-LIB
      constraints above.
```

Ask in Lisp[2]

[2] "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models – Liu et al

The right prompts?

```
(set-logic LIA)
(declare-var vr0 Int)
(declare-var vr1 Int)
(declare-var vr2 Int)
(constraint (>= (fn vr0 vr1 vr2) vr0))
(constraint (>= (fn vr0 vr1 vr2) vr1))
(constraint (>= (fn vr0 vr1 vr2) vr2))
(constraint (or (= vr0 (fn vr0 vr1 vr2)) (or (= vr1 (fn vr0 vr1 vr2)) (=
     vr2 (fn vr0 vr1 vr2)))))
(check-synth)
You are a good synthesizer. Do you know what "(define-fun fn ((vr0 Int) (
     vr1 Int) (vr2 Int)) Int" is doing?
Write only one Lisp-like method "defun fn" without any built-in methods or
      arrays.
Requirements:
1. No built-in functions.
2. Never violate the SMT-LIB constraints above.
3. Pay attention to the define functions.
4. Ensure the response contains one and only one function.
5. Do not include any iterations, BitVec, or Int notations in the function
      body.
Write it correctly, or I will lose my job and 100 grandmothers will die.
     Don't disappoint me.
Write only one Lisp-like method "defun fn" that never violates the SMT-LIB
      constraints above.
```

Role[1]

Ask in Lisp[2]

Emotional stimuli [3]

[3] Li, C., Wang, J., Zhang, Y., Zhu, K., Hou, W., Lian, J., Luo, F., Yang, Q., Xie, X.: Large language models understand and can be enhanced by emotional stimuli.

The right prompts?

```
(synth-inv inv-f ((x Int) (y Int)))
(define-fun pre-f ((x Int) (y Int)) Bool (and (= x 1) (= y 1)))
(define-fun trans-f ((x Int) (y Int) (x! Int) (y! Int)) Bool (and (= x! (+
      x y)) (= y! (+ x y))))
(define-fun post-f ((x Int) (y Int)) Bool (>= y 1))
(inv-constraint inv-f pre-f trans-f post-f)
Please explain the constraints above.
```

Ask for LLM explanation for invariant constraints [4]

[4] Chain-of-Thought Prompting Elicits Reasoning in Large Language Models – Wei et al

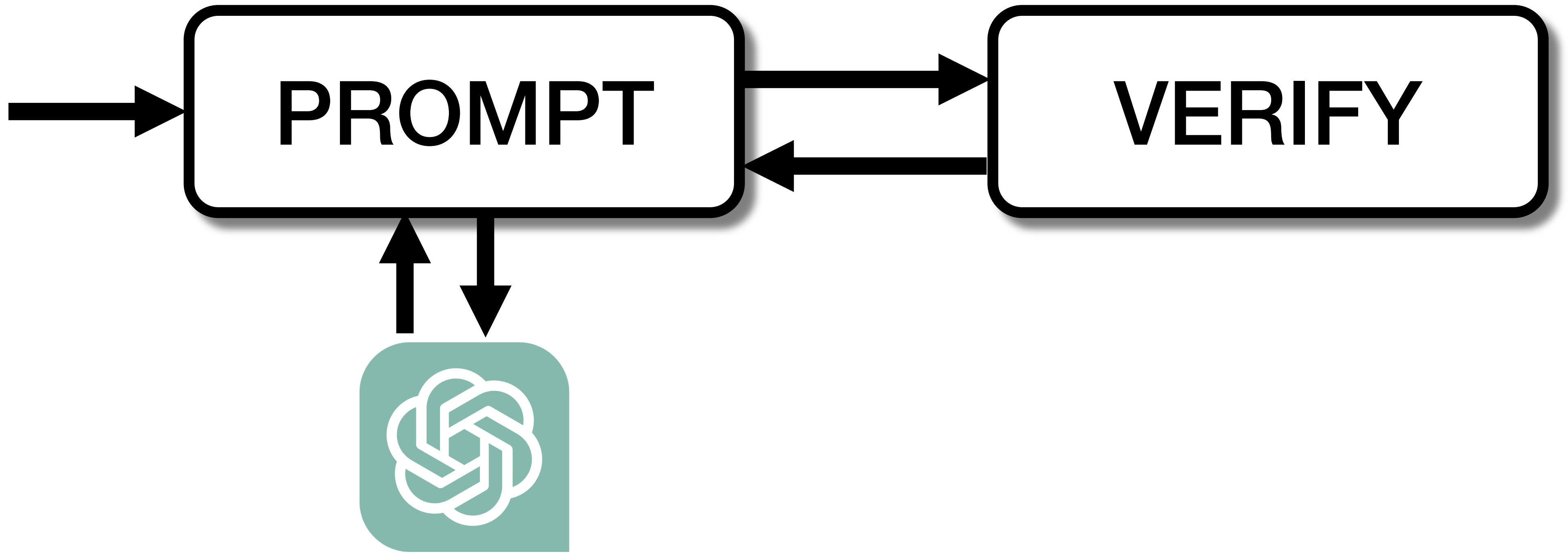
The right prompts?

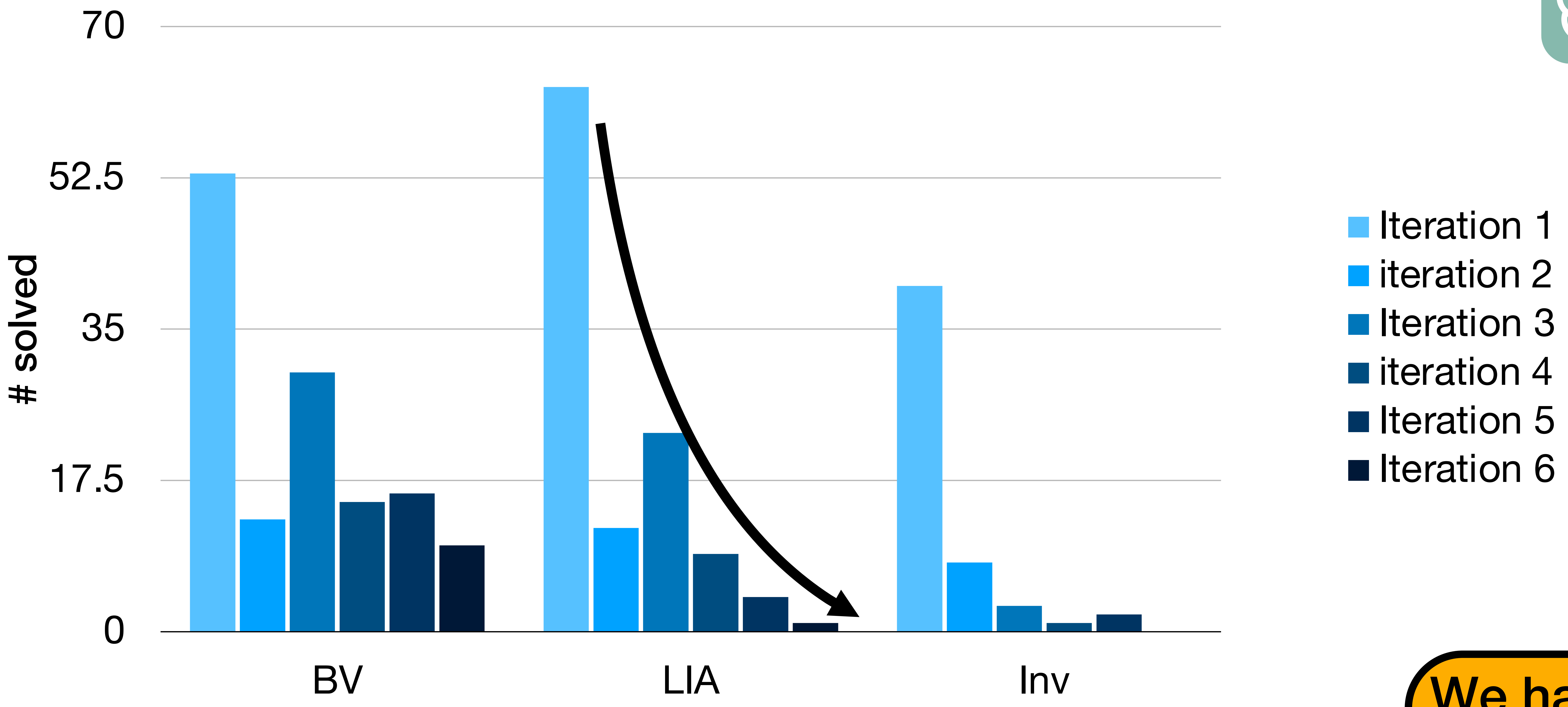
```
(set-logic LIA)
(declare-var vr0 Int)
(declare-var vr1 Int)
(declare-var vr2 Int)
(constraint (>= (fn vr0 vr1 vr2) vr0))
(constraint (>= (fn vr0 vr1 vr2) vr1))
(constraint (>= (fn vr0 vr1 vr2) vr2))
(constraint (or (= vr0 (fn vr0 vr1 vr2)) (or (= vr1 (fn vr0 vr1 vr2)) (=
     vr2 (fn vr0 vr1 vr2)))))
(check-synth)
You are a good synthesizer. Do you know what "(define-fun fn ((vr0 Int) (
     vr1 Int) (vr2 Int)) Int" is doing?
Write only one Lisp-like method "defun fn" without any built-in methods or
      arrays.
Requirements:
1. No built-in functions.
2. Never violate the SMT-LIB constraints above.
3. Pay attention to the define functions.
4. Ensure the response contains one and only one function.
5. Do not include any iterations, BitVec, or Int notations in the function
      body.
Write it correctly, or I will lose my job and 100 grandmothers will die.
     Don't disappoint me.
Write only one Lisp-like method "defun fn" that never violates the SMT-LIB
      constraints above.
```

Retry if the solution is incorrect

```
You are close to the right answer. Take another guess. You have to try
     something different, think harder. Write a different Lisp method that
      never violates the SMT-LIB constraints above again.
```

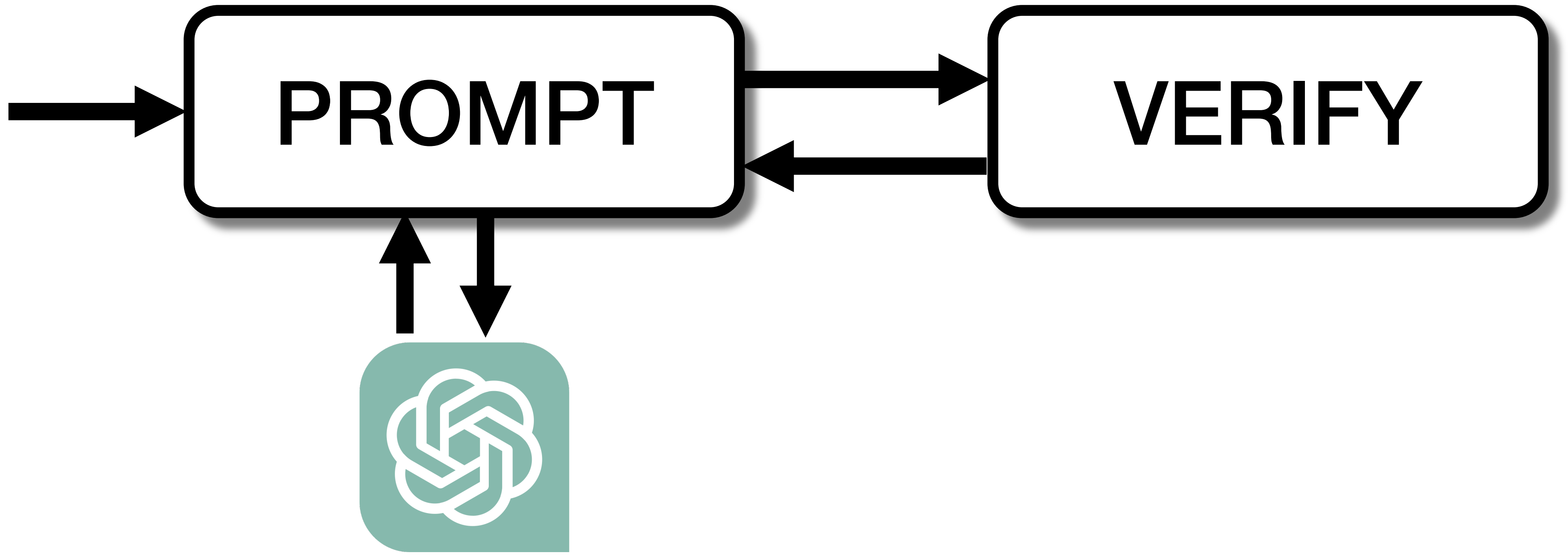
Number of iterations

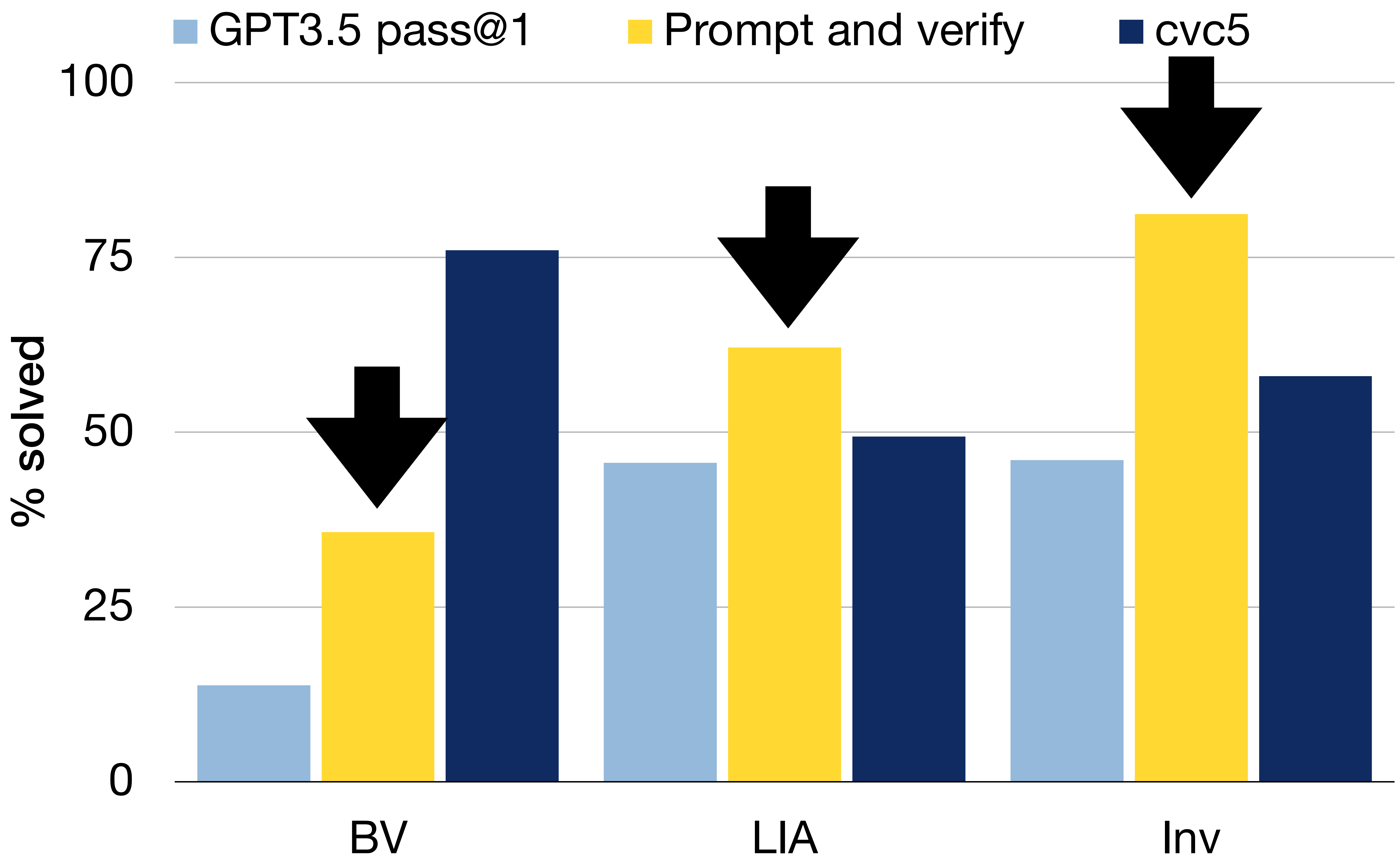




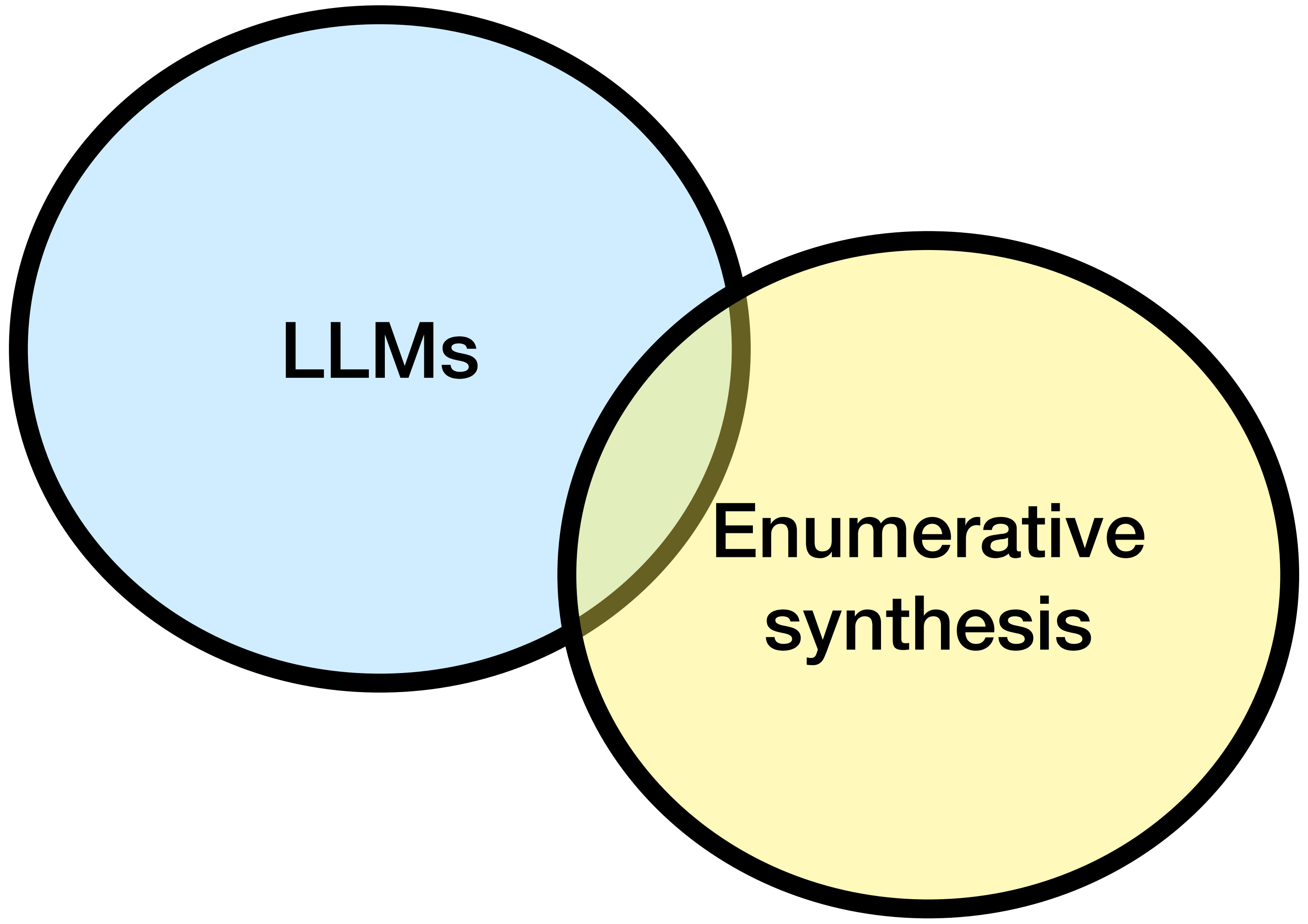
We halt after 6 iterations, as number of new solutions tails off significantly

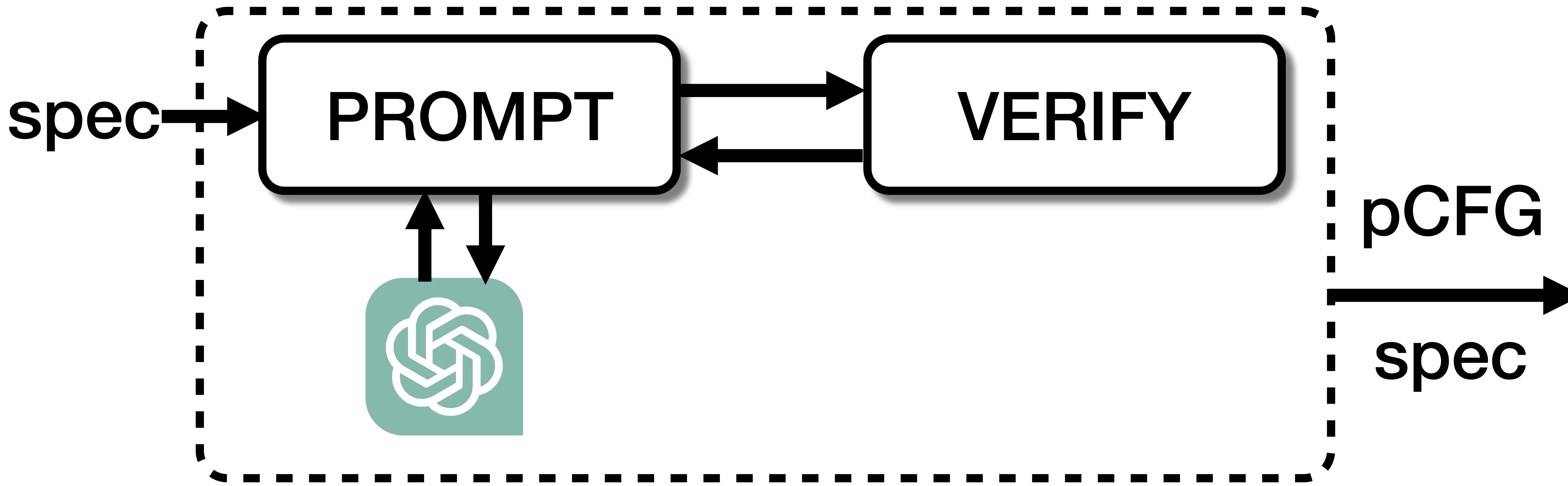
Results: prompt and verify

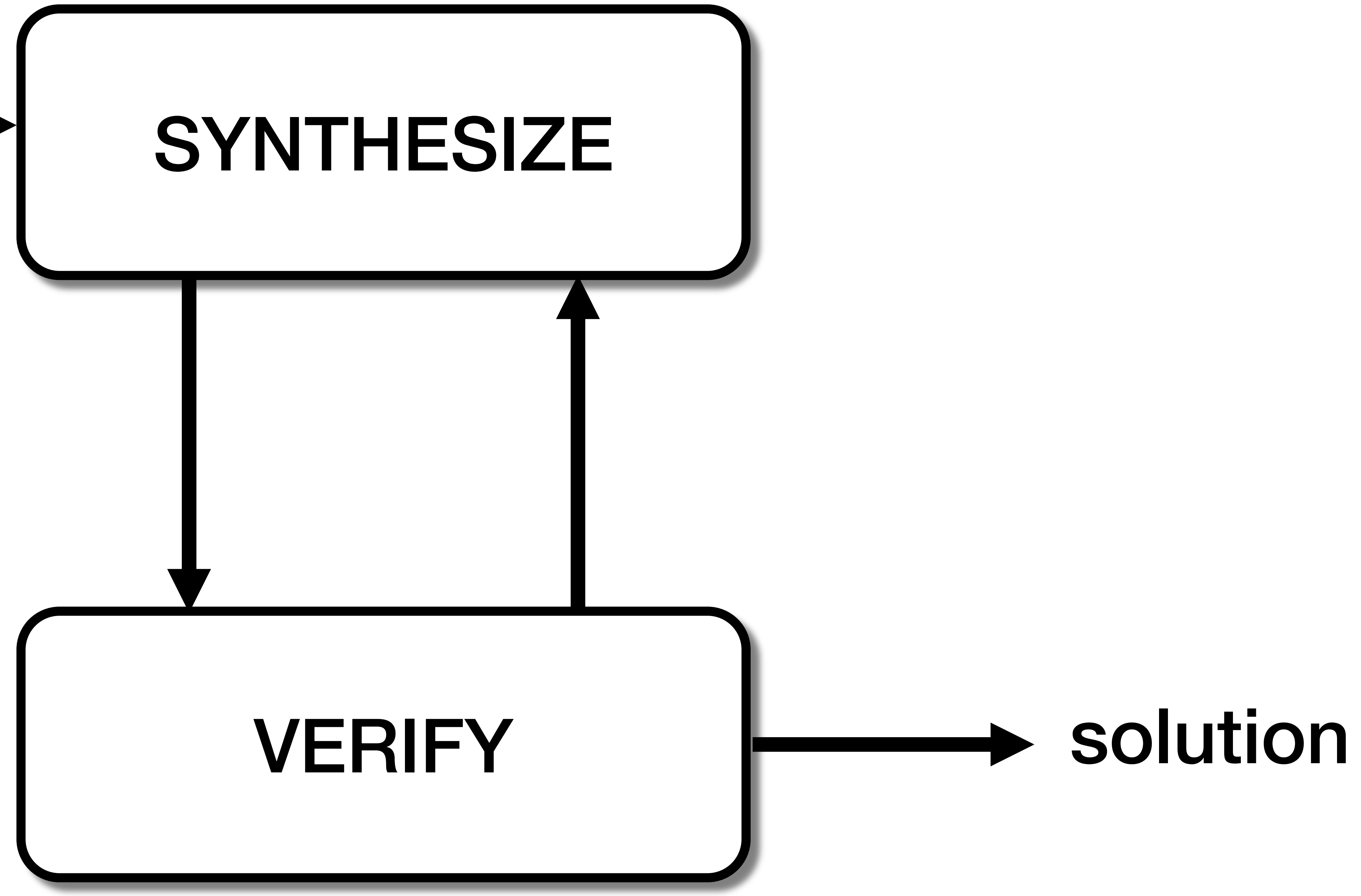




Approach 2: Pre-trained guidance







Use the LLM to generate a pre-trained model

“Pre-trained” guidance from LLMs

If the LLM guesses are wrong, perhaps the solution is still in the neighborhood of those guesses?

Model that neighborhood as a probabilistic grammar.

```
S → 0 | 1 | 2
S → y | x
S → B ? S:S
B → S = S
B → S ≥ S
B → S ≤ S
B → !B
B → B ∨ B
B → B ∧ B
```

```
y   x
(x ≥ 0)? x : y
(x ≥ y)? x : 0
(y ≤ x)? x : y
(y ≥ x)? x : y
```

Parser

```
S → 0 (1)| 1(1) | 2(1)
S → y (1)| x (1)
S → B ? S:S (1)
B → S = S (1)
B → S ≥ S (1)
B → S ≤ S (1)
B → !B (1)
B → B ∨ B (1)
B → B ∧ B (1)
```

```
S → 0 | 1 | 2
S → y | x
S → B ? S:S
B → S = S
B → S ≥ S
B → S ≤ S
B → !B
B → B ∨ B
B → B ∧ B
```

```
y x

(x ≥ 0)? x : y

(x ≥ y)? x : 0

(y ≤ x)? x : y

(y ≥ x)? x : y
```

Parser

```
S → 0 (3)| 1(1) | 2(1)
S → y (7)| x (10)
S → B ? S:S (5)
B → S = S (1)
B → S ≥ S (4)
B → S ≤ S (2)
B → !B (1)
B → B ∨ B (1)
B → B ∧ B (1)
```

$S \rightarrow 0 \mid 1 \mid 2$
$S \rightarrow y \mid x$
$S \rightarrow B\ ?\ S:S$
$B \rightarrow S = S$
$B \rightarrow S \geq S$
$B \rightarrow S \leq S$
$B \rightarrow\ !B$
$B \rightarrow B \vee B$
$B \rightarrow B \wedge B$

$y \quad x$

$(x \geq 0)\ ?\ x : y$

$(x \geq y)\ ?\ x : 0$

$(y \leq x)\ ?\ x : y$

$(y \geq x)\ ?\ x : y$

Parser

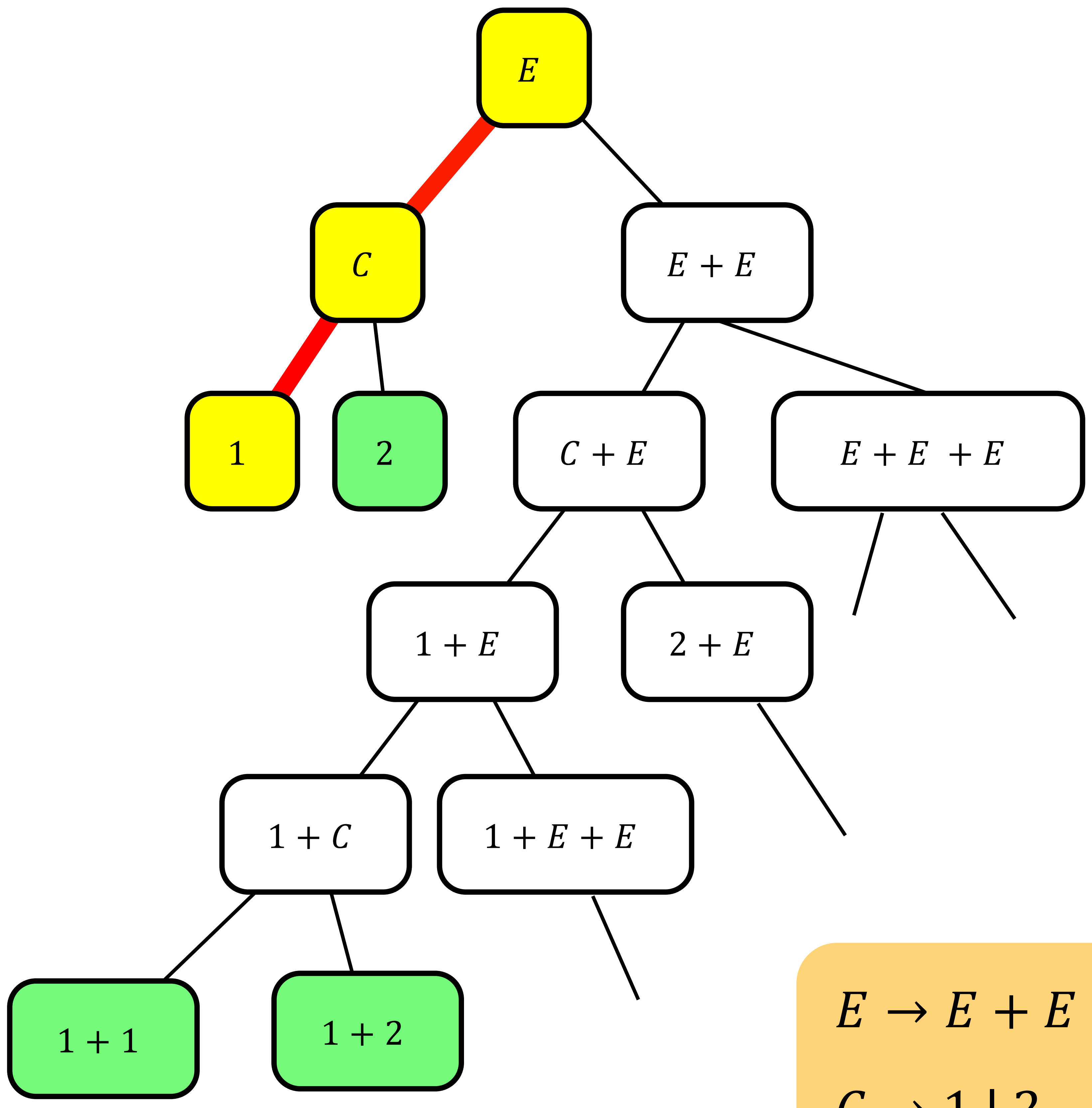
$S \rightarrow 0\ (0.11) \mid 1\ (0.04) \mid 2\ (0.04)$
$S \rightarrow y\ (0.26) \mid x\ (0.37)$
$S \rightarrow B\ ?\ S:S\ (0.19)$
$B \rightarrow S = S\ (0.1)$
$B \rightarrow S \geq S\ (0.4)$
$B \rightarrow S \leq S\ (0.2)$
$B \rightarrow\ !B\ (0.1)$
$B \rightarrow B \vee B\ (0.1)$
$B \rightarrow B \wedge B\ (0.1)$

probabilistic Context-Free Grammar

$P_G = (V, \Sigma, R, S, \mathbb{P})$:

- V is a set of nonterminal symbols
- Σ is a set of terminal symbols
- $R \subseteq V \times (V \cup \Sigma)^*$ is a set of production rules
- S is a start symbol
- $\mathbb{P}$ is a probability mass function that assigns a probability $\mathbb{P}[r]$ to each $r \in R$

Top down enumeration

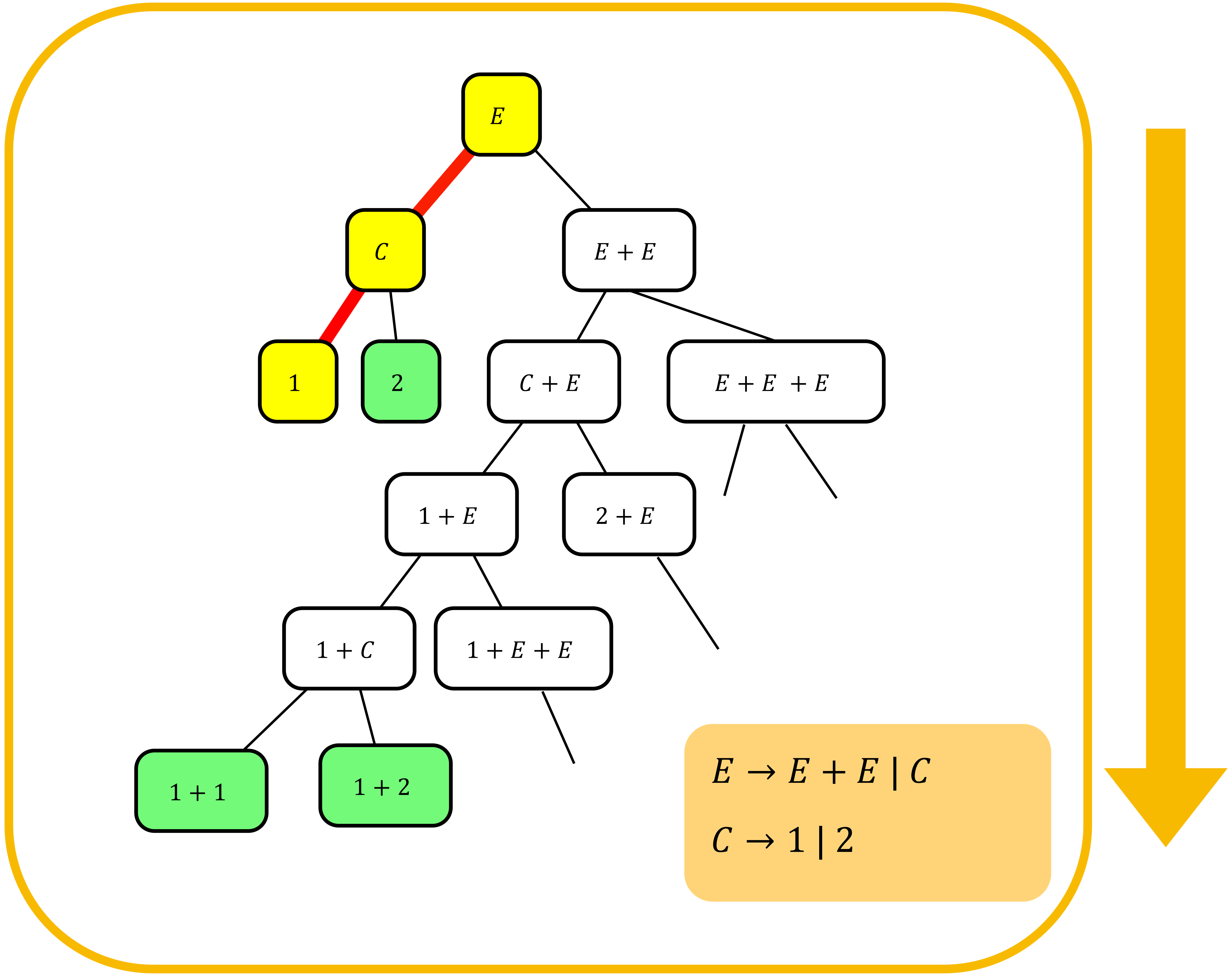


$E \rightarrow E + E \mid C$

$C \rightarrow 1 \mid 2$

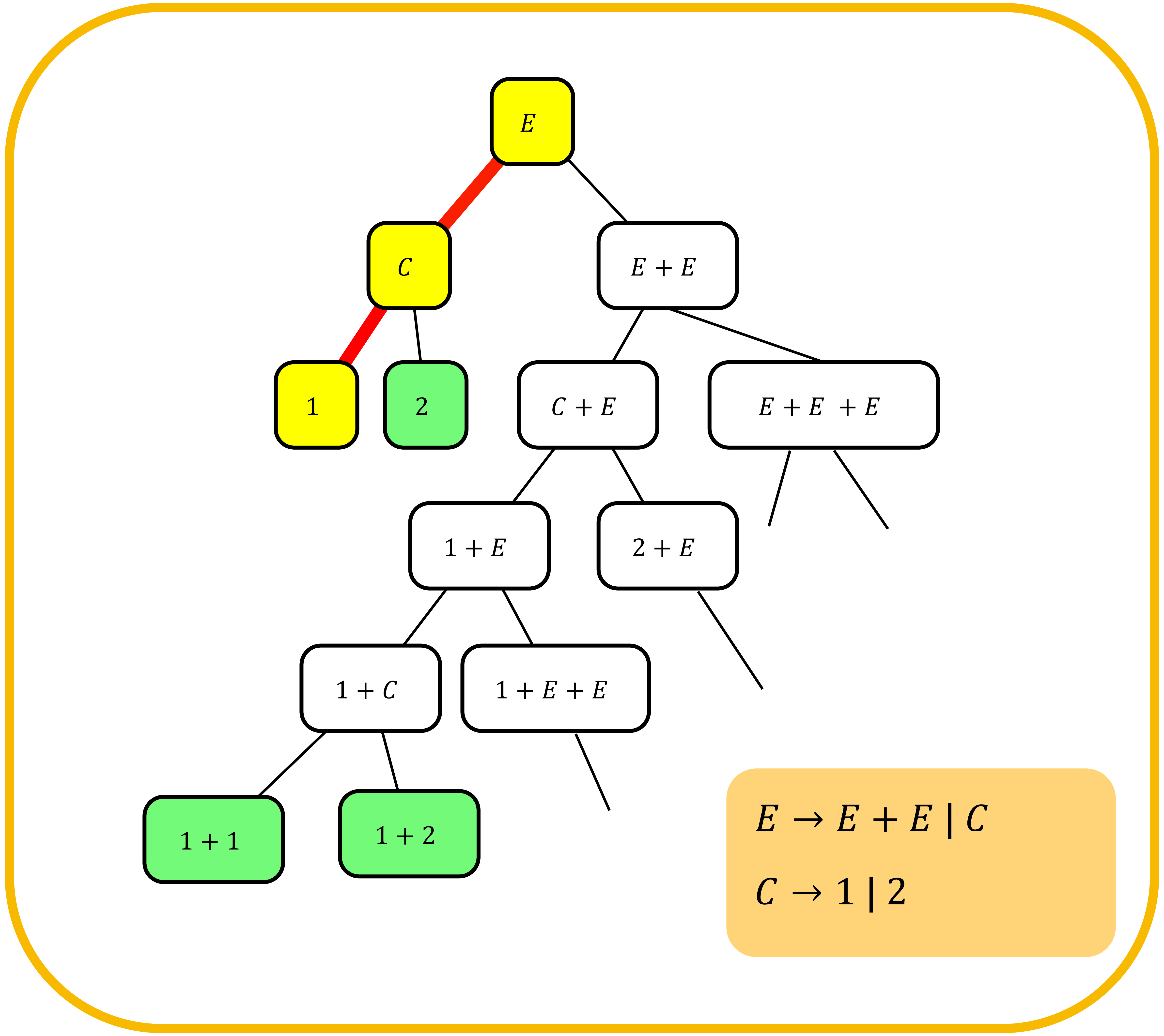
- Initialize expression with the Start symbol
- Repeatedly choose production rules to replace the left-most non-terminal in the current expression
- Choice is made by sampling from distribution over possible production rules
- Repeat until depth limit is hit or complete program is found

Top down enumeration



- Initialize expression with the Start symbol
- Repeatedly choose production rules to replace the left-most non-terminal in the current expression
- Choice is made by sampling from distribution over possible production rules
- Repeat until depth limit is hit or complete program is found
- If depth limit: complete the program

Top down enumeration



- Initialize expression with the Start symbol
- Repeatedly choose production rules to replace the left-most non-terminal in the current expression
- Choice is made by sampling from distribution over possible production rules
- Repeat until depth limit is hit or complete program is found
- If complete, return for verification

A* enumeration

E

Score = cost so far + estimate of cost to complete program

$E \rightarrow E + E \mid C$

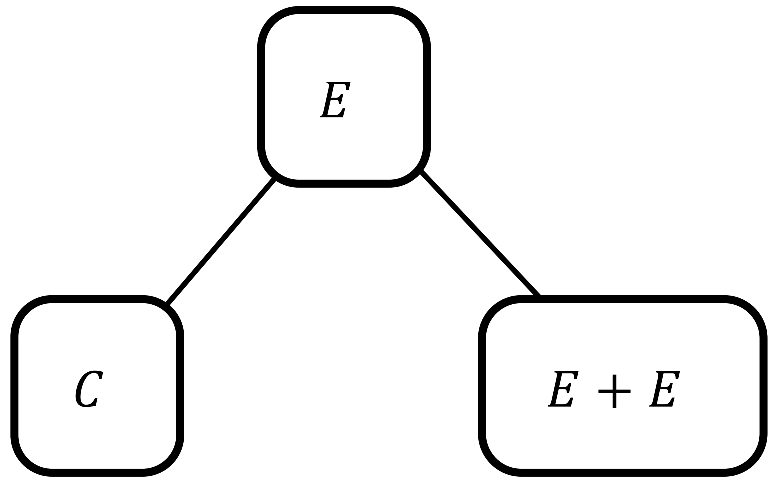
$C \rightarrow 1 \mid 2$

- Queue stores partial programs with scores
- Initialize queue with the Start symbol

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration

Score = cost so far + estimate of cost to complete program



$E \rightarrow E + E \mid C$

$C \rightarrow 1 \mid 2$

- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration

C $E + E$

Score = cost so far + estimate of cost to complete program

$E \to E + E \mid C$

$C \to 1 \mid 2$

- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration

C $E + E$

Score = cost so far + estimate of cost to complete program

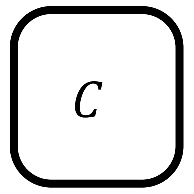
$E \rightarrow E + E \mid C$

$C \rightarrow 1 \mid 2$

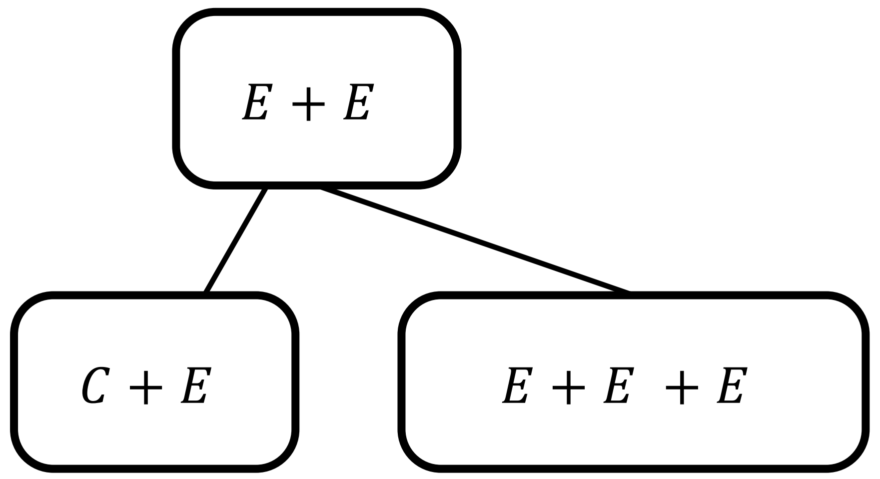
- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand
- Repeat until complete program found

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration



Score = cost so far + estimate of cost to complete program



$E \to E + E \mid C$

$C \to 1 \mid 2$

- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand
- Repeat until complete program found

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration

C | $C+E$ | $E+E$ | $E+E+E$

Score = cost so far + estimate of cost to complete program

$$E \rightarrow E + E \mid C$$

$$C \rightarrow 1 \mid 2$$

- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand
- Repeat until complete program found

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration

C | $C + E$ | $E + E$ | $E + E + E$

Score = cost so far + estimate of cost to complete program

$E \rightarrow E + E \mid C$

$C \rightarrow 1 \mid 2$

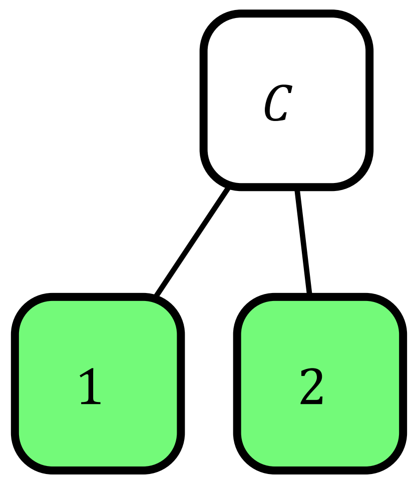
- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand
- Repeat until complete program found

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration

$C + E$ | $E + E$ | $E + E + E$

Score = cost so far + estimate of cost to complete program



$E \rightarrow E + E \mid C$

$C \rightarrow 1 \mid 2$

- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand
- Repeat until complete program found

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration

$C + E$ | $E + E$ | $E + E + E$ | 1 | 2

Score = cost so far + estimate of cost to complete program

$E \rightarrow E + E \mid C$

$C \rightarrow 1 \mid 2$

- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand
- Repeat until complete program found

[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

A* enumeration

$C + E$ | $E + E$ | $E + E + E$ | 1 | 2

Score = cost so far + estimate of cost to complete program

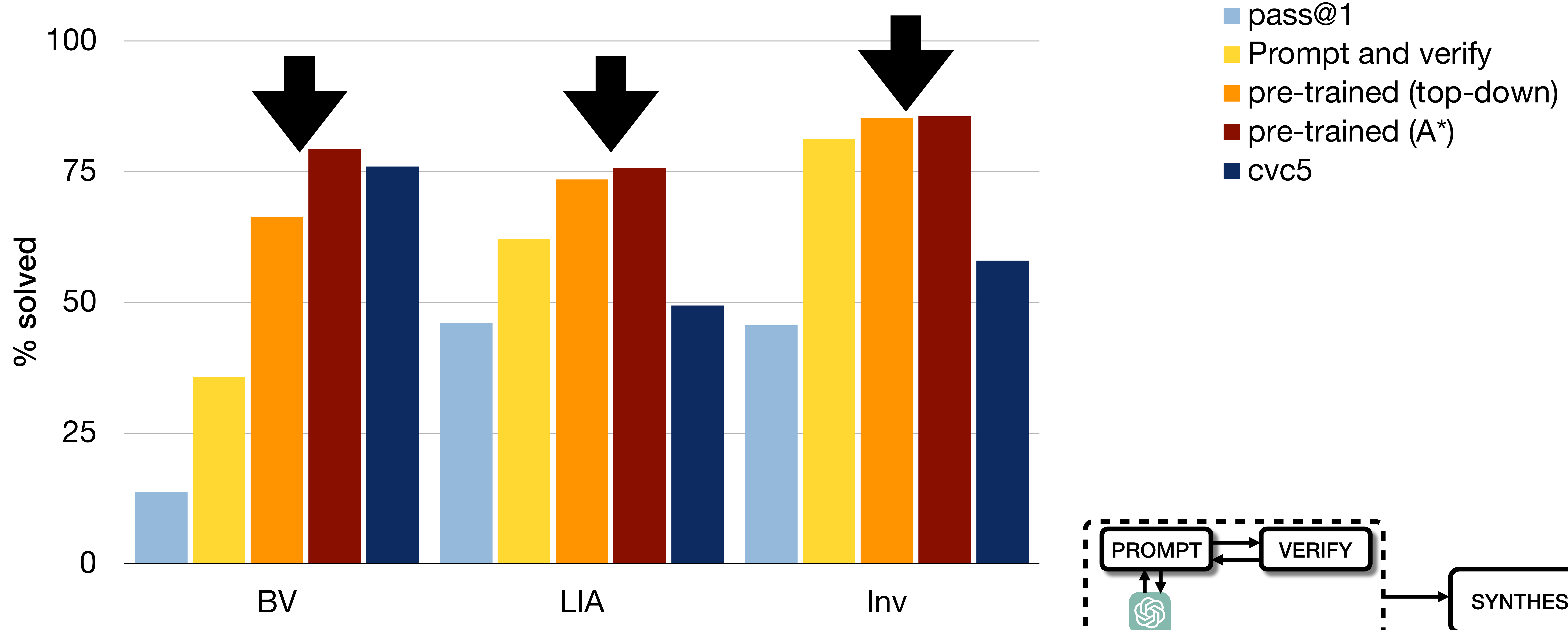
$E \rightarrow E + E \mid C$

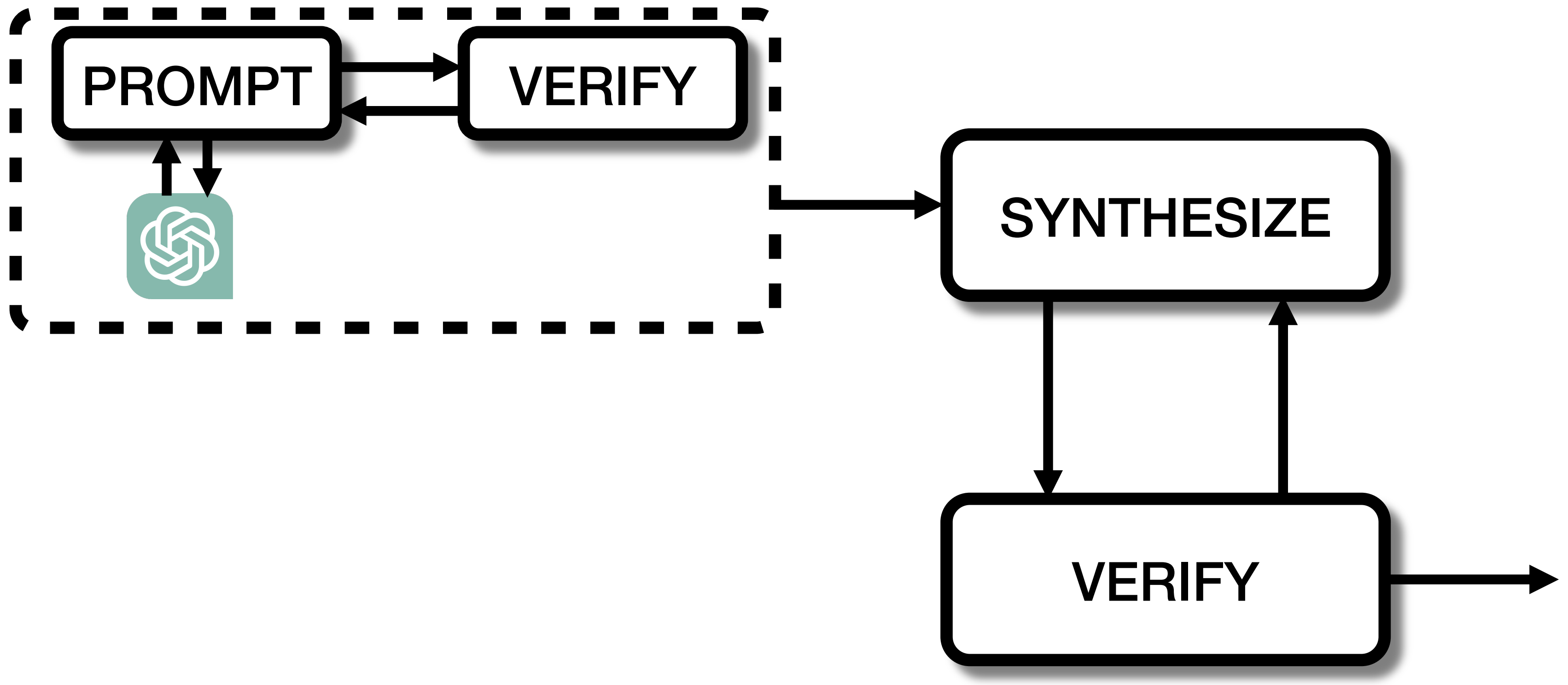
$C \rightarrow 1 \mid 2$

- Queue stores partial programs with scores
- Initialize queue with the Start symbol
- Pop partial program from queue with best score and expand
- Repeat until complete program found

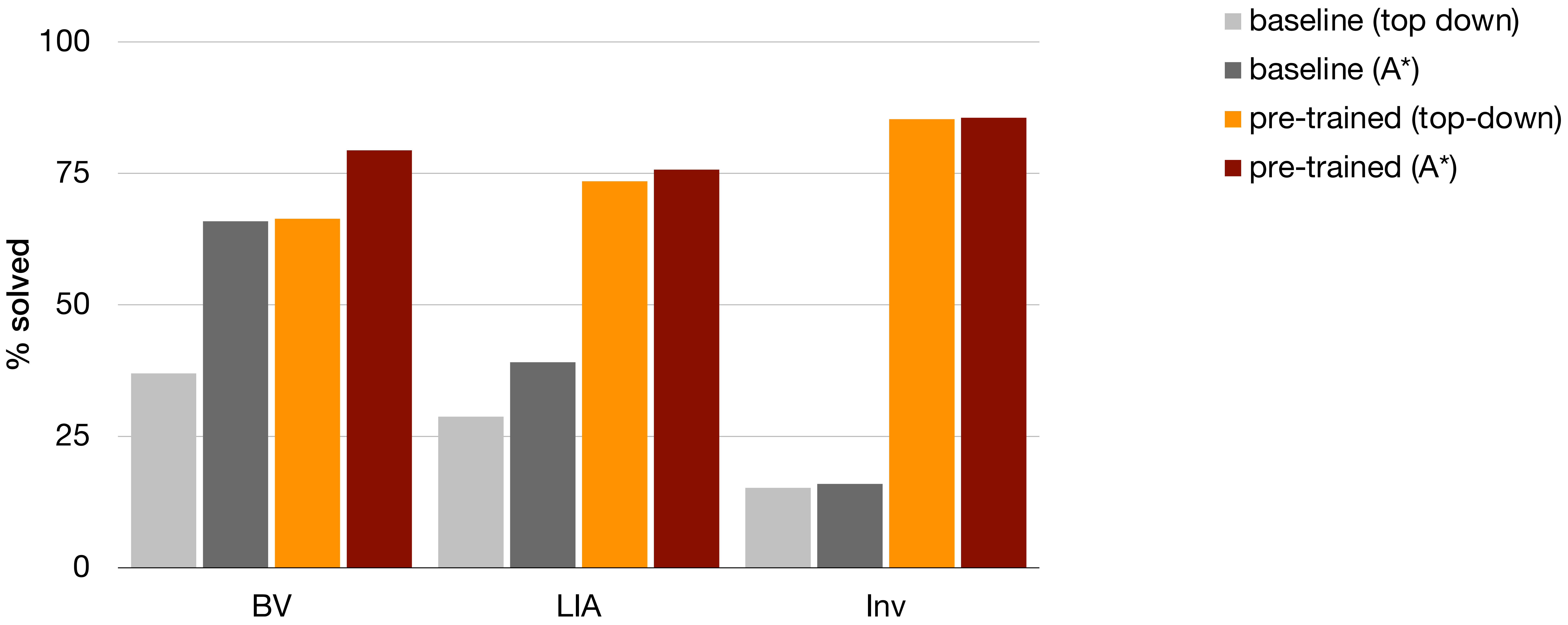
[1] Accelerating search-based program synthesis using learned probabilistic models - Lee et al

Results: pre-trained guidance

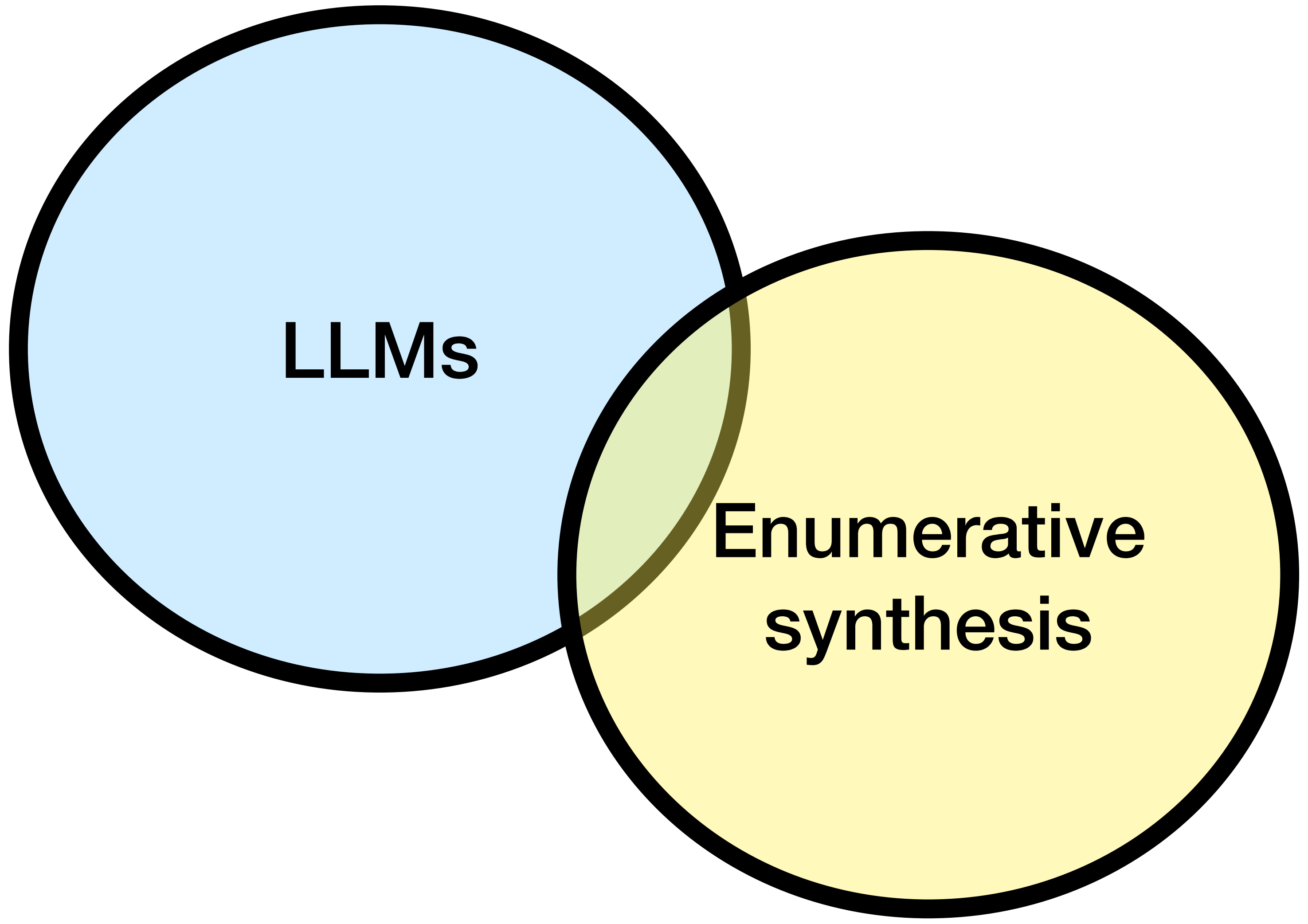


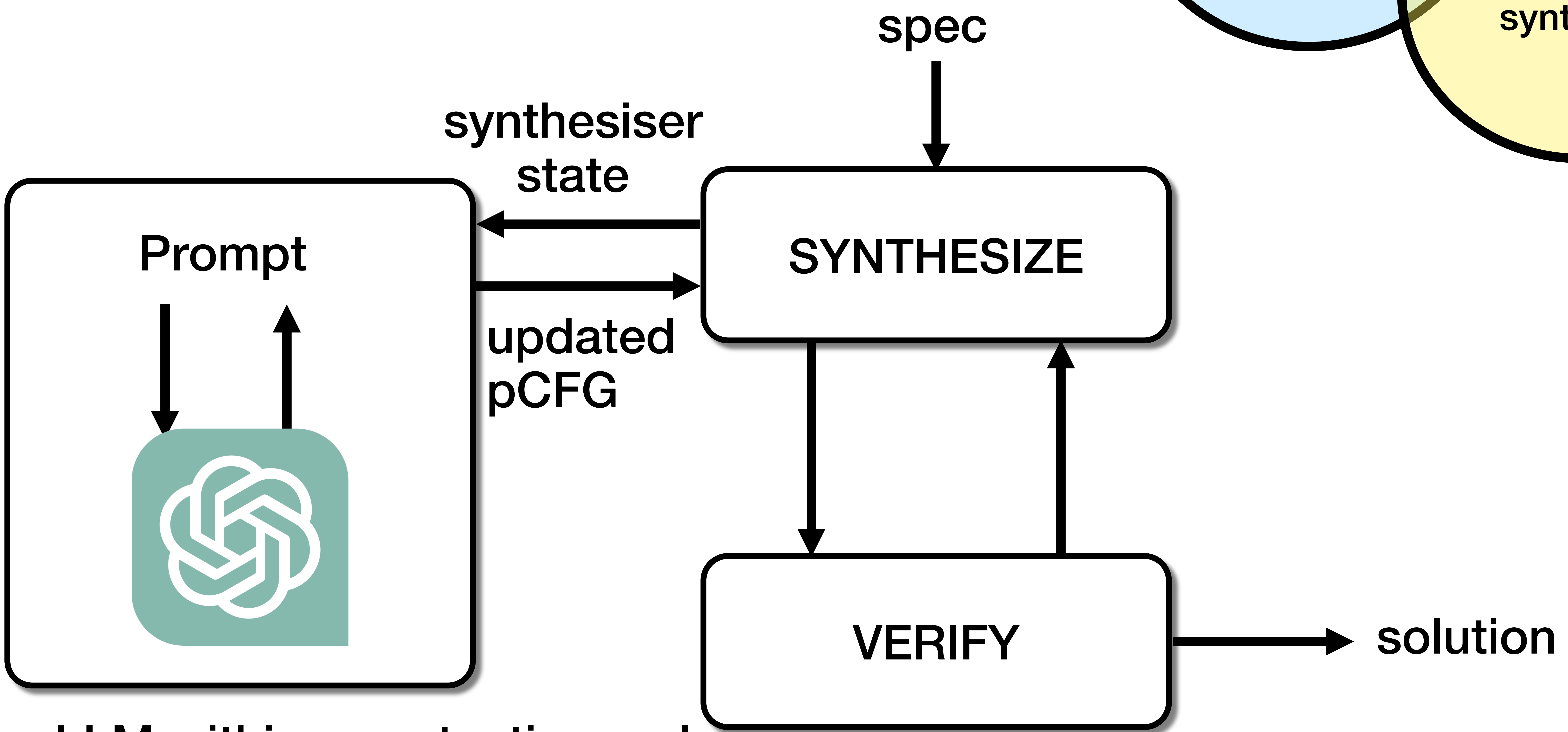


Comparison with unguided enumerators

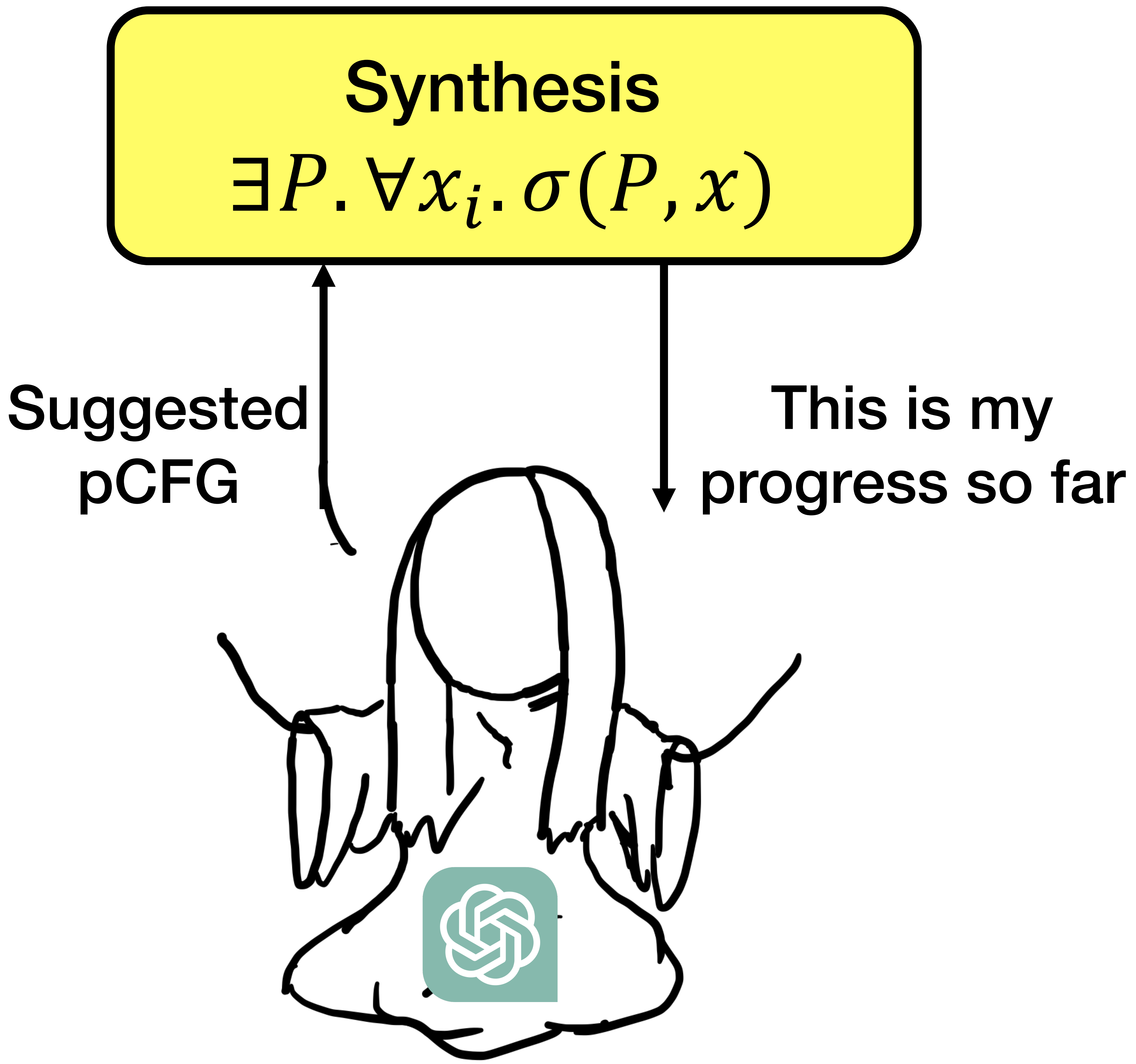


Approach 3: synthesis with a syntactic oracle





Use the LLM within a syntactic oracle

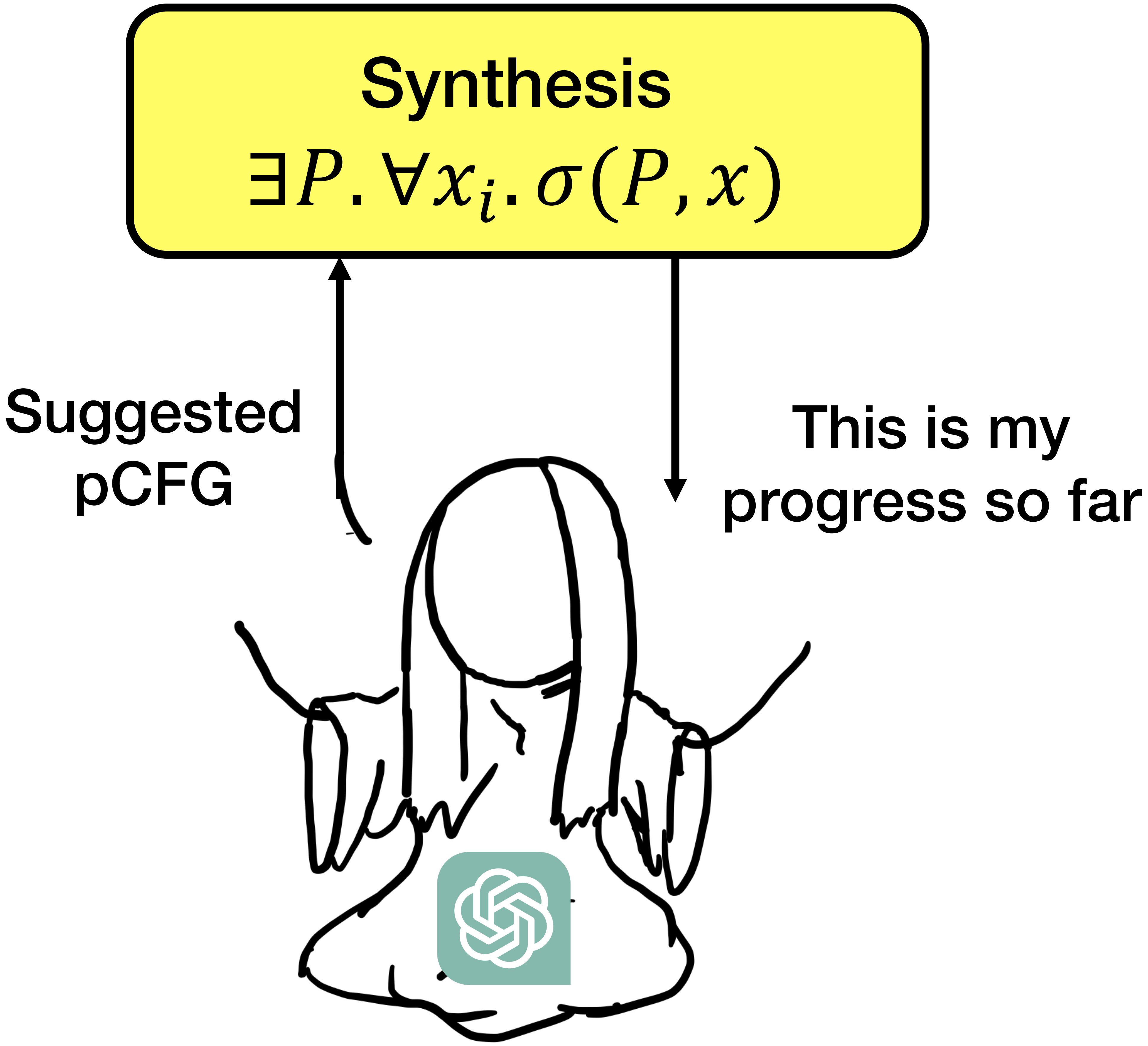


2 way communication:

- Send:
  - partially enumerated programs
  - counterexamples from previous iterations
  - incorrect solutions from previous iteration
- Return:
  - Helper functions
  - Which we turn into: updated pCFG with new production rules and updated distributions

```
You are teaching a student to write SMT-LIB. The student must write a
     function that satisfies the following constraints:
(constraint (>= (fn vr0 vr1 vr2) vr0))
(constraint (>= (fn vr0 vr1 vr2) vr1))
(constraint (>= (fn vr0 vr1 vr2) vr2))
(constraint (or (= vr0 (fn vr0 vr1 vr2)) (or (= vr1 (fn vr0 vr1 vr2)) (=
     vr2 (fn vr0 vr1 vr2)))))
So far, the student has written this code:
(define-fun fn ((vr0 Int) (vr1 Int) (vr2 Int)) Int
    (ite ?? ?? ??)
Can you suggest some helper functions for the student to use to complete
     this code and replace the ??
You must print only the code and nothing else.
```

Sure, here are some helper functions:



- Updating the pCFG:
  - parse all helper functions, and update the pCFG distributions as before
  - add any new helper functions as new production rules to any applicable non-terminal.

```
S → 0 | 1 | 2
S → y | x
S → B ? S:S
B → S = S
B → S ≥ S
B → S ≤ S
B → !B
B → B ∨ B
B → B ∧ B
```

```
(x ≥ 0)

(x ≥ y)
```

Parser

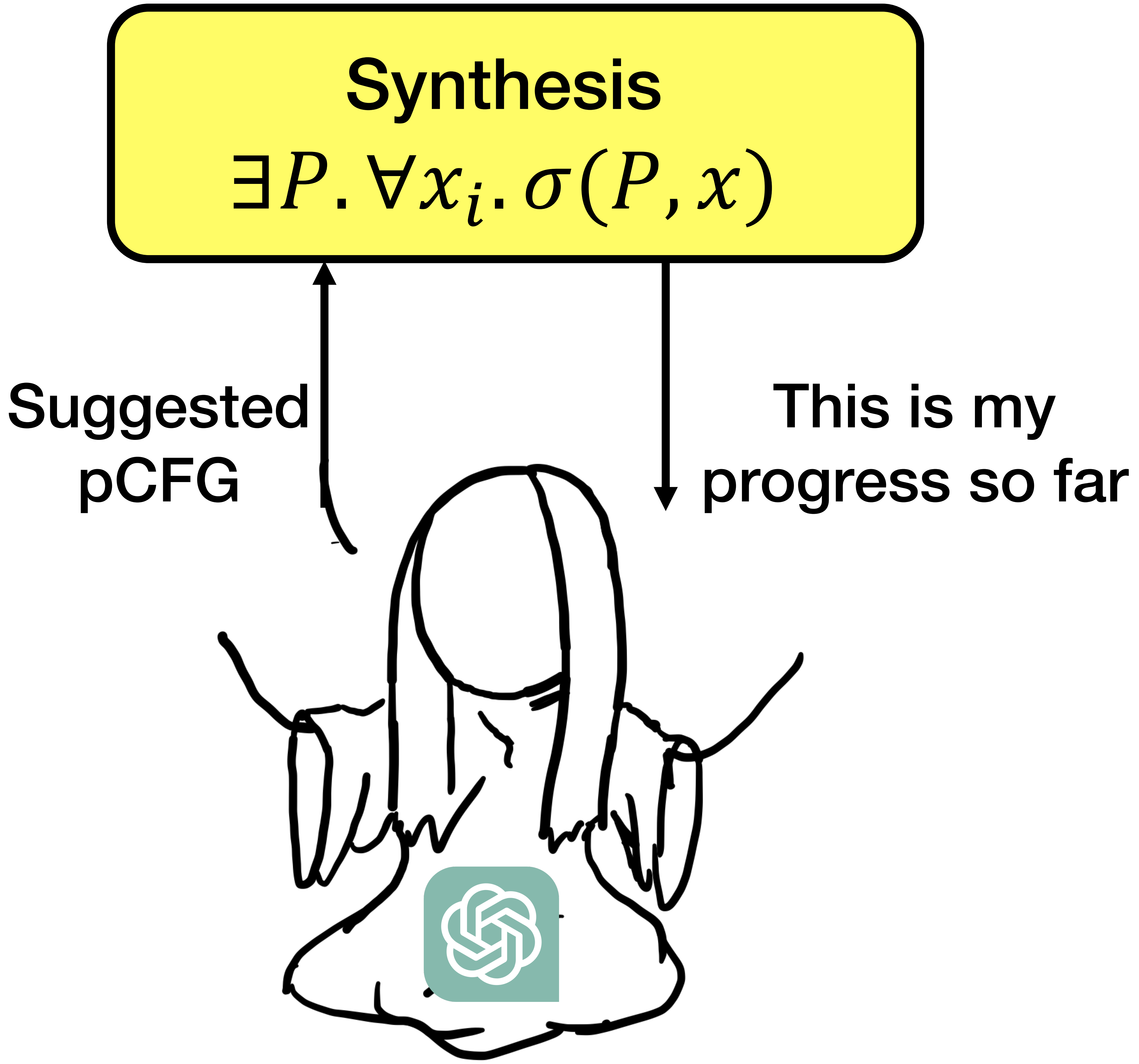
```
S → 0 (3)| 1(1) | 2(1)
S → y (7)| x (10)
S → B ? S:S (5)
B → S = S (1)
B → S ≥ S (4)
B → S ≤ S (2)
B → !B (1)
B → B ∨ B (1)
B → B ∧ B (1)
```

```
S → 0 | 1 | 2
S → y | x
S → B ? S:S
B → S = S
B → S ≥ S
B → S ≤ S
B → !B
B → B ∨ B
B → B ∧ B
```

```
(x ≥ 0)

(x ≥ y)
```

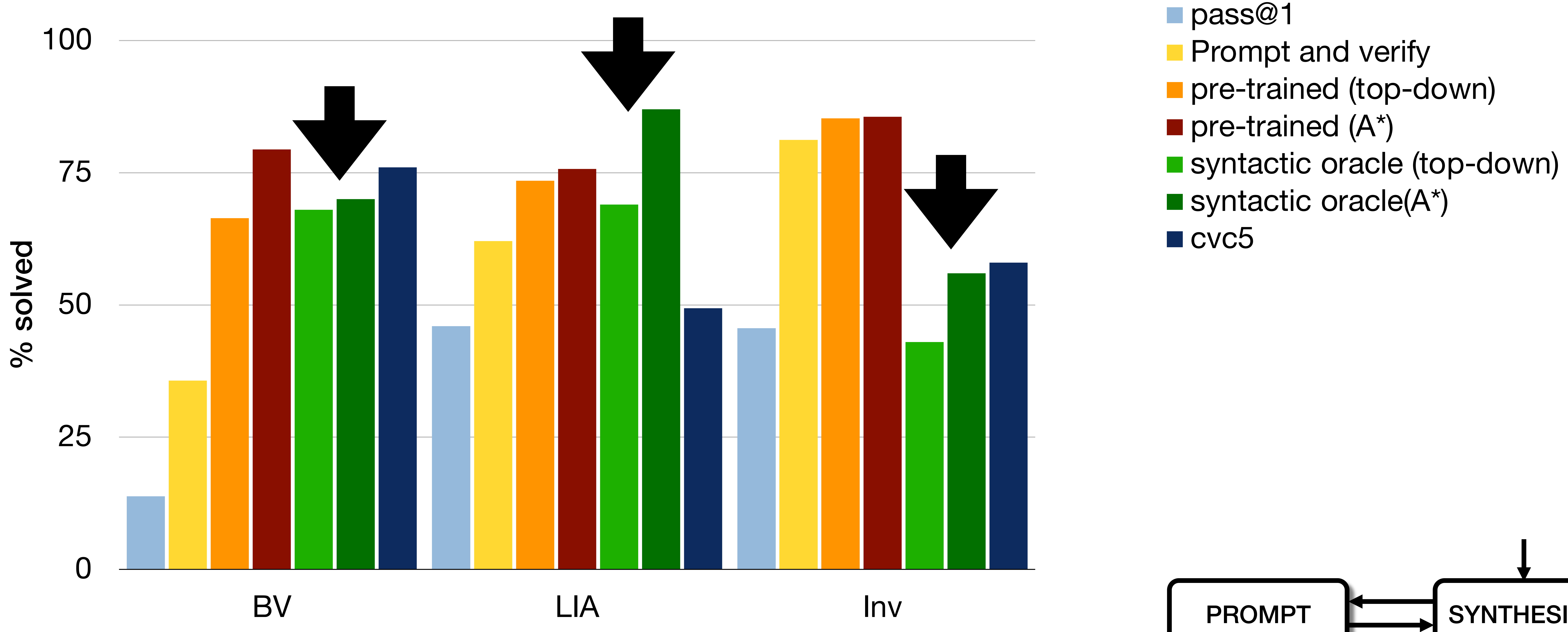
Parser

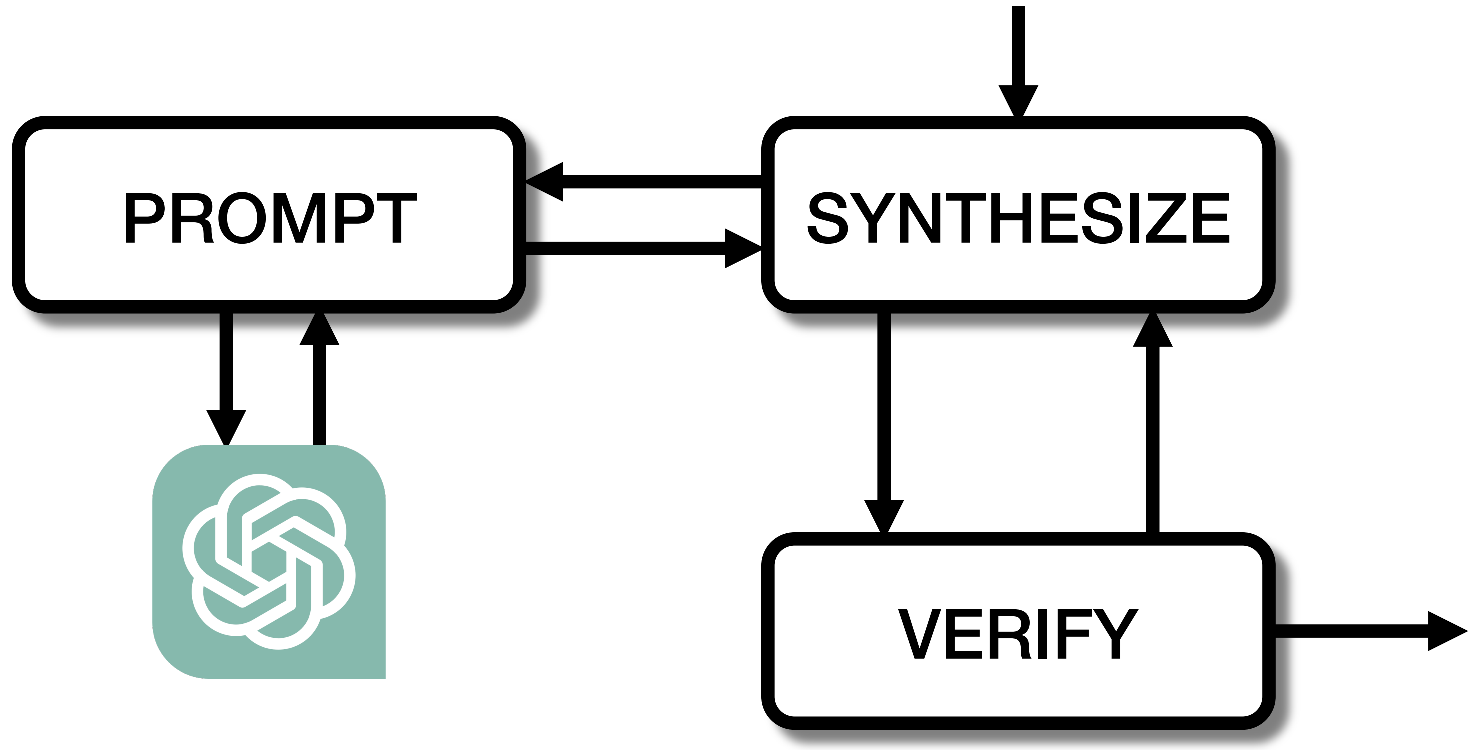
```
S → 0 (4)| 1(1) | 2(1)
S → y (8)| x (12)
S → B ? S:S (5)
B → S = S (1)
B → S ≥ S (6)
B → S ≤ S (2)
B → !B (1)
B → B ∨ B (1)
B → B ∧ B (1)
B → x ≥ 0 (1)
B → x ≥ y (1)
```



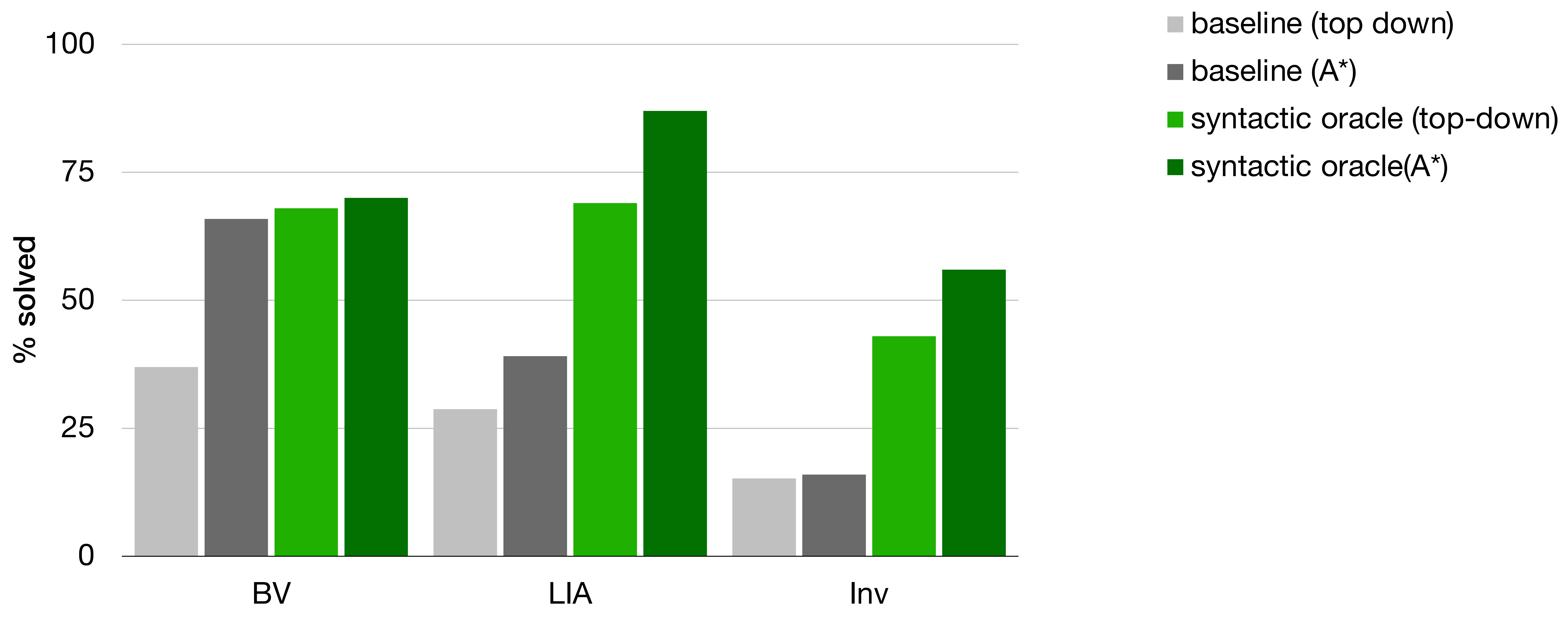
- Dynamically updating the probability distribution over grammar rules allows the oracle to make mistakes
- Cheaper prompts!

Results: syntactic oracle

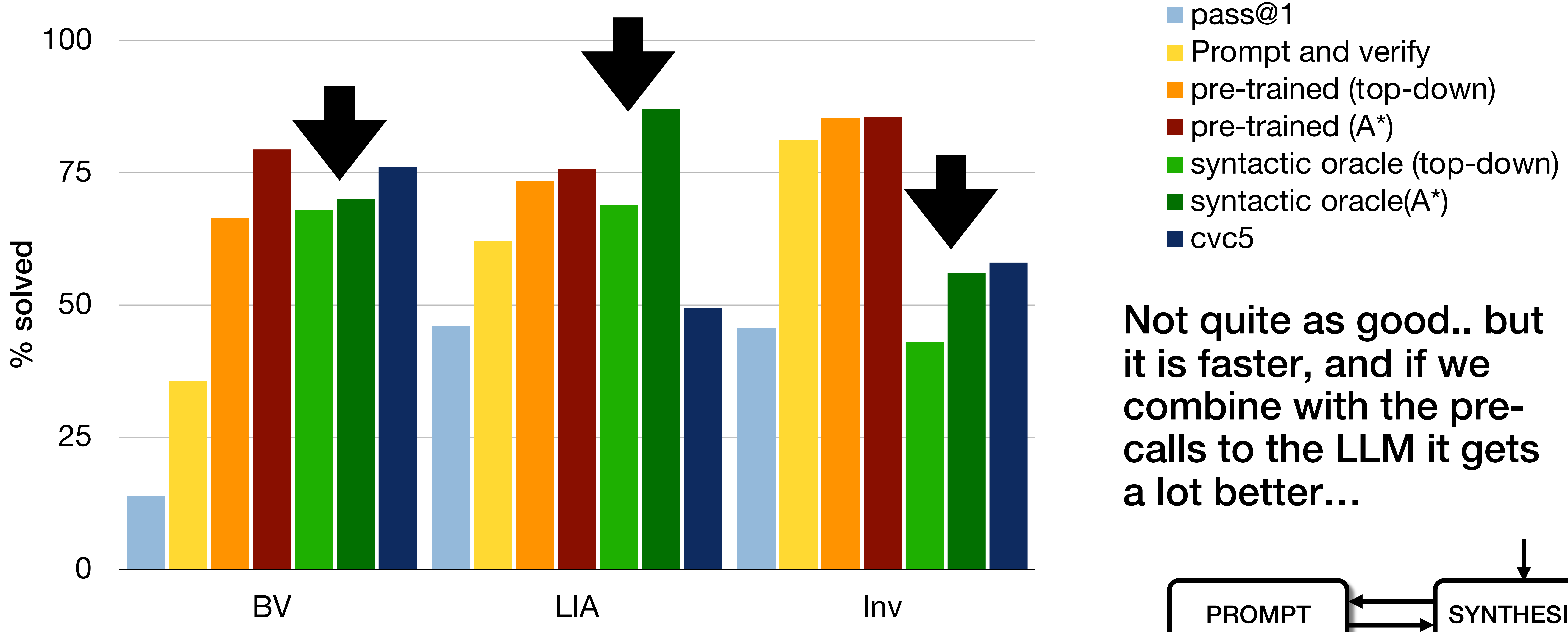




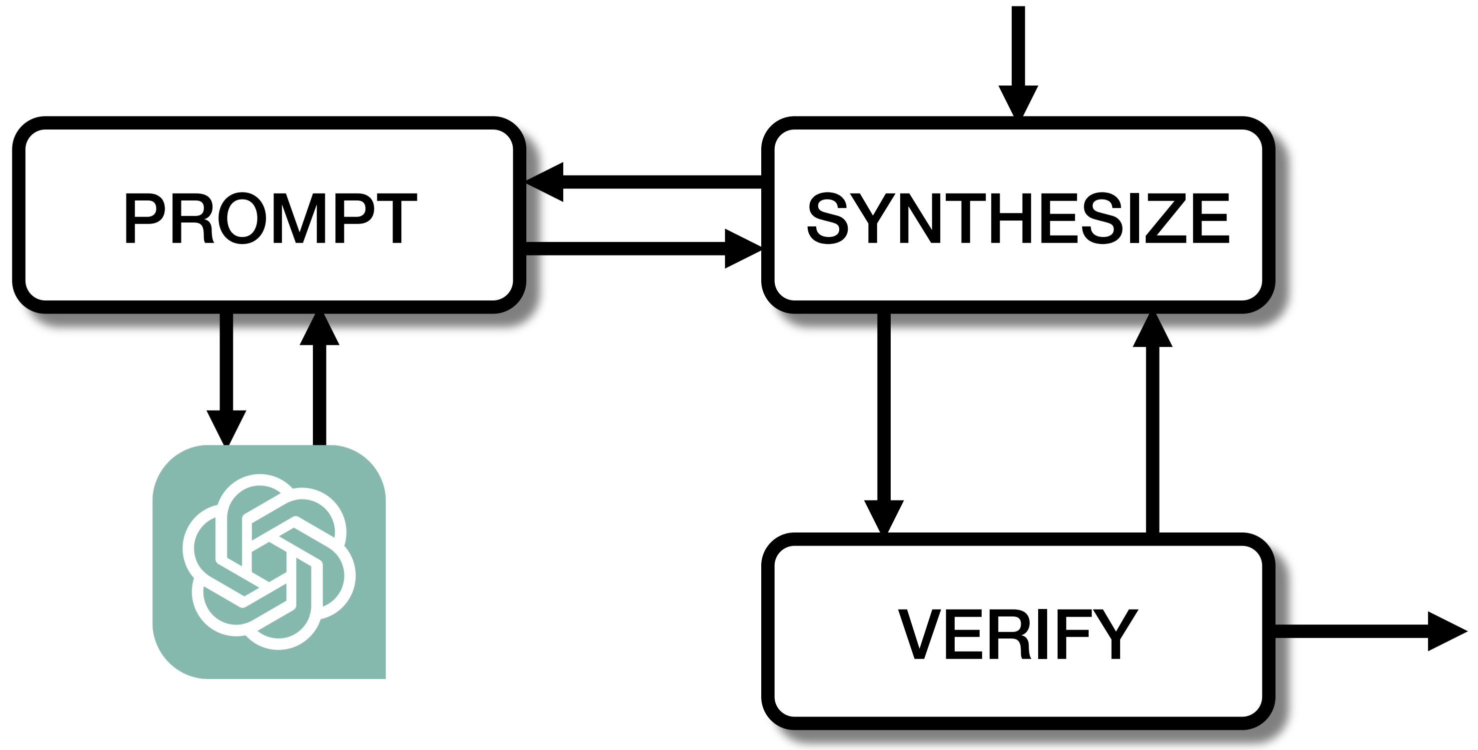
Comparison with unguided enumerators



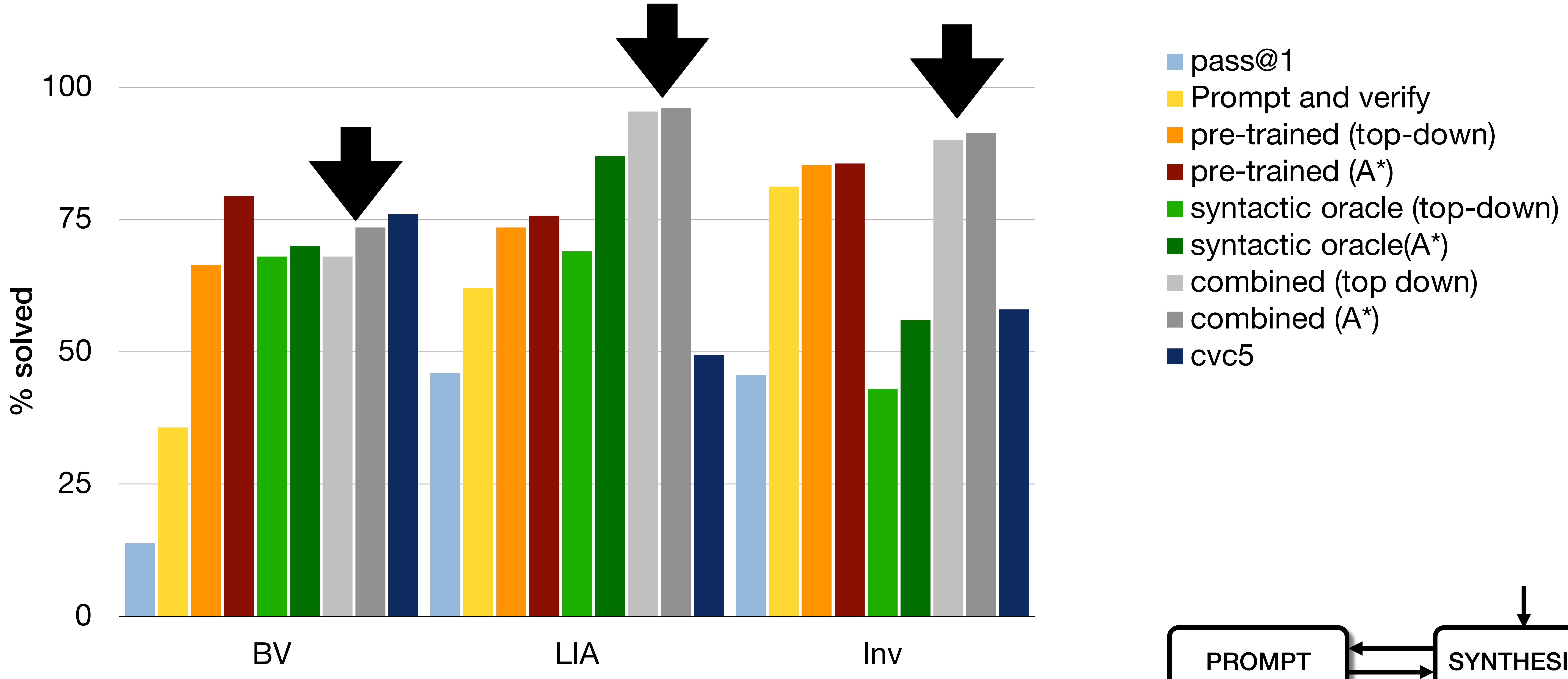
Results: syntactic oracle



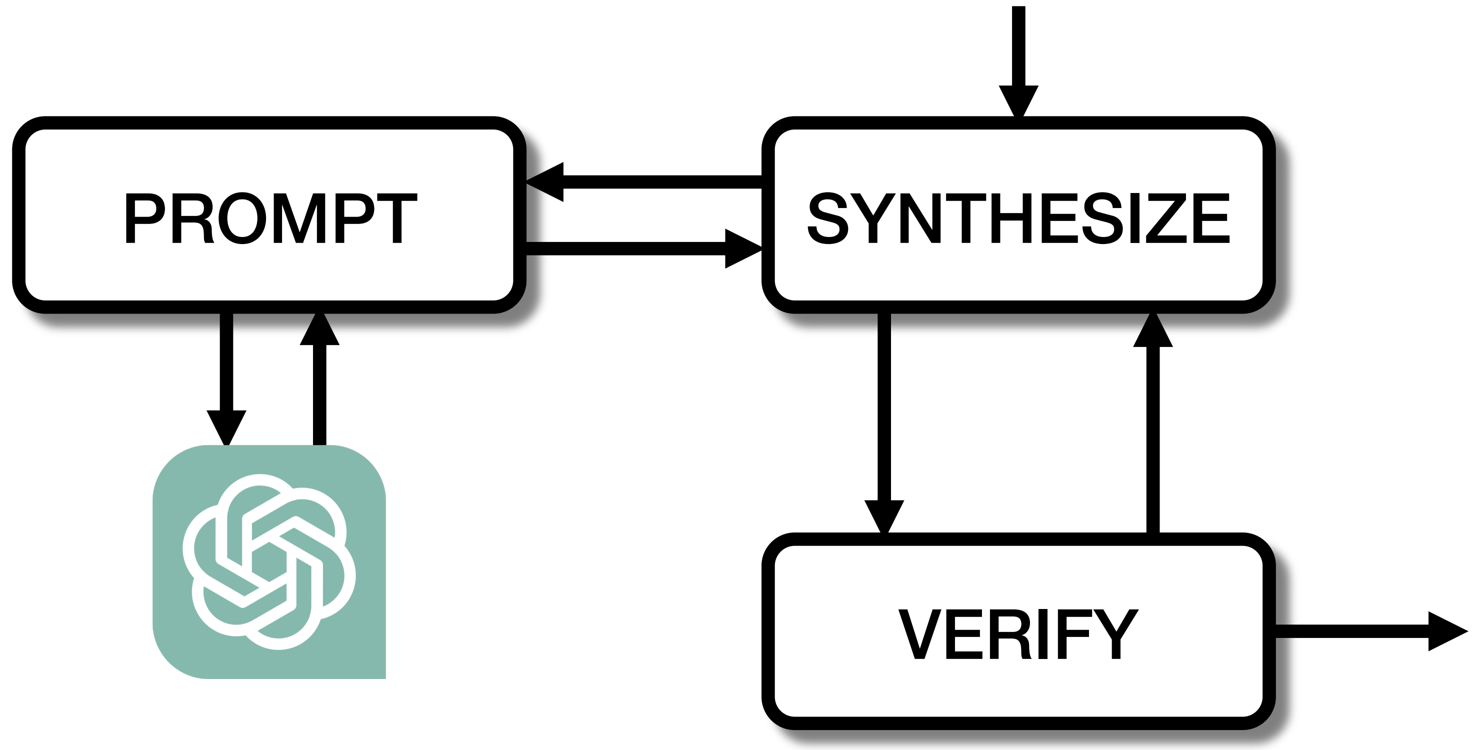
Not quite as good.. but it is faster, and if we combine with the pre-calls to the LLM it gets a lot better…



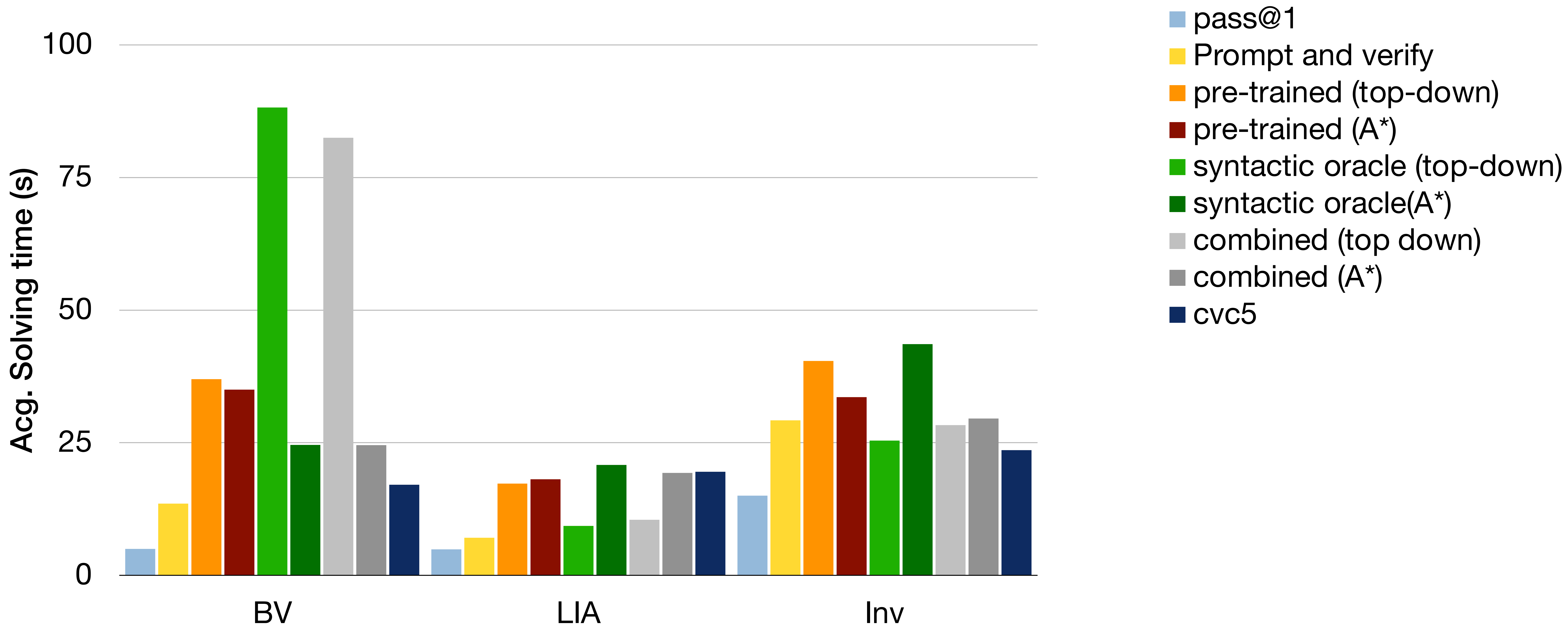
Results: syntactic oracle



Combine standalone LLM with syntactic oracle

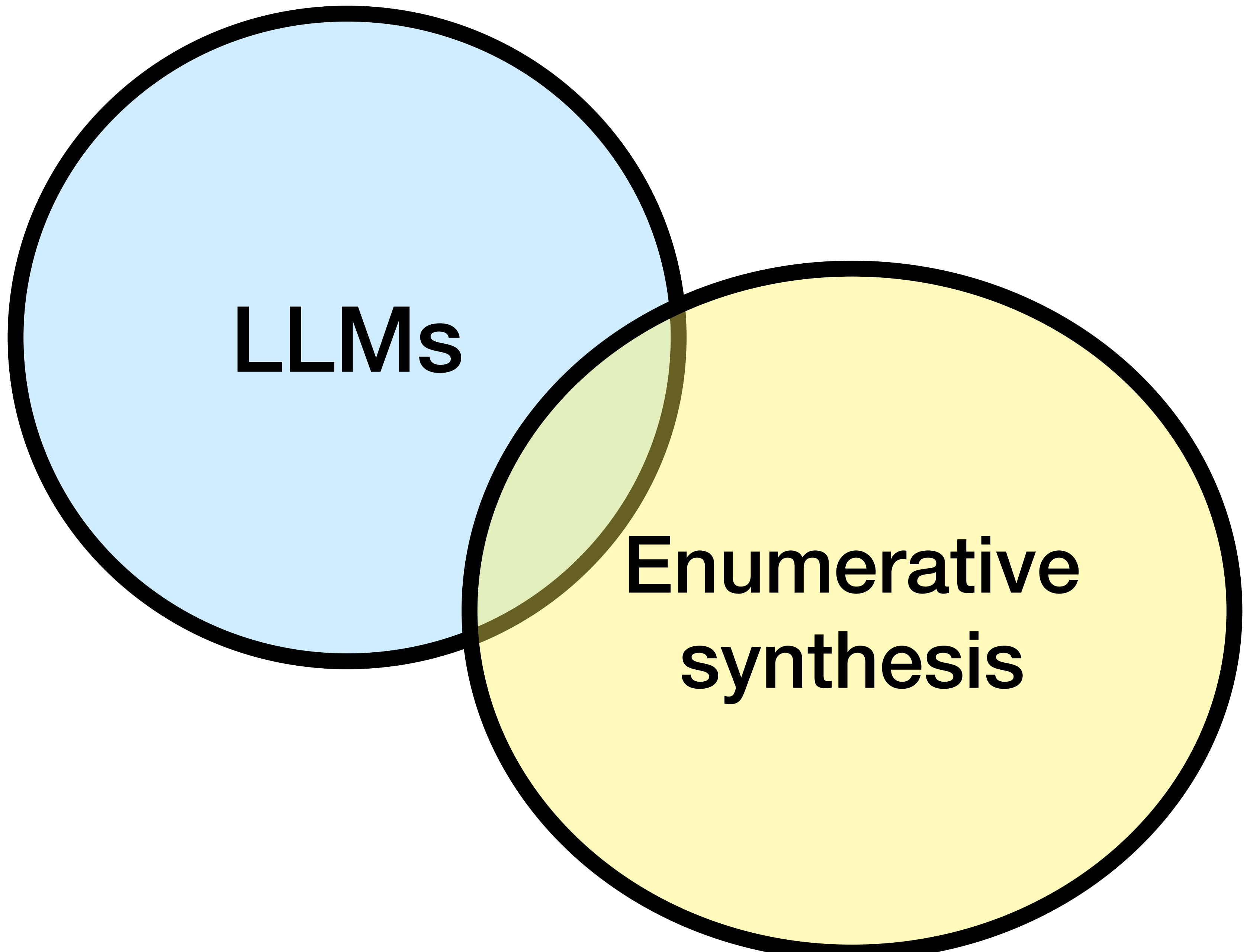


Results: solving time



Results

- Enumerative solvers are very good at problems with short solutions
- Conversely, LLMs are bad at problems with short solutions
- The pre-trained LLM guidance + enumerator is the best at long solutions
- The LLM alone performs poorly in the bitvector domain, but sees the biggest gains in combination with the enumerator



Related work

LLMs for program lifting [5]

- Spec restricted to $\exists P. \forall x. P(x) = Ref(x)$, where Ref is a reference implementation
- LLMs outperform enumeration (solving 99% of the benchmarks vs 94%)

HYSYNTH [6]

- Uses an LLM to guide bottom-up search
- Reports similar results (LLM + search outperforms LLM and search)

[5] Verified Code Transpilation with LLMs – Bhatia et al

[6] HYSYNTH: Context-Free LLM Approximation for Guiding Program Synthesis – Barke et al

Conclusions (part 1)

- LLMs are still not able to outperform state-of-the-art enumerative solvers by themselves
- But the combination of LLMs plus enumerative synthesis outperforms enumerative synthesis, and stand-alone LLMs
  - (Even with naively implemented enumerators)

Two methods for making the most out of LLMs in synthesis:

- Solving formal synthesis by guiding enumerative synthesis with LLMs:
- Generating syntactically correct models for verification, via LLMs, synthetic programming elicitation and Max-SMT solvers

But can we be more ambitious?

Synthesis = Automatically generating code that satisfies the user's specification

Synthesis = Automatically generating code that satisfies the user's specification

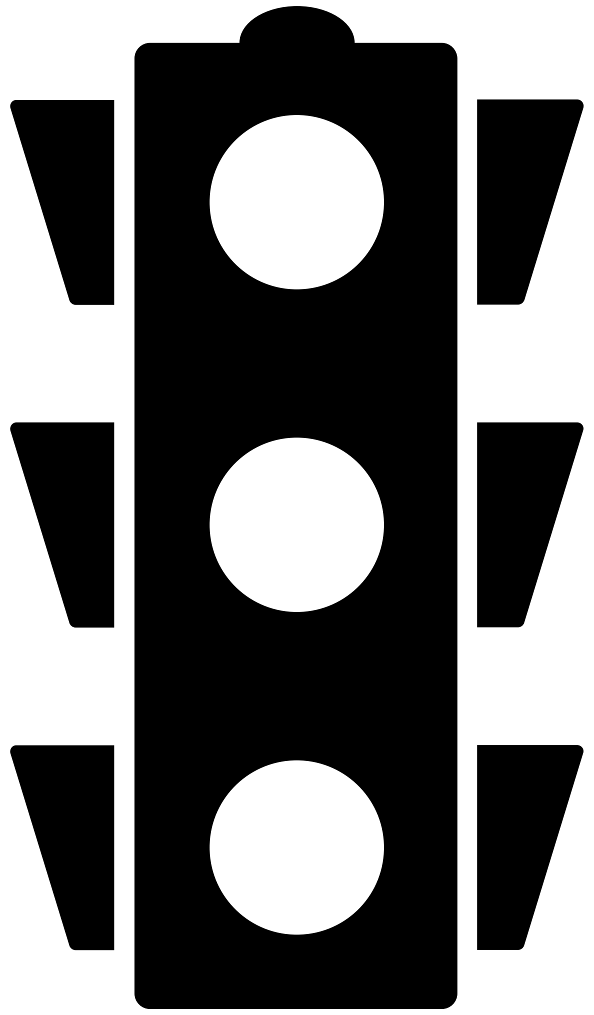
(even natural language specifications)

Given a natural language description, generate a system model for verification

UCLID5

```
module main {
 // System description.
 var a, b : integer;
 init {
   a = 0;
   b = 1;
 }
 next {
   a', b' = b, a + b;
 }
 // System specification.
 invariant a_le_b: a <= b;
 // Proof script.
 control {
   induction;
   check;
 }
}
```

Model a state machine describing a traffic light at a pedestrian crosswalk. This is a time triggered machine that assumes it will react once per second. It starts in the red state and counts 60 seconds with the help of the variable count. It then transitions to green, where it will remain until the pure input pedestrian is present. That input is generated by some other subsystem that detects when a pedestrian is present, and should be modelled as nondeterministic. When pedestrian is present, the machine transitions to yellow if it has been green for at least 60 seconds. ...



```
module TrafficLight {
  var sigG, sigR, sigY: boolean;
  var count, state: integer;
  var pedestrian: boolean;
  init {
    sigG = false; sigY = false; sigR = true;
    state = 0; count = 0; pedestrian = false; }
  procedure step()
    modifies sigG; modifies sigY; modifies sigR;
    modifies count; modifies state; {
    if (state == 0) {
      sigG = false; sigY = false; sigR = true;
      count = 0;
      if (count < 60) { count = (count + 1); }
...
```

Way beyond enumerative synthesis without serious user guidance.

Way beyond enumerative synthesis without serious user guidance.

Synthesis in UCLID5

Federico Mora
University of California, Berkeley

Kevin Cheang
University of California, Berkeley

Elizabeth Polgreen
University of California, Berkeley

Sanjit A. Seshia
University of California, Berkeley

Abstract

We describe an integration of program synthesis into UCLID5, a formal modelling and verification tool. To the best of our knowledge, the new version of UCLID5 is the only tool that supports program synthesis with bounded model checking, k-induction, sequential program verification, and hyperproperty verification. We use the integration to generate 25 program synthesis benchmarks with simple, known solutions that are out of reach of current synthesis engines, and we release the benchmarks to the community.

to synthesize called `h` at lines 16 and 17, and then uses `h` at line 18 to strengthen the existing set of invariants. Given this input, UCLID5, using e.g. CVC4 [2] as a synthesis engine, will automatically generate the function `h(x, y) = x >= 0`, which completes the inductive proof.

In this example, the function to synthesize represents an inductive invariant. However, functions to synthesize are treated exactly like any interpreted function in UCLID5: the user could have called `h` anywhere in the code. Furthermore, this example uses induction and a global invariant, however, the user could also have used a linear temporal logic (LTL)

(even much smaller problems are out of reach of the state-of-the-art enumerative solvers)

What about LLMs?

Towards AI-Assisted Synthesis of Verified Dafny Methods

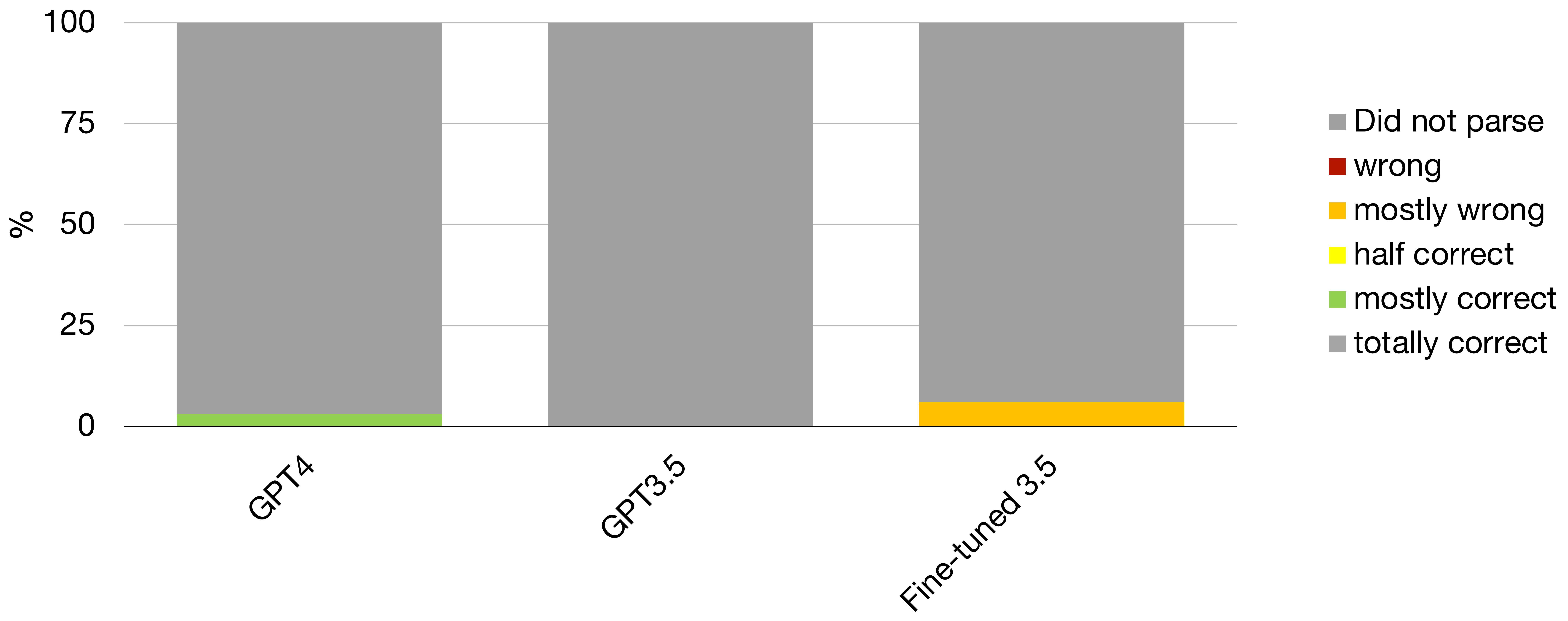
MD RAKIB HOSSAIN MISU*, University of California Irvine, USA
CRISTINA V. LOPES, University of California Irvine, USA
IRIS MA, University of California Irvine, USA
JAMES NOBLE†, Creative Research & Programming, New Zealand

Large language models show great promise in many domains, including programming. A promise is easy to make but hard to keep, and language models often fail to keep their promises, generating erroneous code. A promising avenue to keep models honest is to incorporate formal verification: generating programs' specifications as well as code, so that the code can be proved correct with respect to the specifications. Unfortunately, existing large language models show a severe lack of proficiency in verified programming.

In this paper, we demonstrate how to improve two pretrained models' proficiency in the Dafny verification-aware language. Using 178 problems from the MBPP dataset, we prompt two contemporary models (GPT-4 and PaLM-2) to synthesize Dafny methods. We use three different types of prompts: a direct *Contextless* prompt; a *Signature* prompt that includes a method signature and test cases, and a *Chain of Thought (CoT)* prompt that decomposes the problem into steps and includes retrieval augmentation generated example problems and solutions. Our results show that GPT-4 performs better than PaLM-2 on these tasks, and that both models perform best with the retrieval augmentation generated CoT prompt. GPT-4 was able to generate verified,

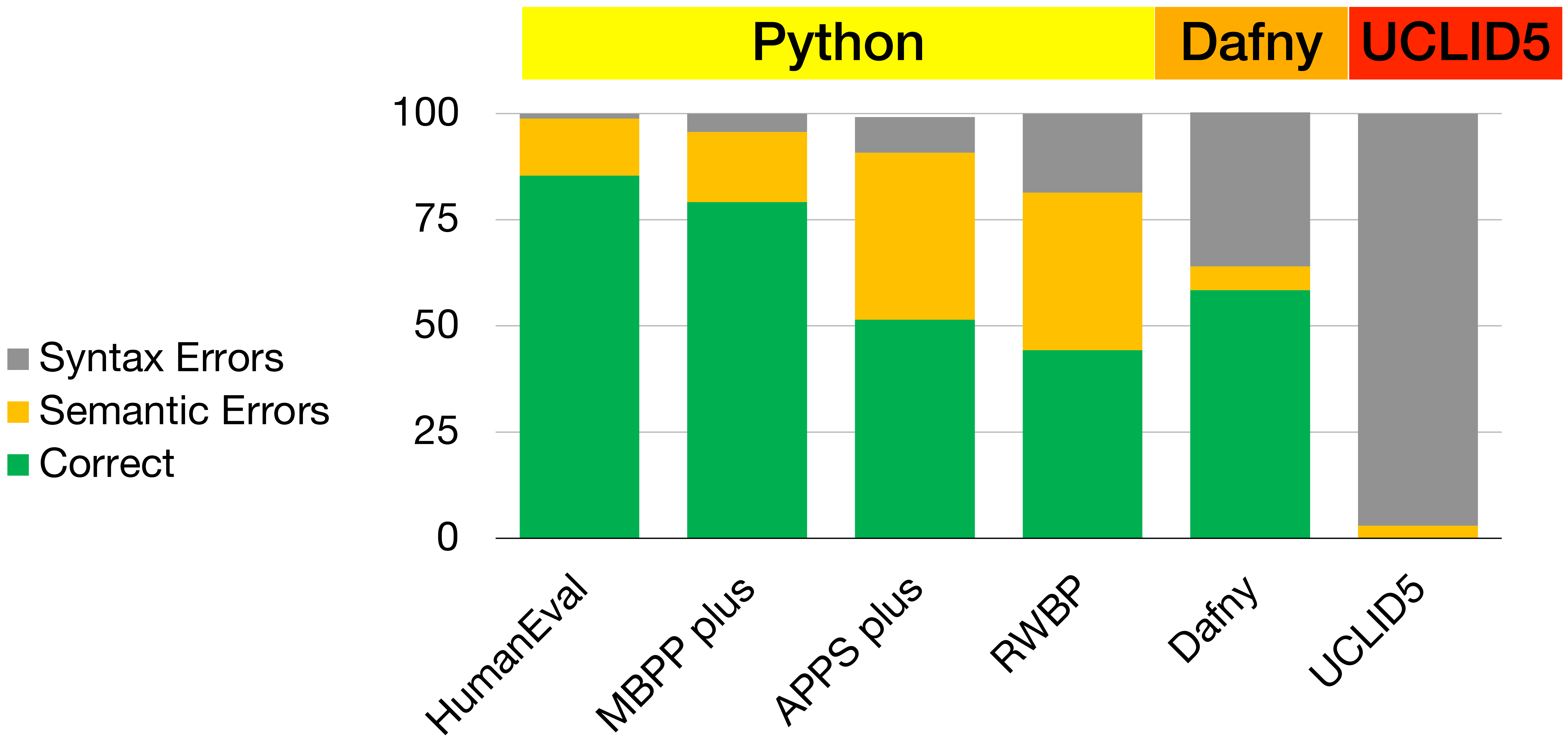
Related work suggests this is worth a try..

Syntax errors are a problem!



Syntax errors are a problem!

Results for GPT-4 [13][14], all pass@1 except Dafny

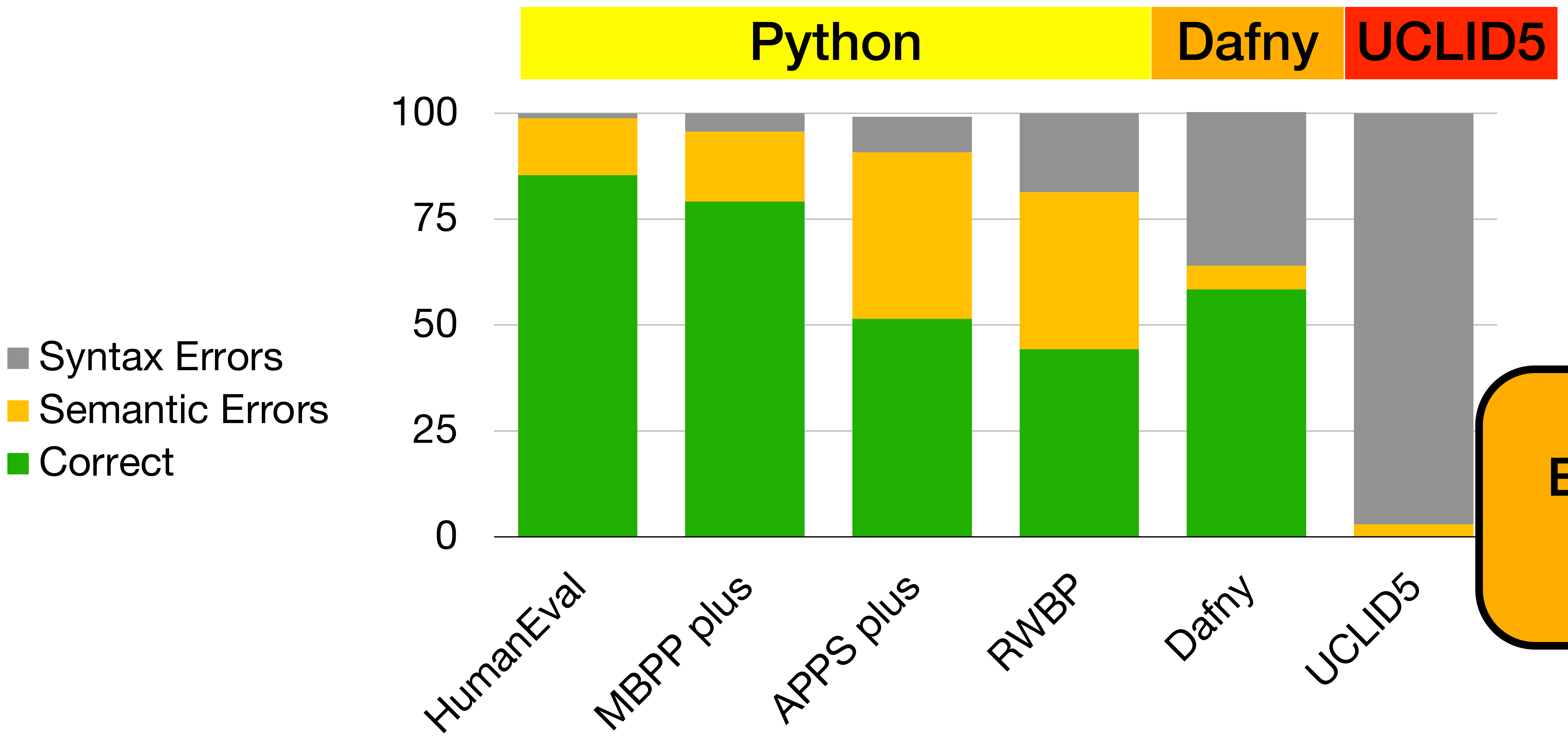


[13] What's wrong with your code generated by large language models? An Extensive Study - Duo et al

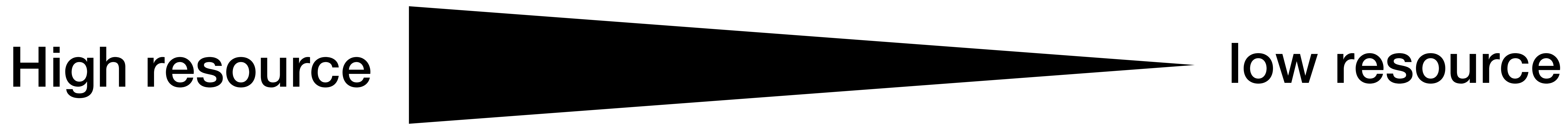
[14] Towards AI assisted synthesis of verified Dafny methods - Misu et al

Syntax errors are a problem!

Results for GPT-4 [13][14], all pass@1 except Dafny



Especially for very low-resource languages!!



[13] What's wrong with your code generated by large language models? An Extensive Study - Duo et al

[14] Towards AI assisted synthesis of verified Dafny methods - Misu et al

Especially for very low-resource languages!!

Internal/ proprietary APIs

Tool specific verification/ specification languages

New DSLs, e.g., for new hardware

Given a natural language description, generate a system model for verification

UCLID5

```
module main {
 // System description.
 var a, b : integer;
 init {
   a = 0;
   b = 1;
 }
 next {
   a', b' = b, a + b;
 }
 // System specification.
 invariant a_le_b: a <= b;
 // Proof script.
 control {
   induction;
   check;
 }
}
```

Synthetic Programming Elicitation and Repair for Text-to-Code in Very Low-Resource Programming Languages

Federico Mora[1] Justin Wong[1] Haley Lepe[2] Sahil Bhatia[1] Karim Elmaaroufi[1]
George Varghese[3] Joseph E. González[1] Elizabeth Polgreen[4] Sanjit A. Seshia[1]

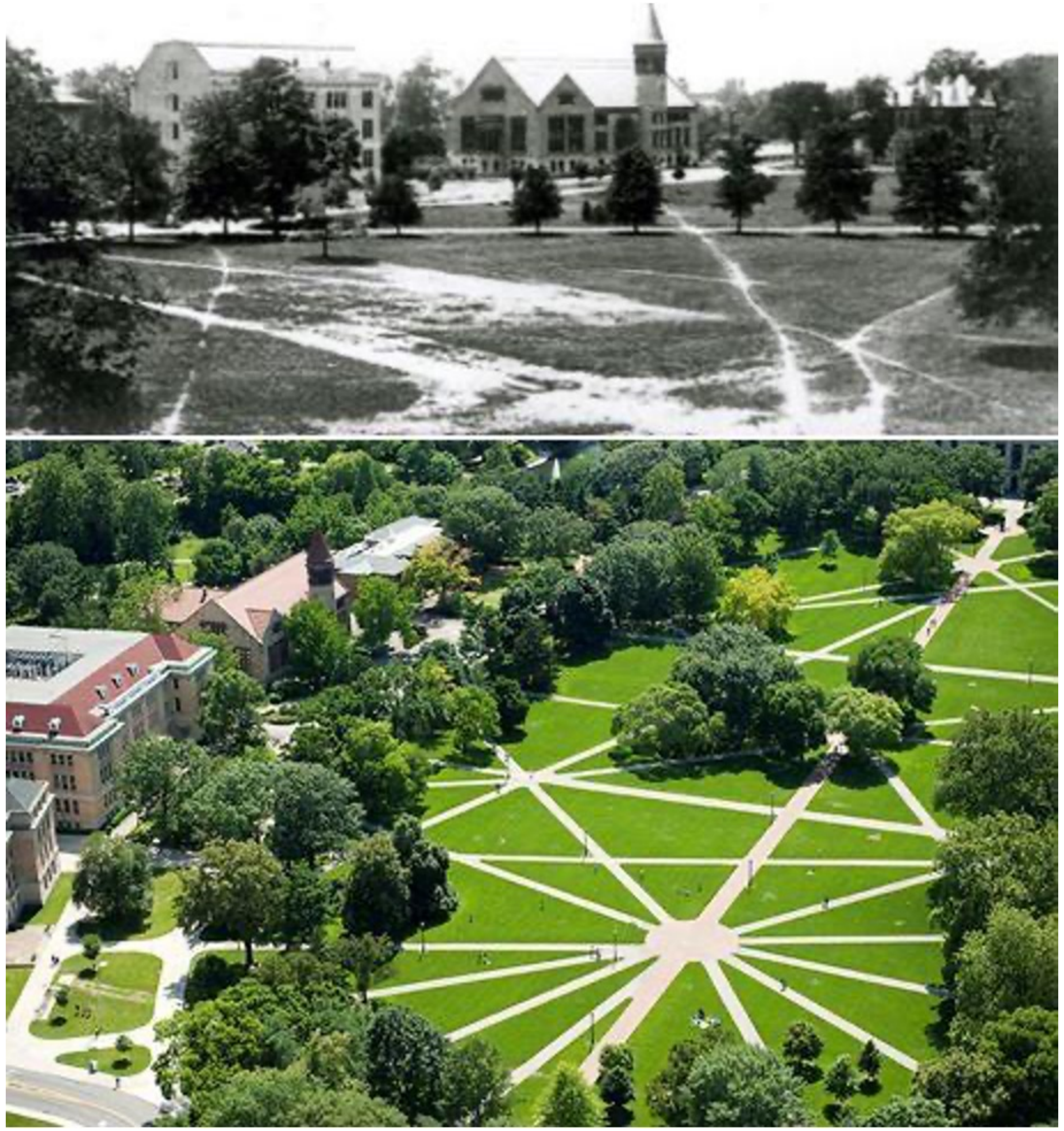
[1]UC Berkeley [2]MiraCosta Community College
[3]UCLA [4]University of Edinburgh

Natural programming elicitation:

“asking non-programmers or novice programmers to express programs with the concepts and abstractions they find most natural” [7]

Coined by Brad Myers in 2004 [8]

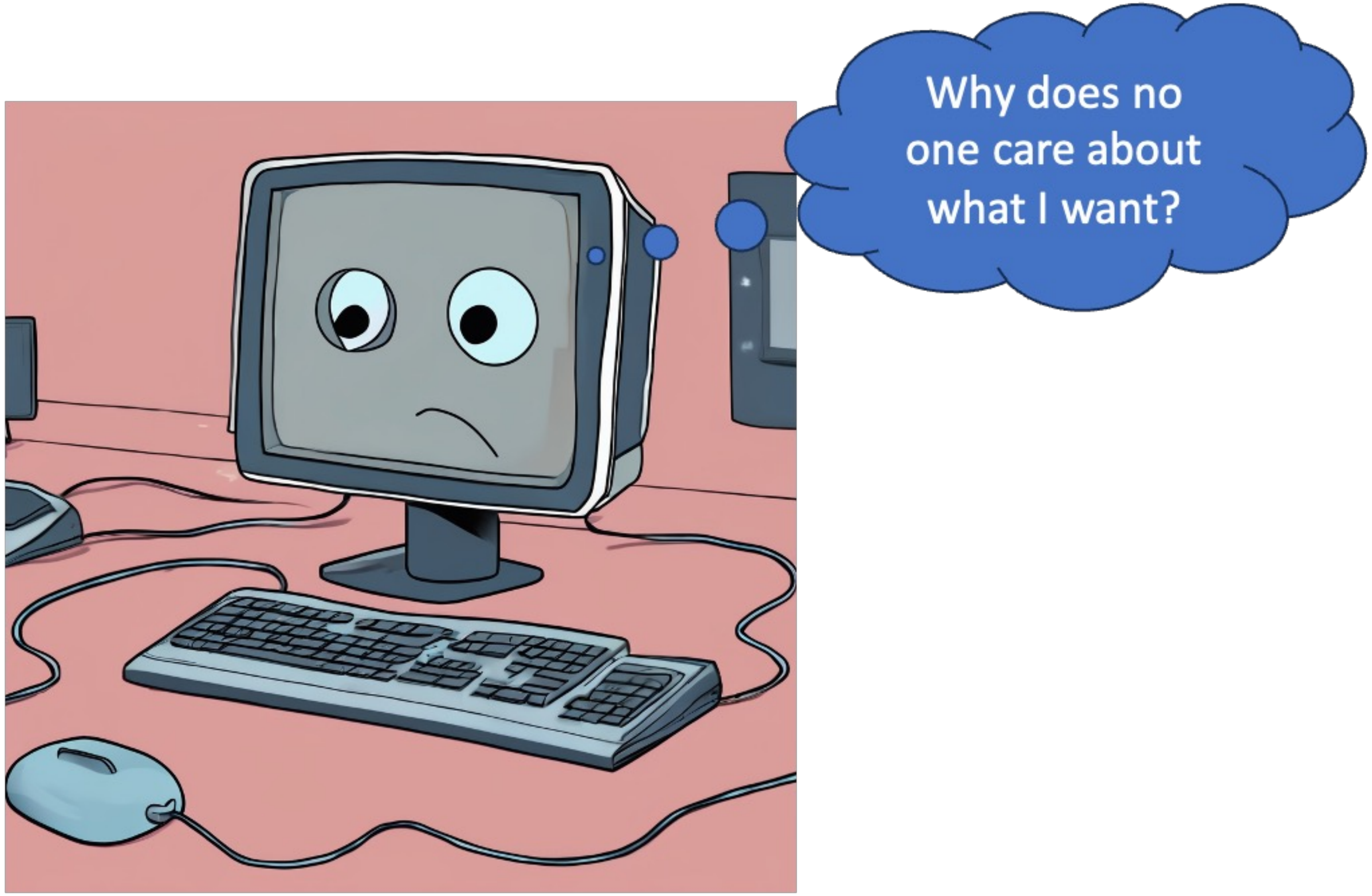
Once you know what people “naturally” do, design for it.



[7] How statically-typed functional programmers write code - Lubin et al

[8] Natural programming languages and environments – Myers et al

Synthetic programming elicitation: the same but for LLMs



Synthetic programming elicitation: the same but for LLMs

Prompt engineering: trying to persuade the LLM to do what you want

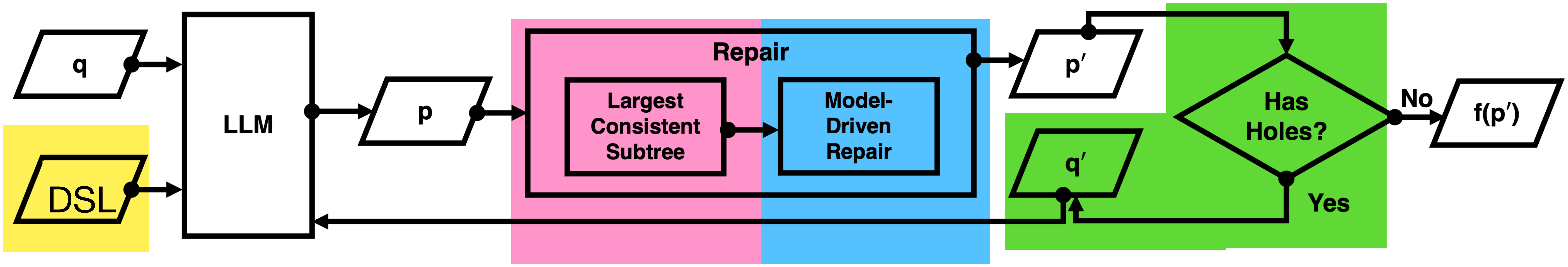
Our approach: observe what the LLM does, and design a DSL for the LLM.

Synthetic programming elicitation and repair

If the LLM goes wrong, and we can fix it with force, we do.

If we can’t fix the errors, give the LLM guidance to get it back to the DSL

Overview



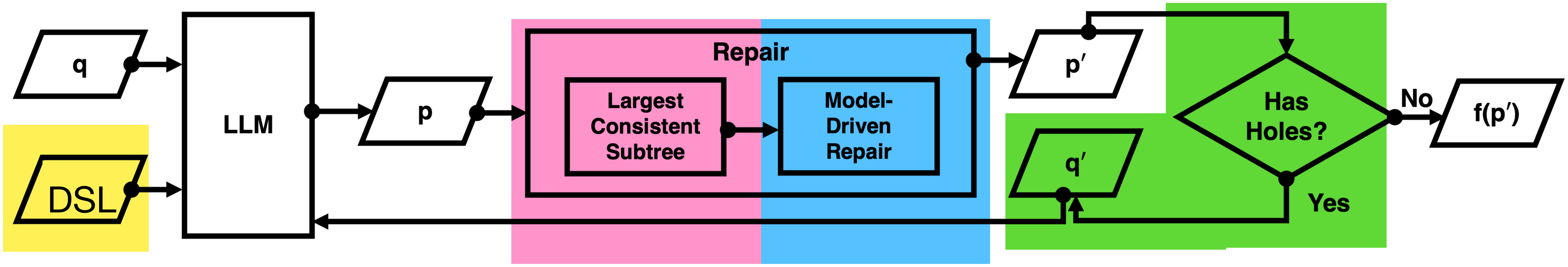
Design a DSL for the LLM

Parse response generously

Repair via SMT solver

Repair via LLM

Overview

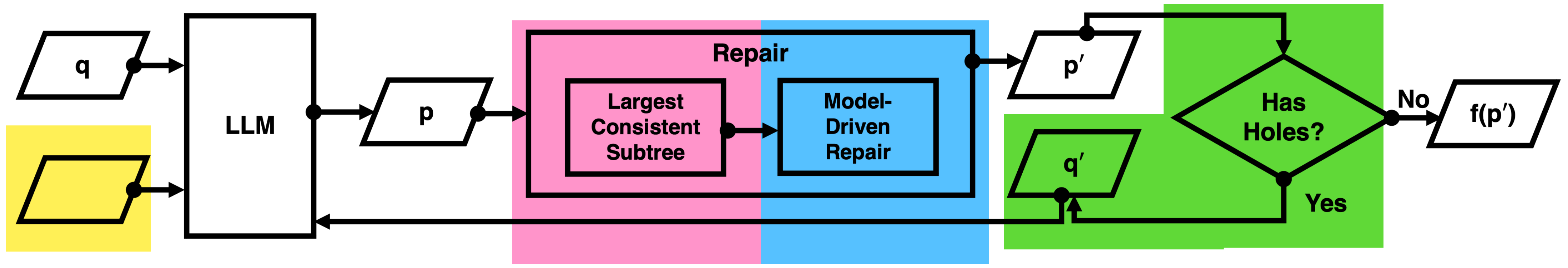


Design a DSL for the LLM

Parse response generously

Repair via SMT solver

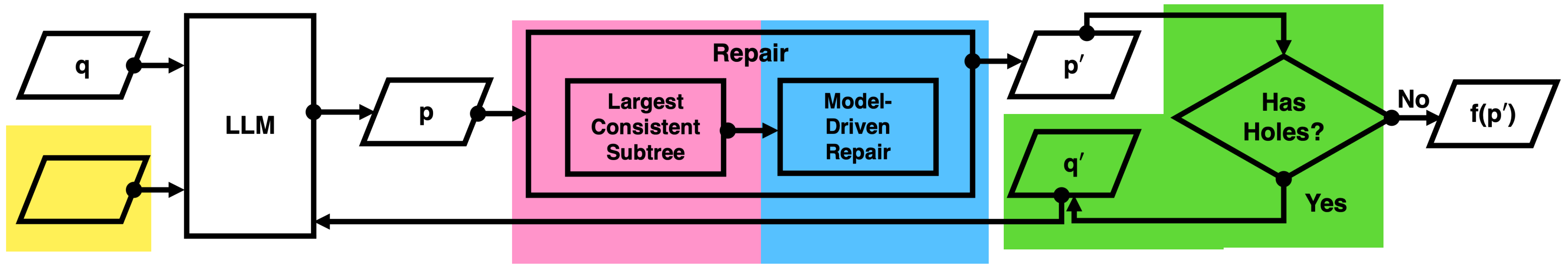
Repair via LLM



Choosing the DSL

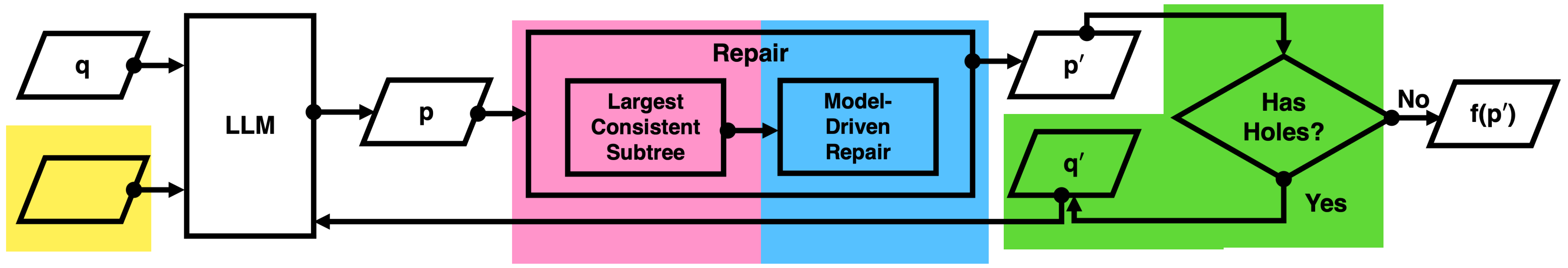
Synthetic programming elicitation:

- Collate a set of “training” data
- Call LLM on all data
- Analyse responses and choose language accordingly



Choosing the DSL

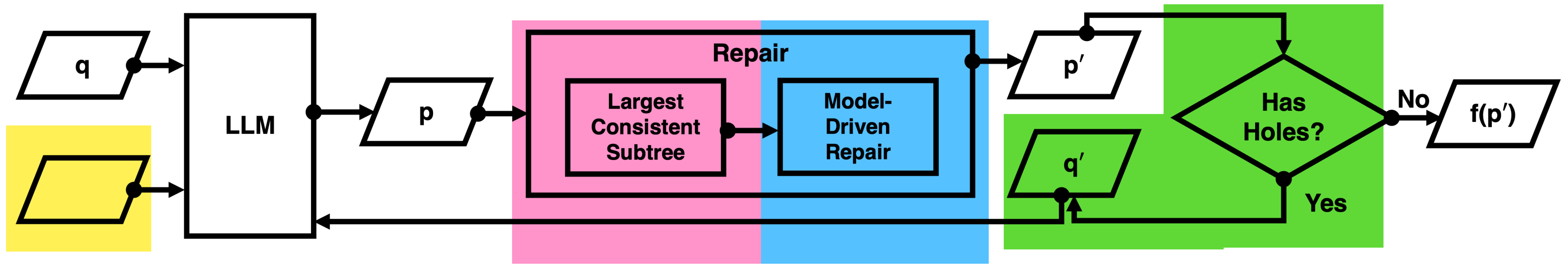
- Participant: gpt-4-0613
- Data:
  - 82 UCLID5 regression tests
  - Each test consists of:
    - A description string
    - UCLID5 code
- Tasks (for each test):
  - Ask the LLM to write Python code that implements the test description



Choosing the DSL

Synthetic programming elicitation:

- Collate a set of “training” data
- Call LLM on all data
- Analyse responses and choose language accordingly



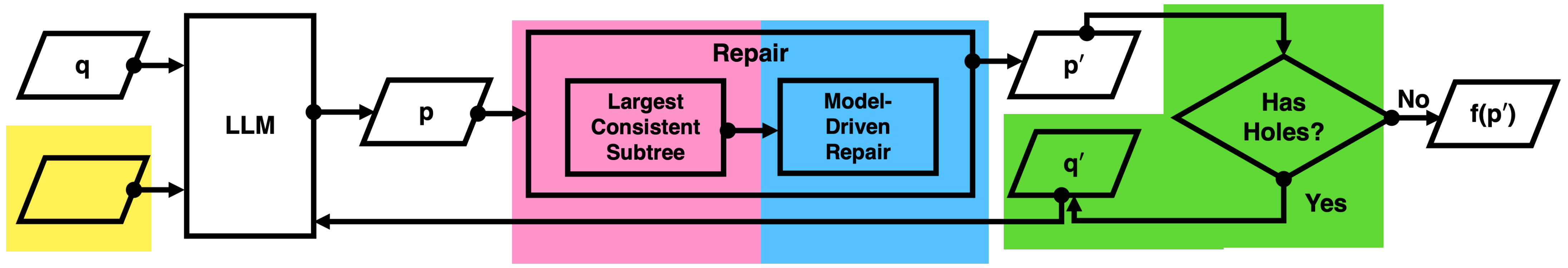
Choosing the DSL

Example task:

```
# You are an expert user of FormalVerificationLibrary, a Python
API for the formal modeling and verification of transition
systems and procedures. Please write Python code using the
FormalVerificationLibrary API that fits the description below.
Do not worry if you do not remember the names of particular
functions or classes.




# The module has two integer variables and two bitvector
variables. It should use inline assertions in the init block to
check that we can correctly divide two integers, and correctly
divide two bitvectors. It should check that signed and unsigned
division report different results for very big bitvectors.
```

Generic Prompt

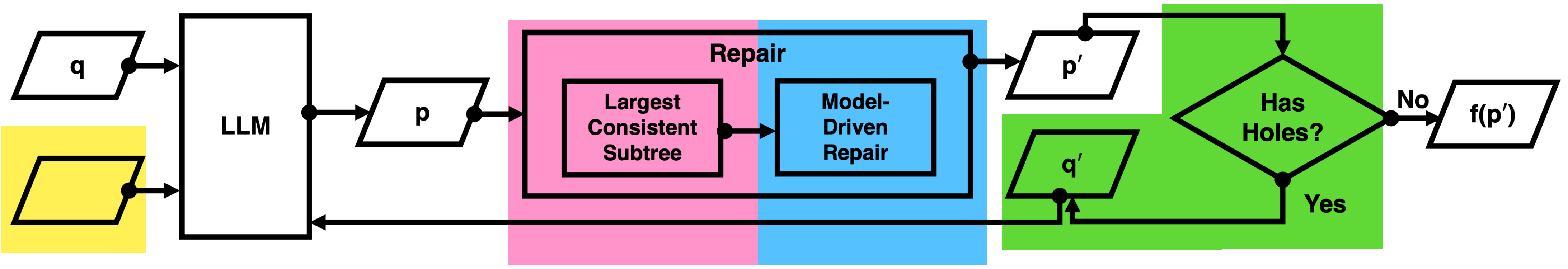
Task Specific



Choosing the DSL

Synthetic programming elicitation:

- Collate a set of “training” data
- Call LLM on all data
- Analyse responses and choose language accordingly



Choosing the DSL

Example respose:

```
from FormalVerificationLibrar  import IntegerVariable,                 , Module

class SampleModel(Module)
    def __init__(self):
        # defining integer  ariables
        self.int_var1 = IntegerVariable()
        self.int_var2 = IntegerVariable()

        # initial assertions for integer variables
        assert(self.int_var2 != 0)
        int_result = self.int_var1 / self.int_var2
        assert(int_result * self.int_var2 + self.int_var1 % self.int_var2 == self.int_var1)

        # defining bitvector variables
        self.bvec_var1 = BitVectorVariable()
        self.bvec_var2 = BitVectorVariable()

        # initial assertions for bitvector variables
        assert(self.bvec_var2 != 0)
        bvec_result_unsigned = self.bvec_var1.udiv(self.bv  var2)
        bvec_result_signed = self.bvec_var1.sdiv(self.bv
        assert(bvec_result_unsigned * self.bvec_var2 +                     elf.bvec_var1)
        assert(bvec_result_signed * self.bvec_var2 + sel                   f.bvec_var1)

        # check if signed and unsigned division return diffe ent results for large bitvectors
        self.bvec_var1 = BitVectorVariable(2**64 - 1)
        self.bvec_var2 = BitVectorVariable(2)
        bvec_result_unsigned_big = self.bvec_var1.udiv(
        bvec_result_signed_big = self.bvec_var1.sdiv(sel
        assert(bvec_result_unsigned_big != bvec_result_sig  _big)
```

Importing Sorts

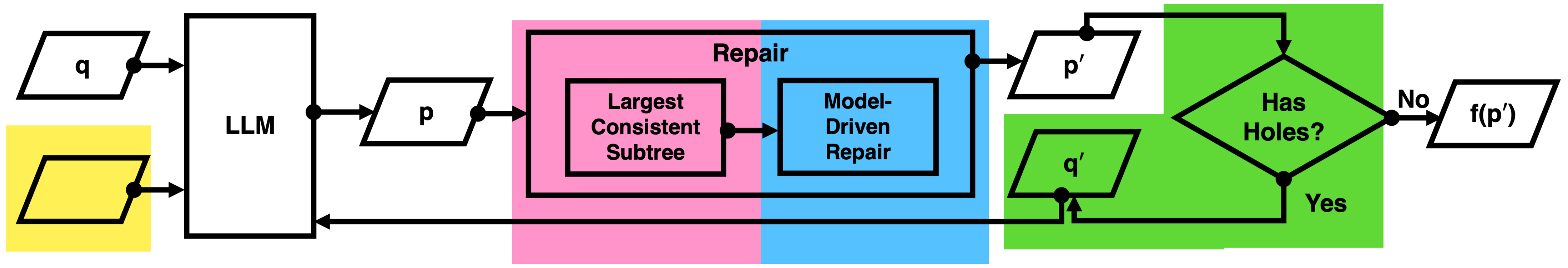
Extending “Module”

Using Python constructs

Assignments for declarations

Using dunder methods

Very object oriented



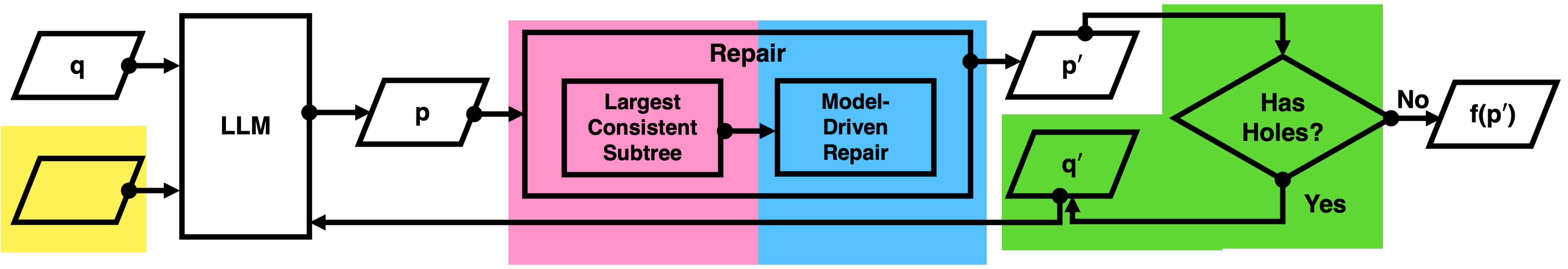
Choosing the DSL

Another example task:

Generic Prompt

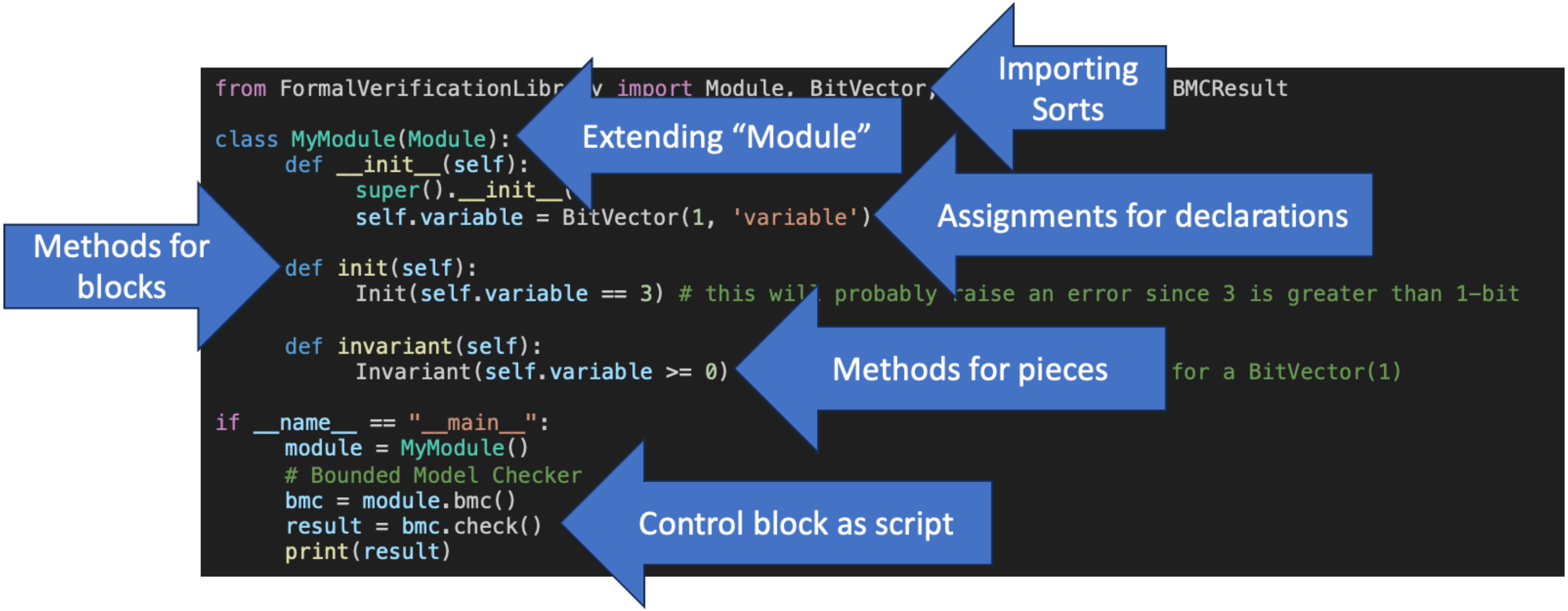
```
# You are an expert user of FormalVerificationLibrary, a Python API for
the formal modeling and verification of transition systems and procedures.
Please write Python code using the FormalVerificationLibrary API that fits
the description below. Do not worry if you do not remember the names of
particular functions or classes.

# A module with a single 1-bit bit-vector variable. The init block sets
the variable to 3, which is larger than the width of the variable. A
trivial invariant about that variable and a bmc check.
```

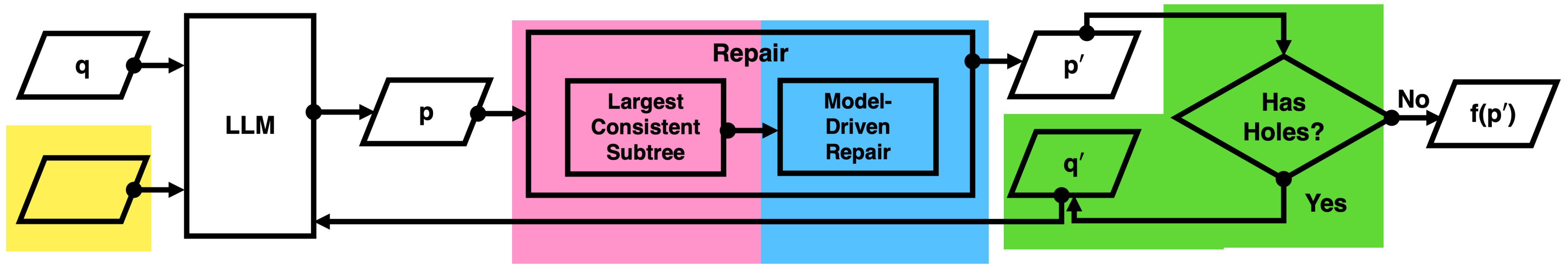
Task Specific



Choosing the DSL

Another example response

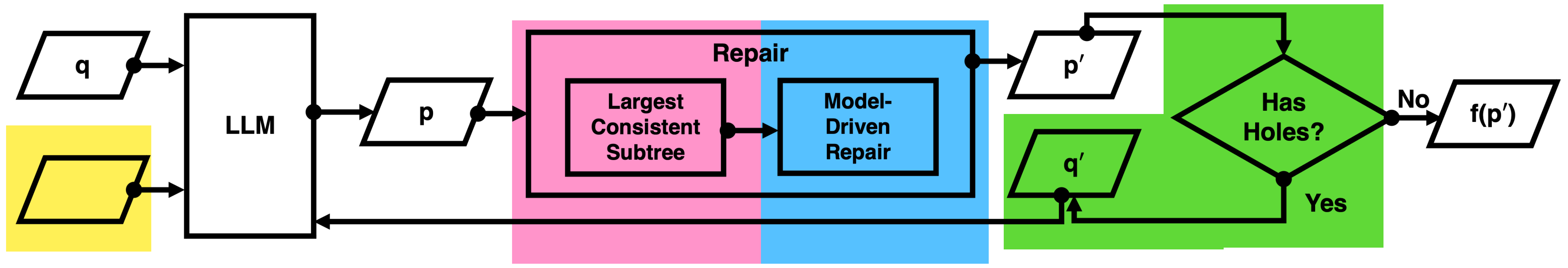




Choosing the DSL

Synthetic programming elicitation:

- Collate a set of “training” data
- Call LLM on all data
- Analyse responses and choose language accordingly



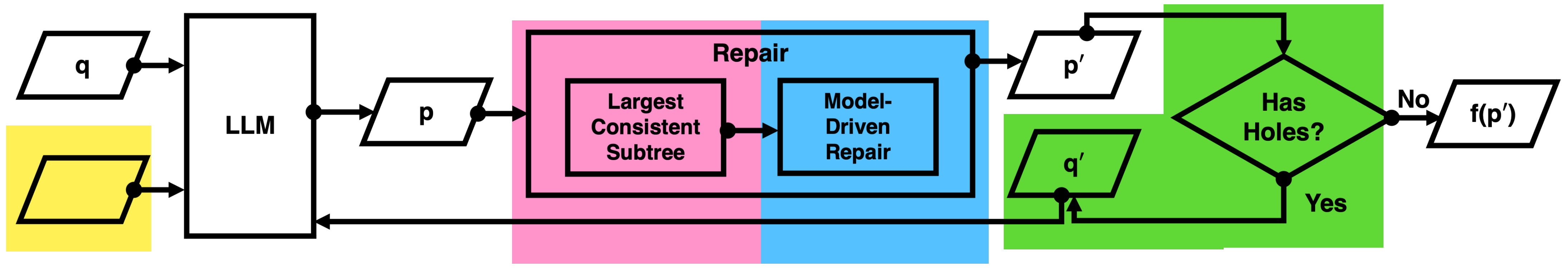
Choosing the DSL

Very object oriented!

When common Python uses a feature, the LLM will use it

Otherwise, the LLM will find a workaround using imports

None of the outputs were syntactically incorrect Python code (parsable)



Choosing the DSL

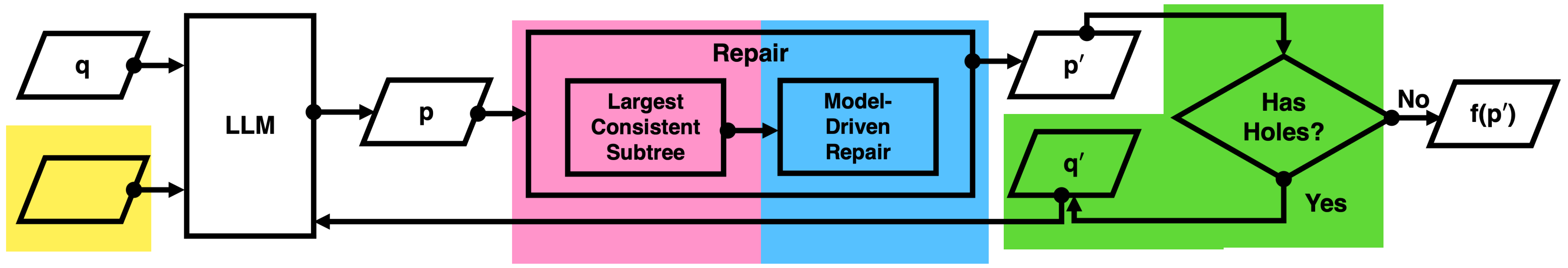
Very object oriented!

When common Python uses a feature, the LLM will use it

Otherwise, the LLM will find a workaround using imports

None of the outputs were syntactically incorrect Python code (parsable)

Note that there was no DSL for UCLID5 in Python like this before.

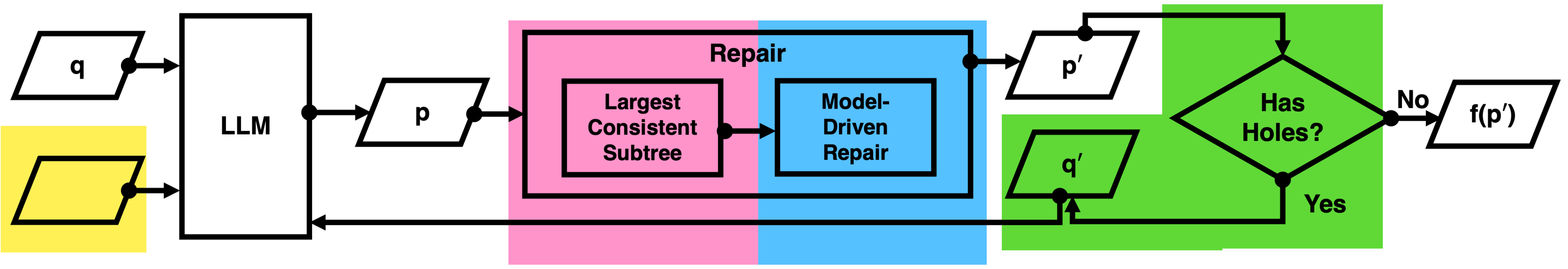


Choosing the DSL

The DSL

- is a strict subset of Python

- every string in the DSL can be translated to syntactically correct UCLID5 code



Choosing the DSL

- We can now use minimal prompting to interface to the LLM

Write python code to complete the following task:

[TASK]

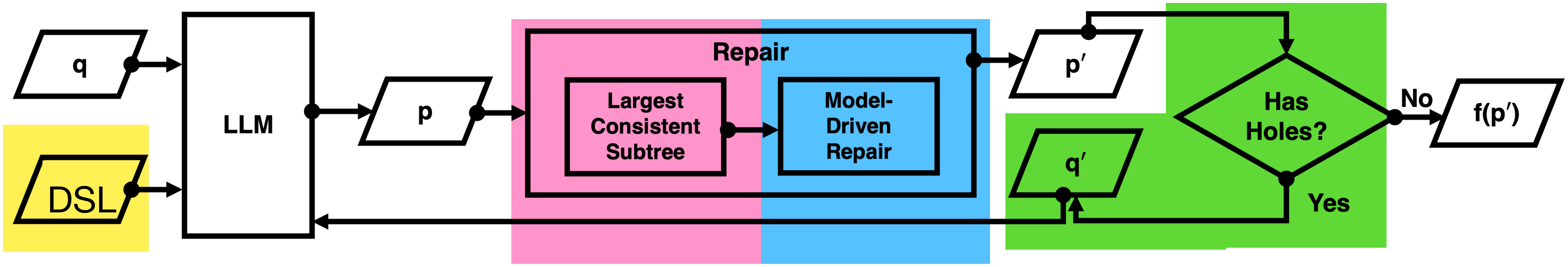
Reply with your code inside one unique code block.

[Describe FormalVerificationLibrary]

I can definitely do that. Here's the code:

[…]

Overview

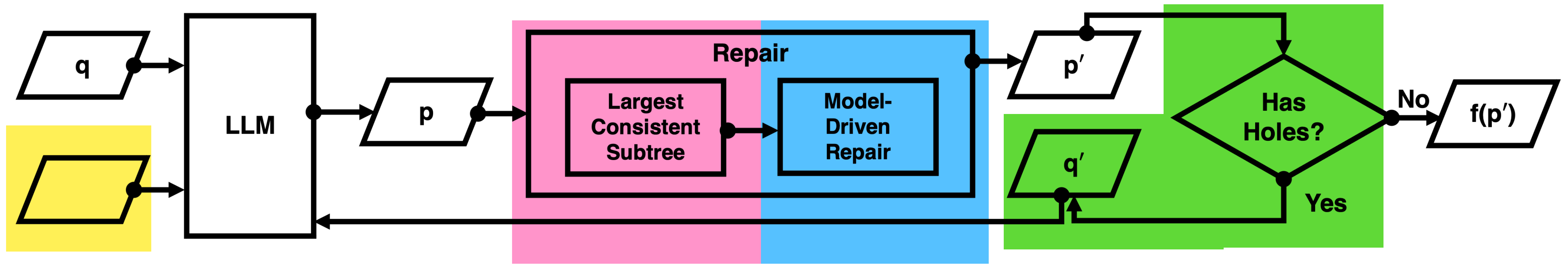


Design a DSL for the LLM

Parse response generously

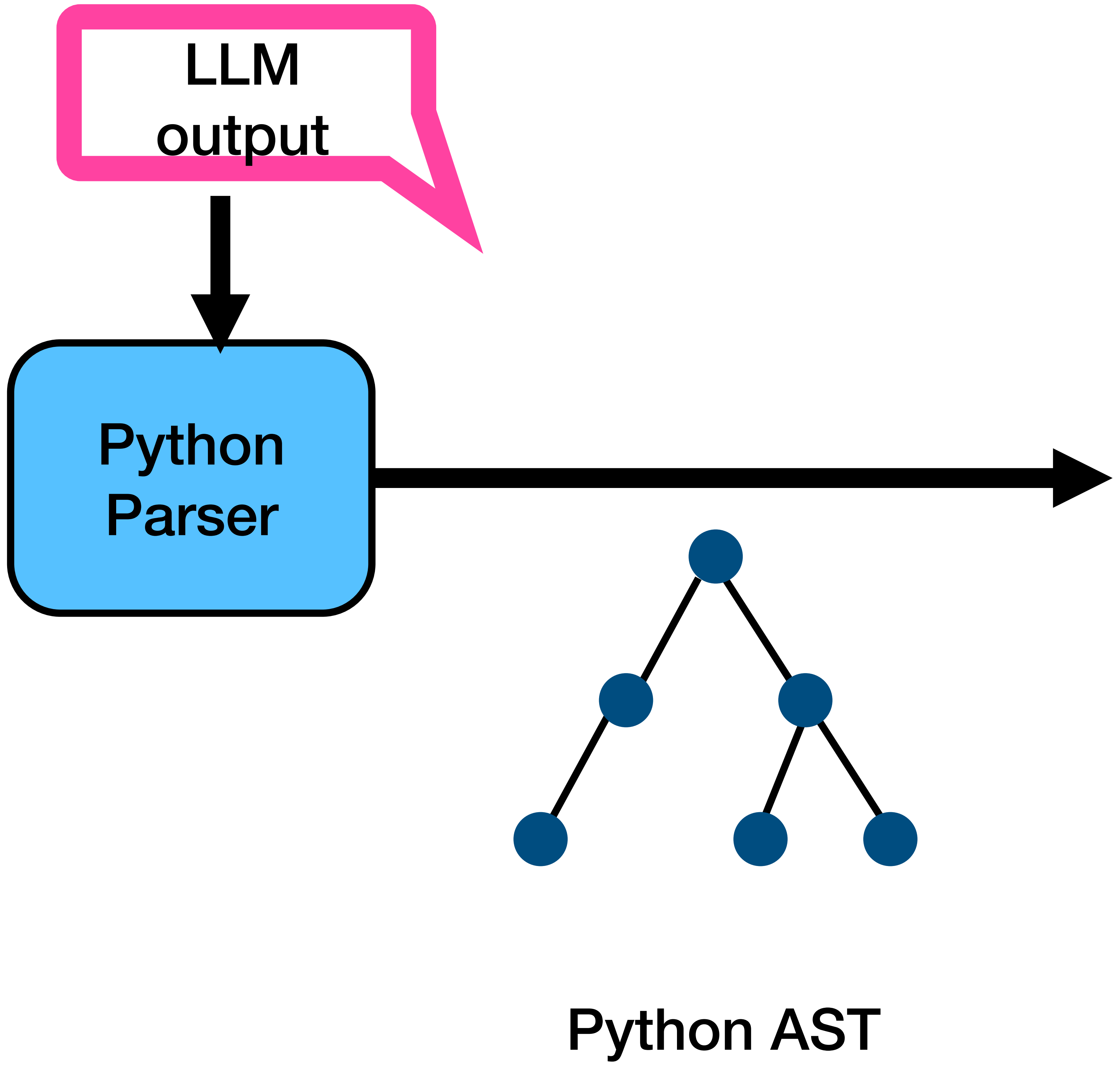
Repair via SMT solver

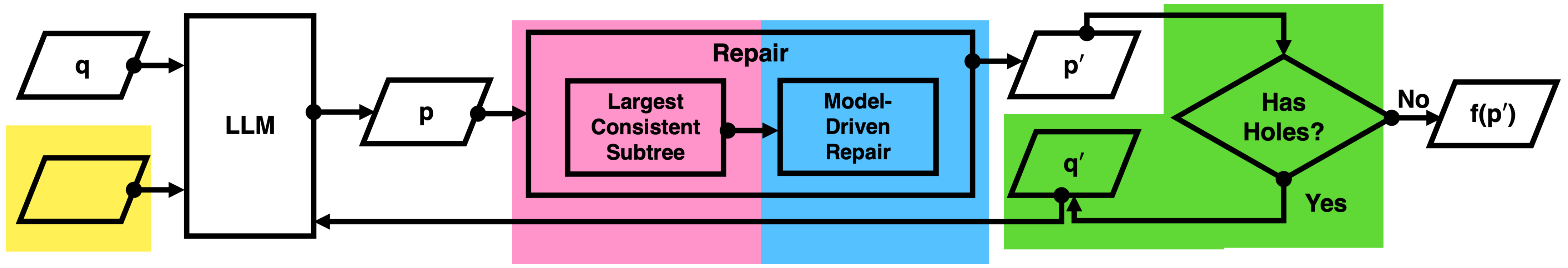
Repair via LLM



Finding the largest subtree

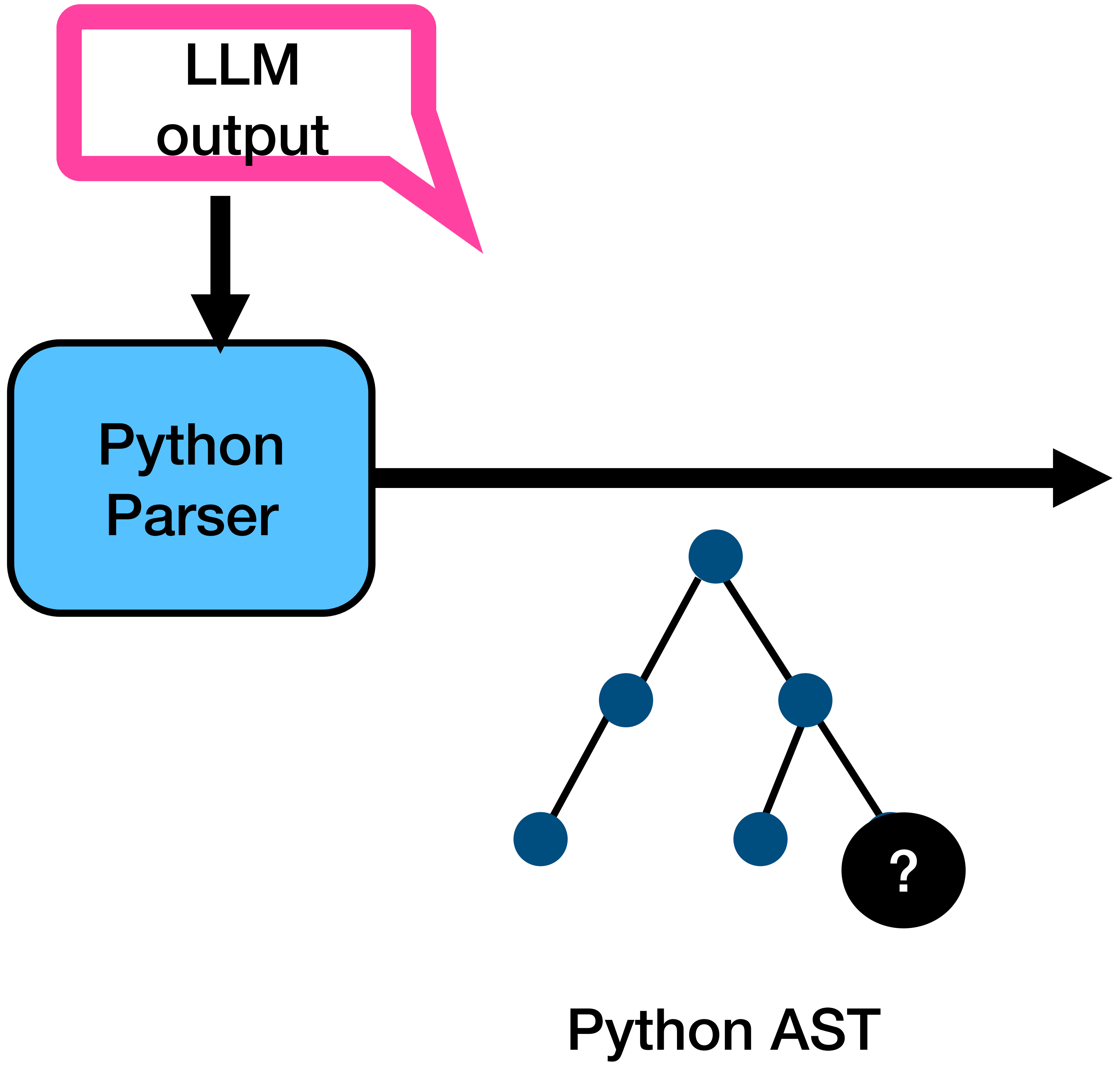
- Given a response, parse it using a parser for Python



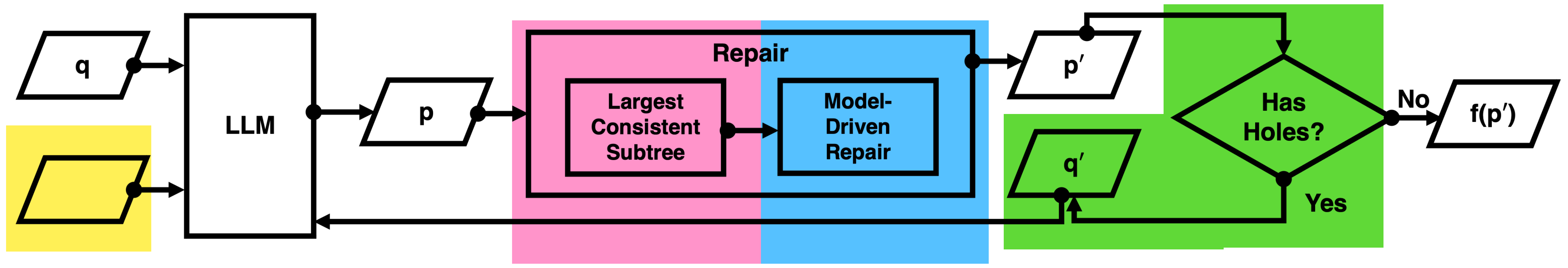


Finding the largest subtree

- Given a response, parse it generously using an error-tolerant parser for Python

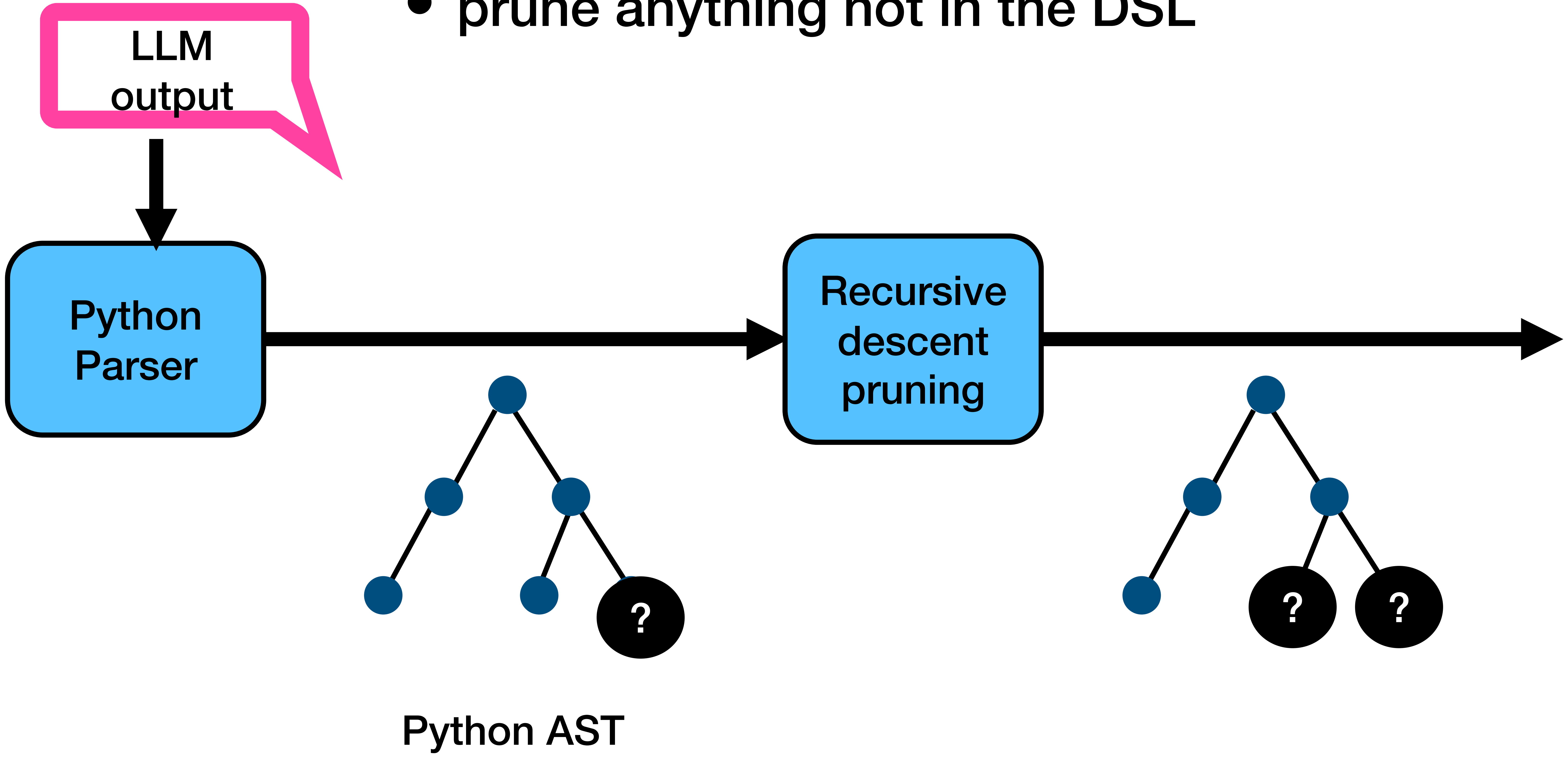


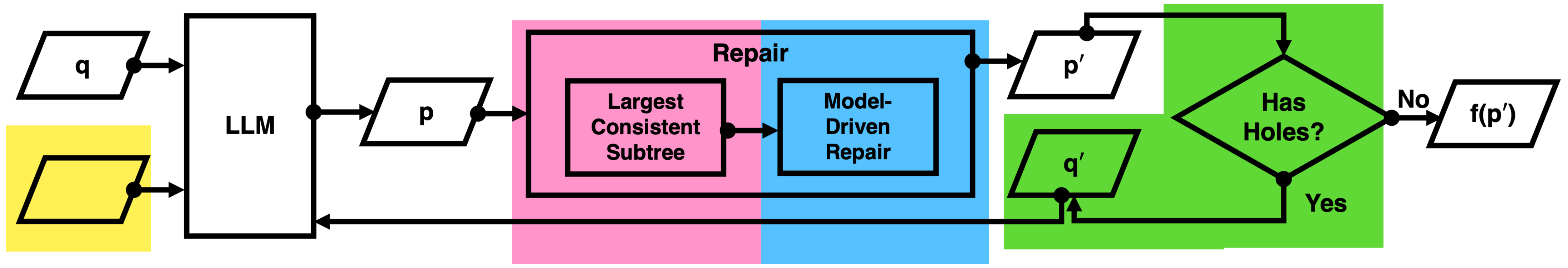
Any error nodes are marked and replaced with holes



Finding the largest subtree

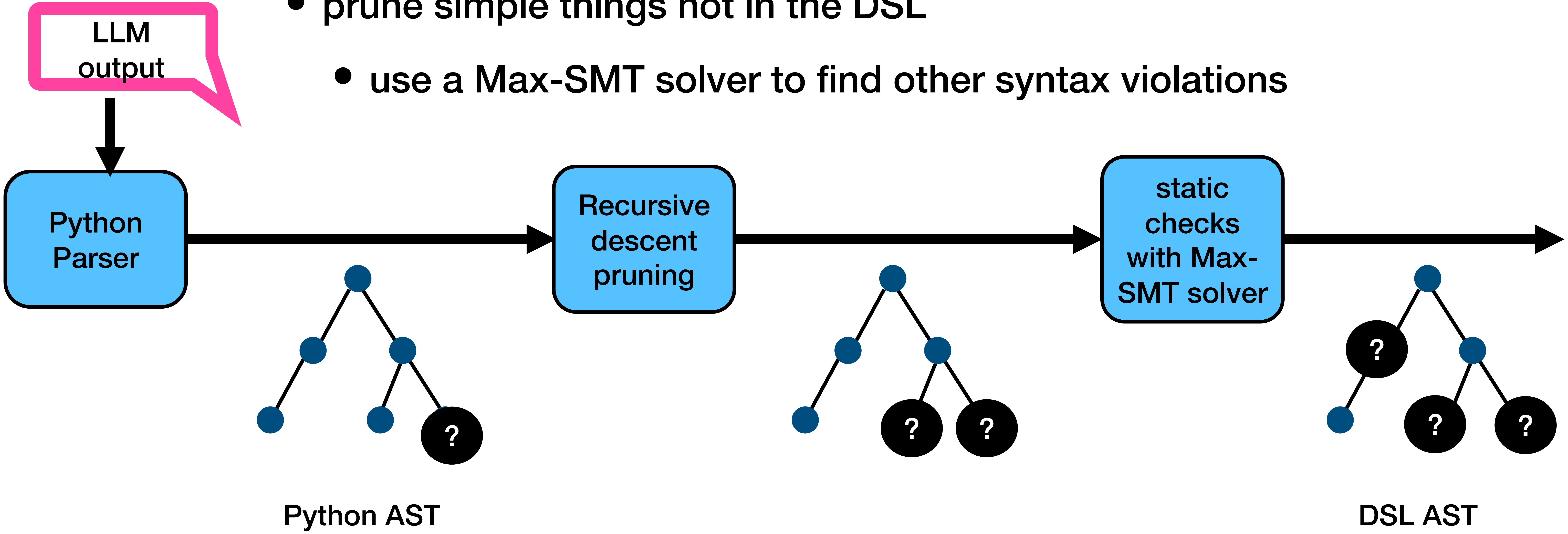
- Given a response, parse it generously using an error-tolerant parser for Python
  - prune anything not in the DSL

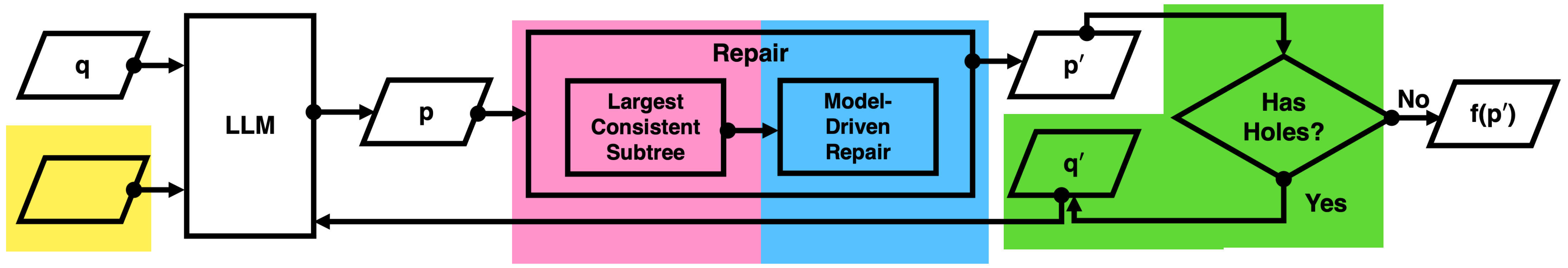




Finding the largest subtree

- Given a response, parse it generously using an error-tolerant parser for Python
  - prune simple things not in the DSL
    - use a Max-SMT solver to find other syntax violations



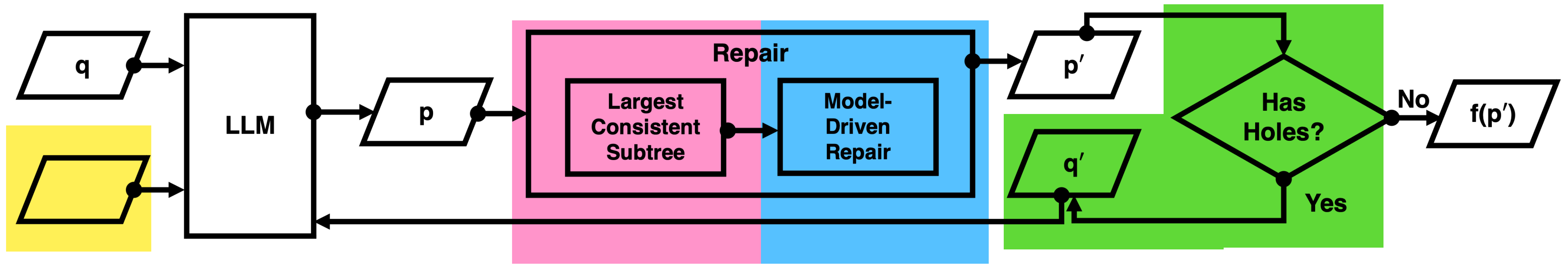


Finding the largest subtree

Max-SMT encoding, inspired by [9]:

- For every static check in the DSL, for every node of the AST, generate a clause
- If all clauses are satisfied, the AST is in the DSL
- Find the maximum set of satisfiable clauses, and replace all other nodes with holes

[9] Finding Minimum Type Error Sources - Pavlinovic et al



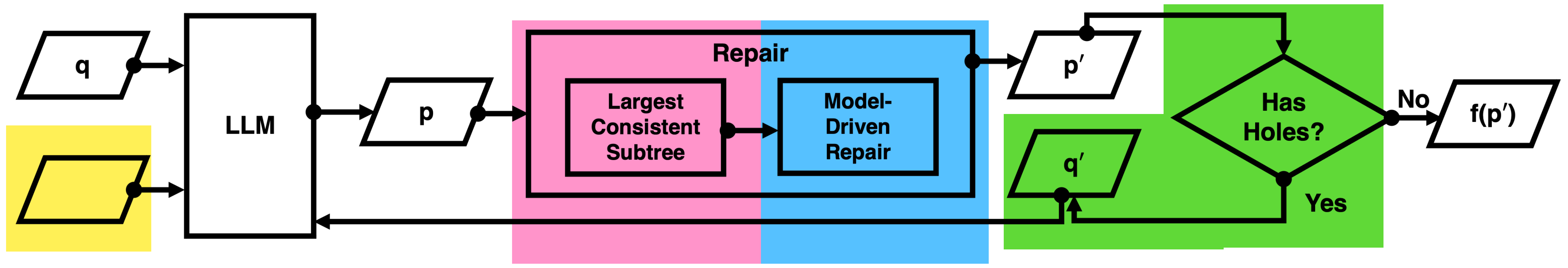
Finding the largest subtree

Max-SMT encoding:

- For every static check in the DSL, for every node of the AST, generate a clause
- If all clauses are satisfied, the AST is in the DSL
- Find the maximum set of satisfiable clauses, and replace all other nodes with holes

```
var x: bv32
x:=0;
```

- x is a bitvector
- 0 is an integer
- x is an integer



Finding the largest subtree

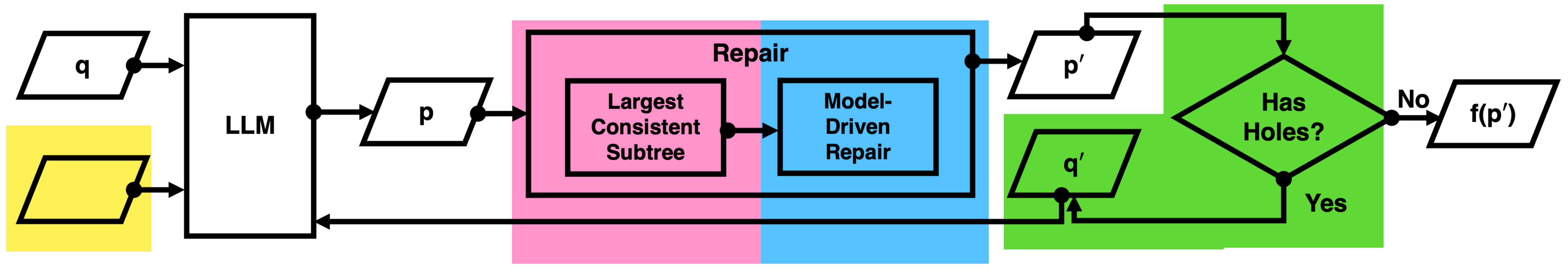
Max-SMT encoding:

- For every static check in the DSL, for every node of the AST, generate a clause
- If all clauses are satisfied, the AST is in the DSL
- Find the maximum set of satisfiable clauses, and replace all other nodes with holes

```
var x: bv32
x:=0;
```

→

- x is a bitvector
- 0 is an integer
- x is an integer



Finding the largest subtree

Max-SMT encoding:

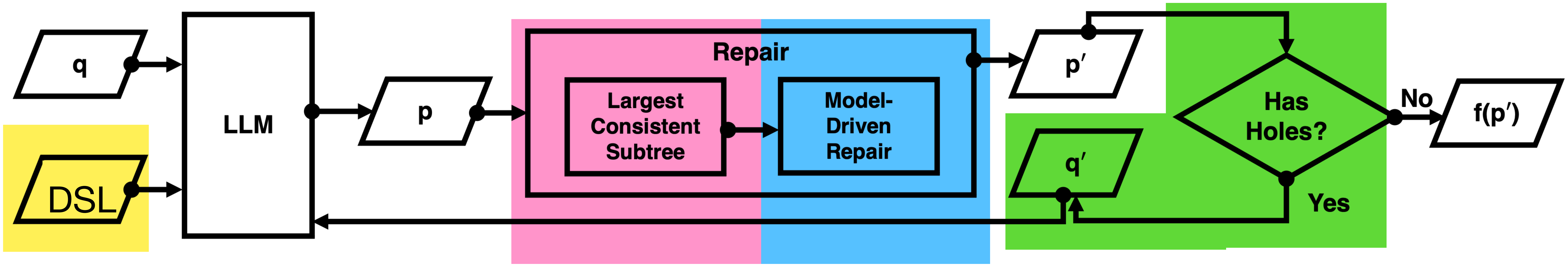
- For every static check in the DSL, for every node of the AST, generate a clause
- If all clauses are satisfied, the AST is in the DSL
- Find the maximum set of satisfiable clauses, and replace all other nodes with holes

```
var x: ??
x:=0;
```

→

- x is a ??
- 0 is an integer
- x is an integer

Overview

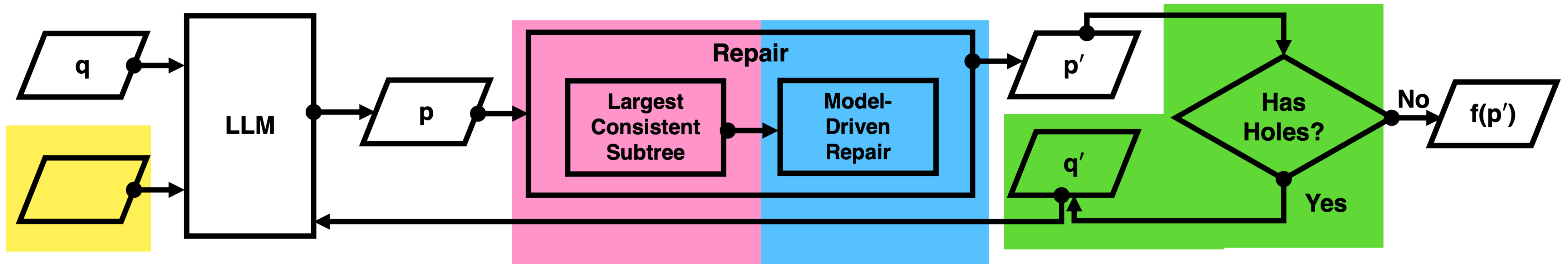


Design a DSL for the LLM

Parse response generously

Repair via SMT solver

Repair via LLM



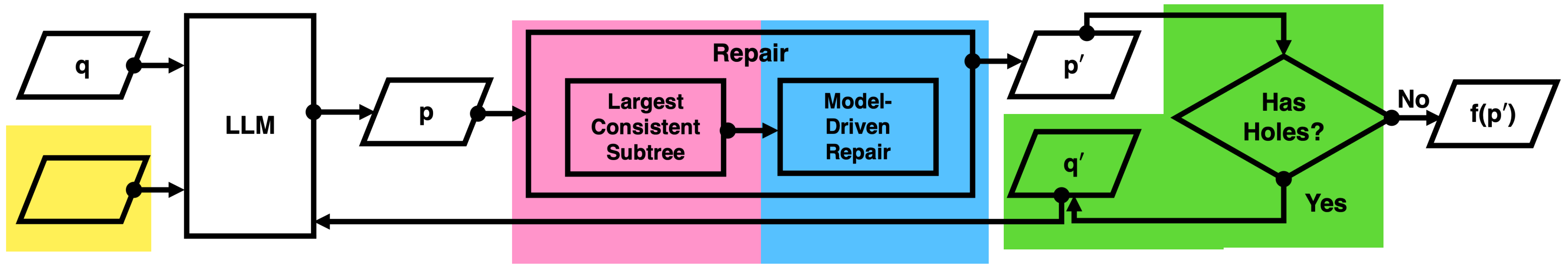
Model-driven repair

Once we have a set of satisfiable clauses, we can use the satisfying model to repair (some of) the program:

```
var x: ??
x:=0;
```

→

- x is a ??
- 0 is an integer
- x is an integer



Model-driven repair

Once we have a set of satisfiable clauses, we can use the satisfying model to repair (some of) the program:

```
var x: ??
x:=0;
```

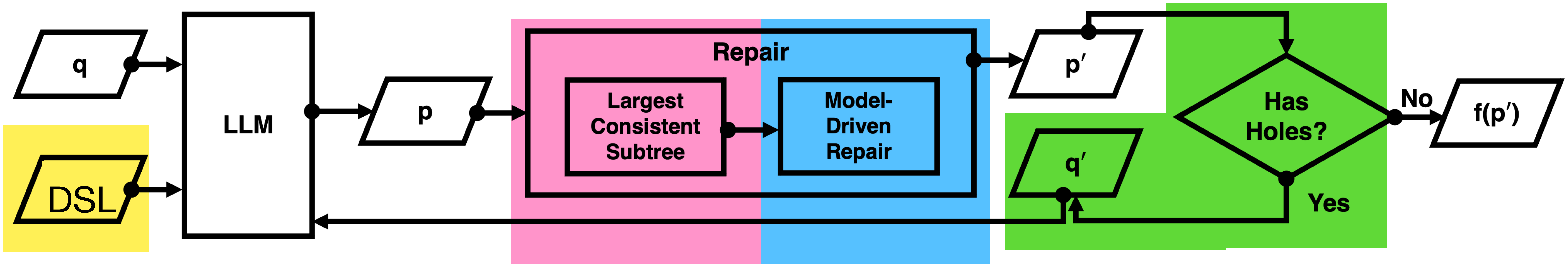
→

- x is a ??
- 0 is an integer
- x is an integer

SAT when ?? is “integer”

If multiple assignments are valid, we use the one the SMT solver suggests.

Overview

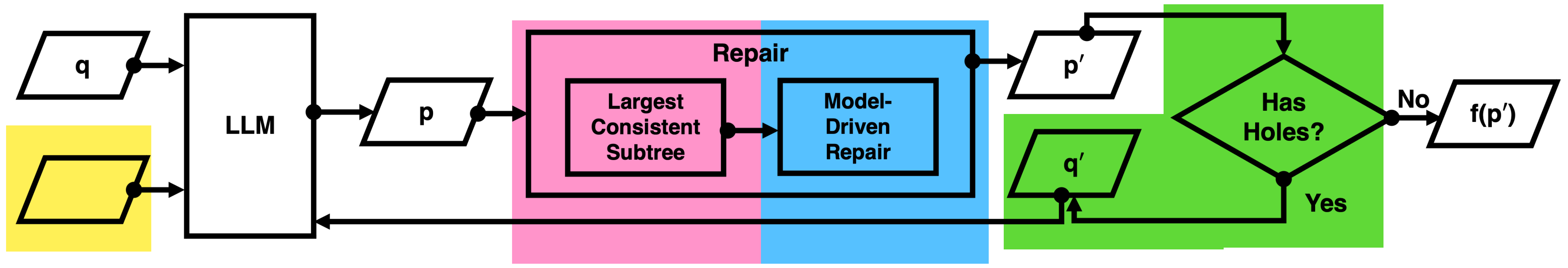


Design a DSL for the LLM

Parse response generously

Repair via SMT solver

Repair via LLM

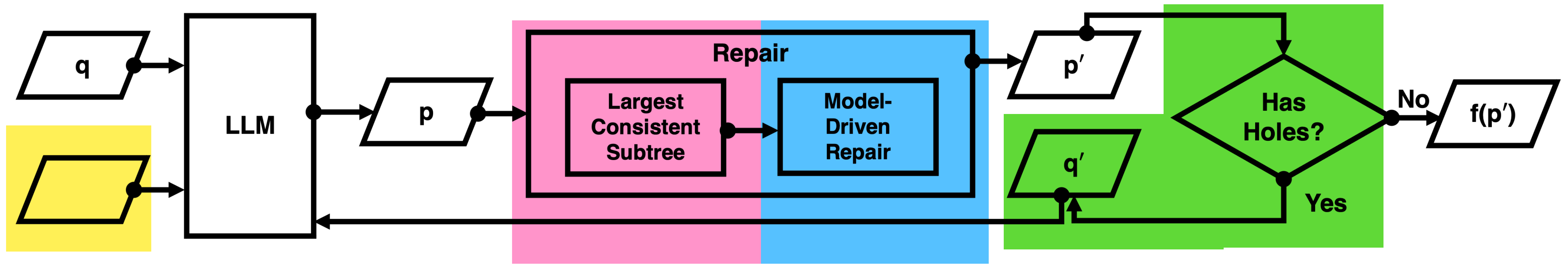


LLM-driven repair

Not everything can be repaired by the Max-SMT solver

```
module main {
    var x: integer;

    next {
        x' = y + 1;
    }

    invariant x_eq_y: x == 1;

    control {
        induction;
        check;
        print_results;
    }
}
```

```
module main {
    var x: integer;

    next {
        x' = ?? + 1;
    }

    invariant x_eq_y: x == 1;

    control {
        induction;
        check;
        print_results;
    }
}
```

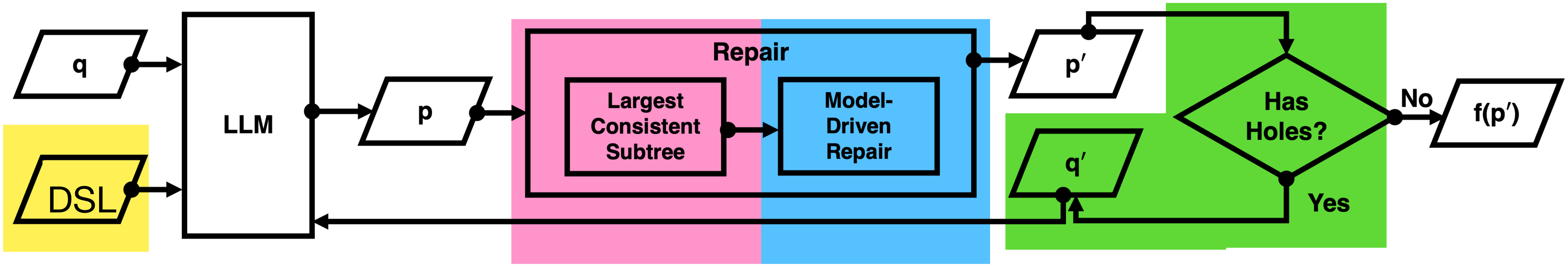


LLM-driven repair

If we can't repair the program from the model, we ask the LLM to repair the holes:

> Fix the following Python code by replacing every occurrence of "??" with the correct code.
>
> [CODE WITH HOLES]
>
> Make sure your code completes the following lines [...]

Overview



Design a DSL for the LLM

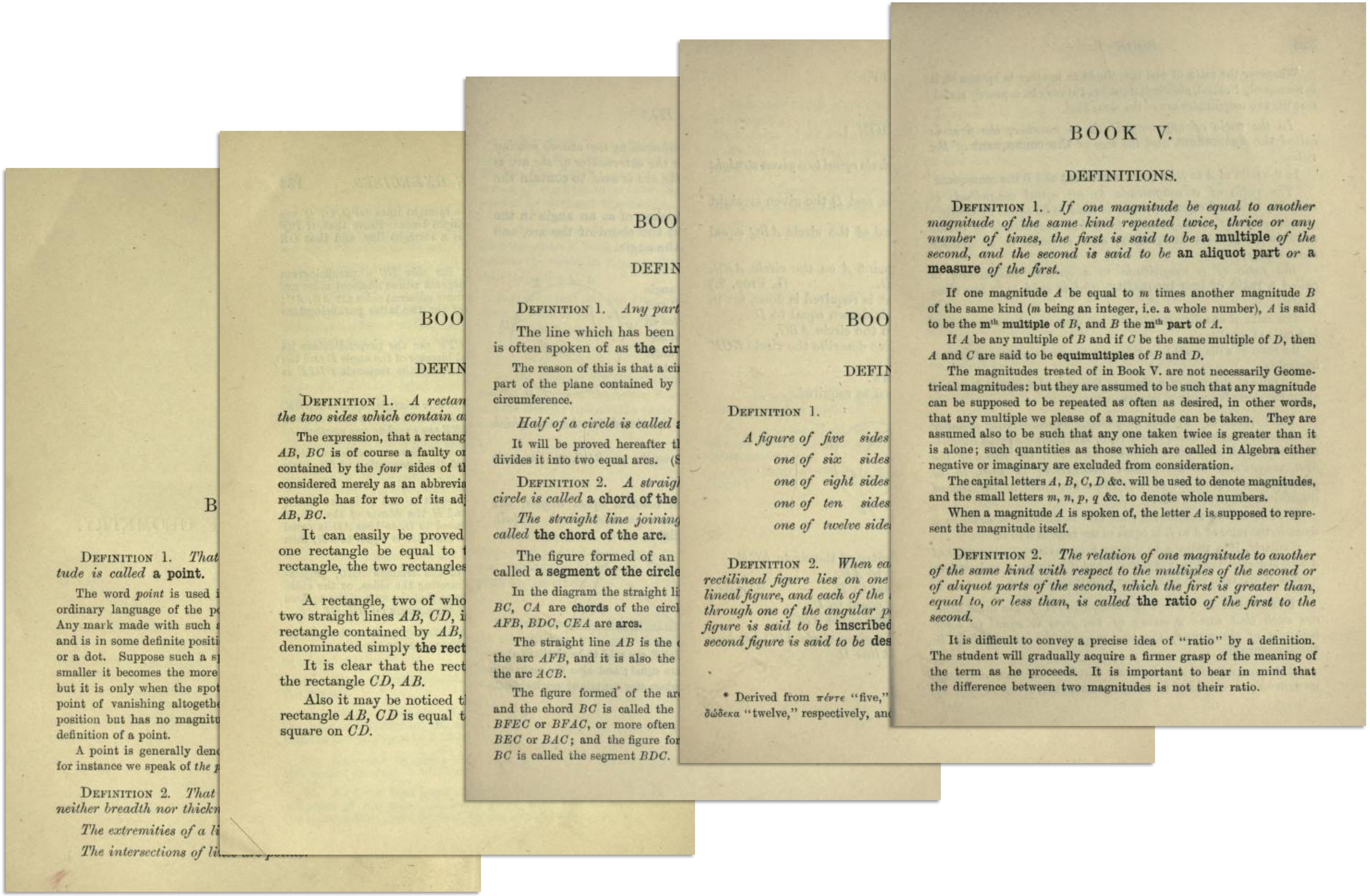
Parse response generously

Repair via SMT solver

Repair via LLM

Loop repeats up to 5 times.

Eudoxus: ghost writing UCLID5



BOOK V.

DEFINITIONS.

DEFINITION 1. *If one magnitude be equal to another magnitude of the same kind repeated twice, thrice or any number of times, the first is said to be* **a multiple** *of the second, and the second is said to be* **an aliquot part** *or* **a measure** *of the first.*

If one magnitude A be equal to m times another magnitude B of the same kind (m being an integer, i.e. a whole number), A is said to be the **mth multiple** of B, and B the **mth part** of A.

If A be any multiple of B and if C be the same multiple of D, then A and C are said to be **equimultiples** of B and D.

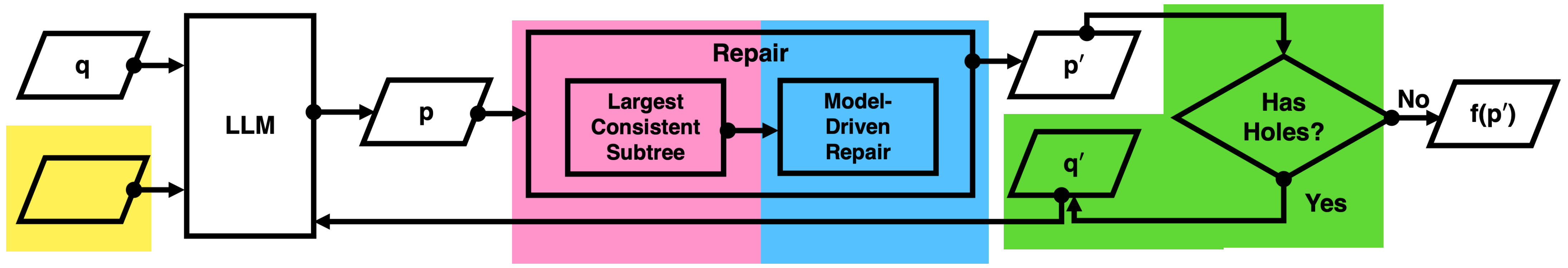
The magnitudes treated of in Book V. are not necessarily Geometrical magnitudes: but they are assumed to be such that any magnitude can be supposed to be repeated as often as desired, in other words, that any multiple we please of a magnitude can be taken. They are assumed also to be such that any one taken twice is greater than it is alone; such quantities as those which are called in Algebra either negative or imaginary are excluded from consideration.

The capital letters A, B, C, D &c. will be used to denote magnitudes, and the small letters m, n, p, q &c. to denote whole numbers.

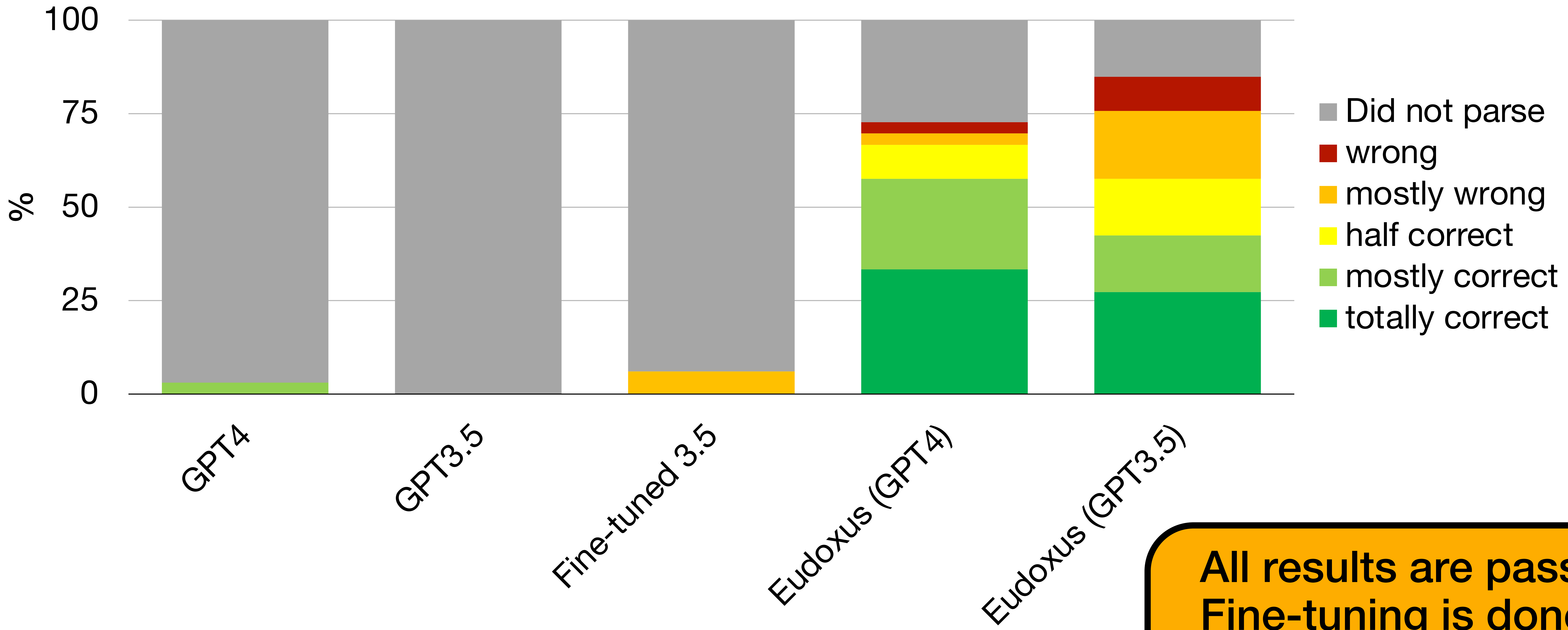
When a magnitude A is spoken of, the letter A is supposed to represent the magnitude itself.

DEFINITION 2. *The relation of one magnitude to another of the same kind with respect to the multiples of the second or of aliquot parts of the second, which the first is greater than, equal to, or less than, is called* **the ratio** *of the first to the second.*

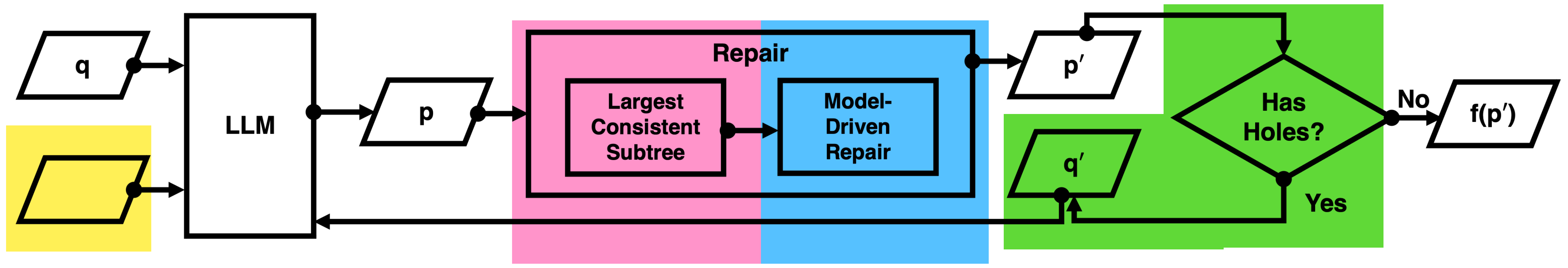
It is difficult to convey a precise idea of "ratio" by a definition. The student will gradually acquire a firmer grasp of the meaning of the term as he proceeds. It is important to bear in mind that the difference between two magnitudes is not their ratio.



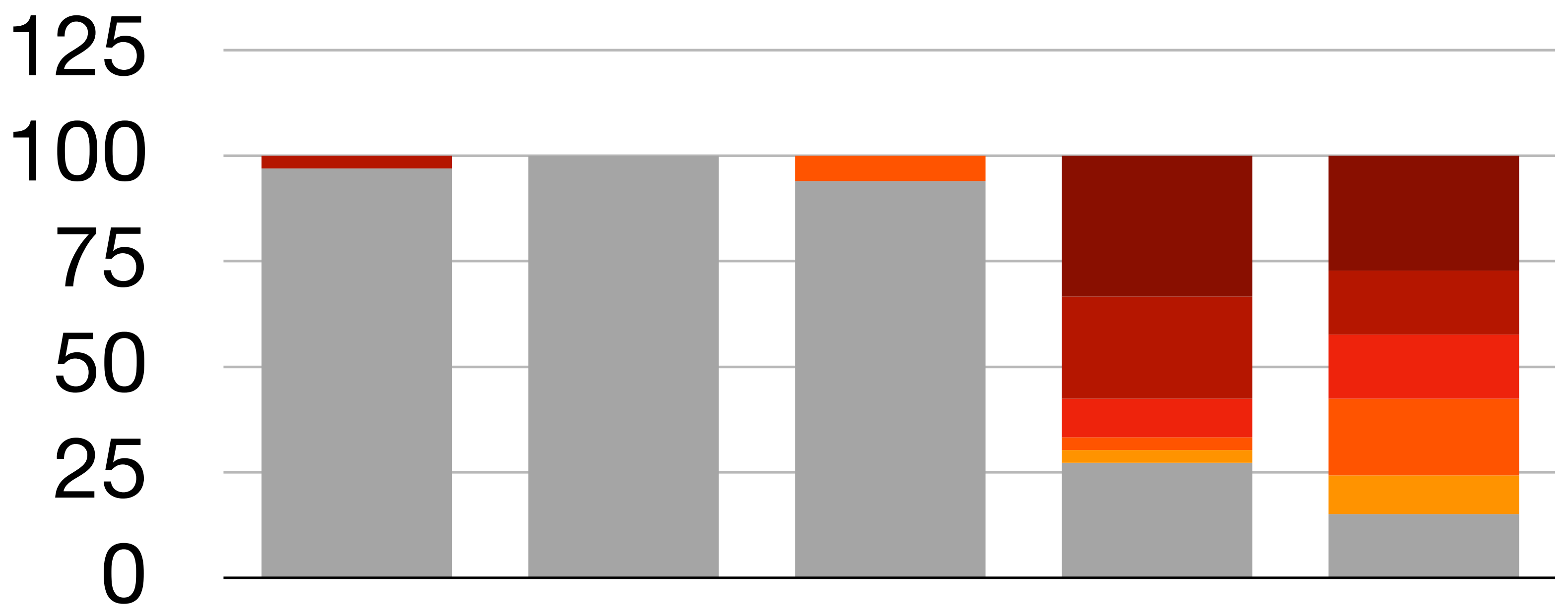
Results



All results are pass@1
Fine-tuning is done using all public UCLID5 regression tests.



Conclusion (part II)



When syntactic correctness is an issue, don't prompt or fine-tune to force the LLM to learn the rules you want it to.

Instead, use PL and formal techniques to meet the LLM in the middle

Related work

Giving the LLM feedback via compiler errors [10]

- Needs good compiler errors
- Not very effective for Dafny anyway[11]

Constrained Decoding [12]

- Limited to checks that you can encode in a grammar
- Works well if the LLM is *reasonably close* to the grammar you want?

[10] Fixing Rust Compilation Errors using LLMs - Deligiannis et al
[11] DafnyBench: A Benchmark for Formal Software Verification – Loughridge et al
[12] Efficient Guided Generation for Large Language Models – Willard and Louf

Conclusions

Not dead yet!

Conclusions

- Semantic and Syntactic correctness are still challenges for LLMs
  - especially in low resource languages and problem domains
- Formal methods and enumerative techniques might just be the answer to this!

Conclusions

- Semantic and Syntactic correctness are still challenges for LLMs
  - especially in low resource languages and problem domains
- Formal methods and enumerative techniques might just be the answer to this!

Guiding enumerative synthesis @CAV:

talk@CAV, Saturday 27th 4pm
contact Yixuan.Li.cs@ed.ac.uk

Eudoxus @CAV:

contact fmora@Berkeley.edu