Distance-Guided Search in Program Synthesis with Imperfect LLM Solutions

Hangyeol Cho, Jaehyung Lee, Woosuk Lee



Hanyang University, Republic of Korea

@ICSE'26

Program Synthesis

- Automatically generate programs from a specification

What the program looks like

Specification

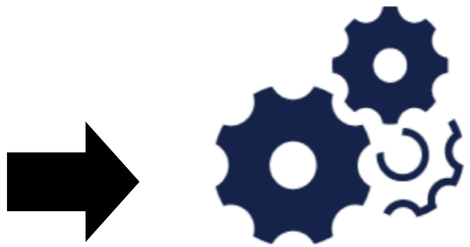
- Grammar:

  $x,\ 0,\ 1,\ 2,$

  $\square_1 + \square_2,\ \square_1 \times \square_2$

- Input-output Examples:

  $1 \to 4,\quad 2 \to 6$



Synthesizer

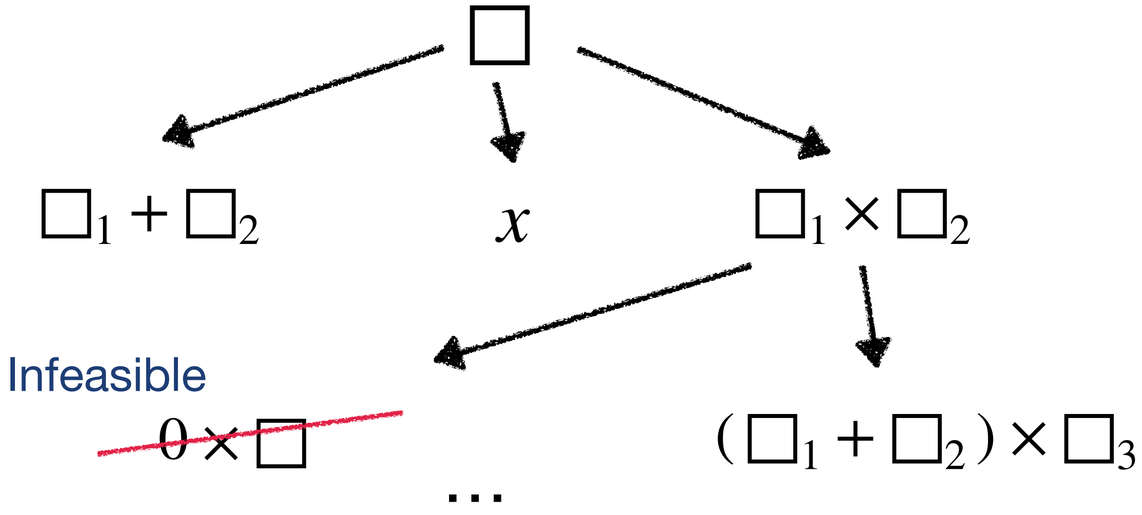
$(x + 1) \times 2$

What the program should do

Search-based Synthesis

Function f x = □ (spec: $1 \to 4$, $2 \to 6$)

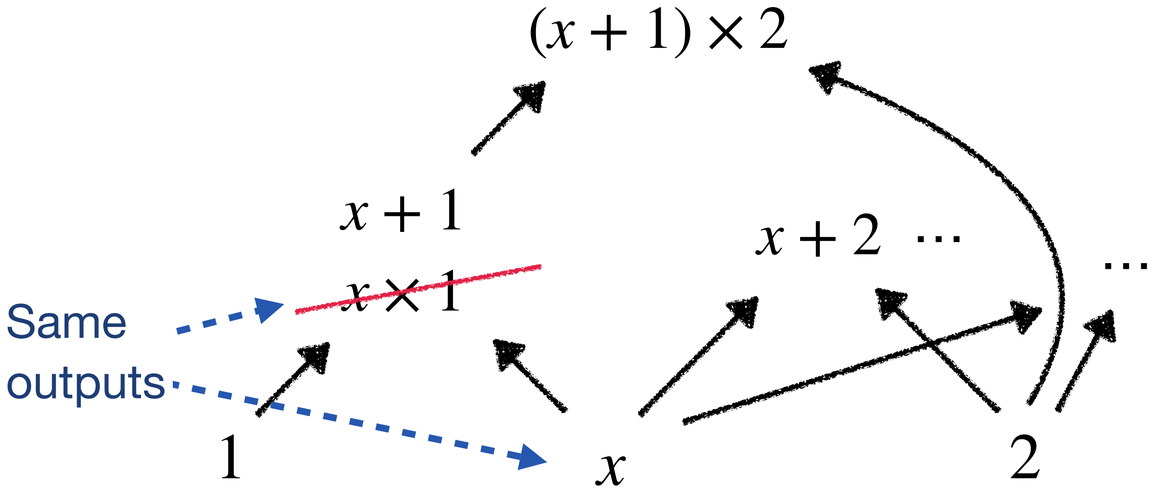
Top-down



Starts from empty program, fills in holes

Prune infeasible partial programs
(i.e., keep only candidates that can satisfy the spec)

Bottom-up



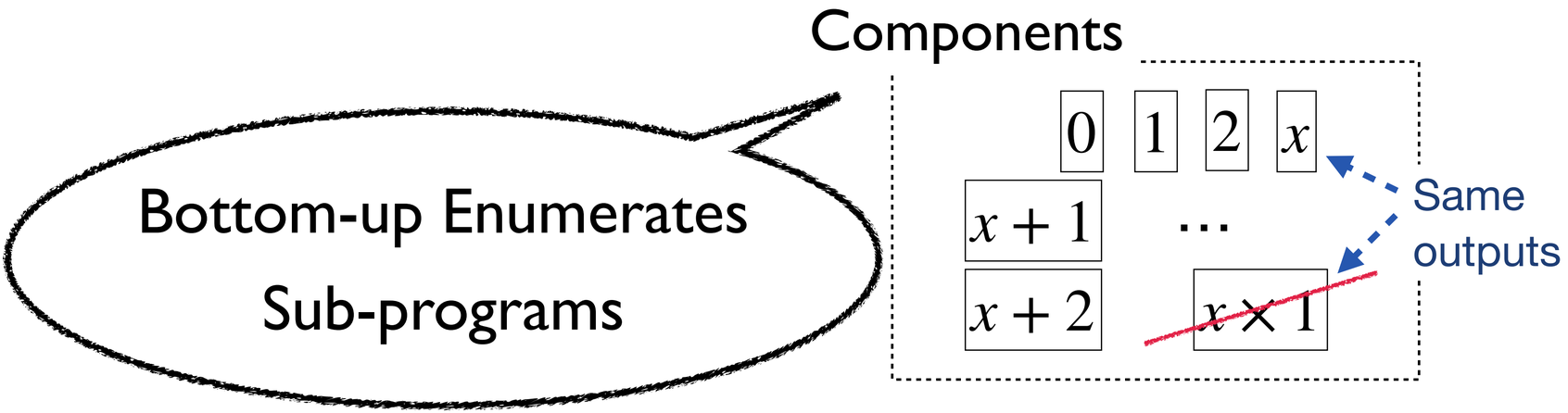
Builds larger programs from smaller ones

Prune redundant subexpressions
(i.e., keep only one representative per same output)

Search-based Synthesis : Bidirectional

`Function f` x `= □ (spec:` $1 \rightarrow 4, \quad 2 \rightarrow 6$`)`

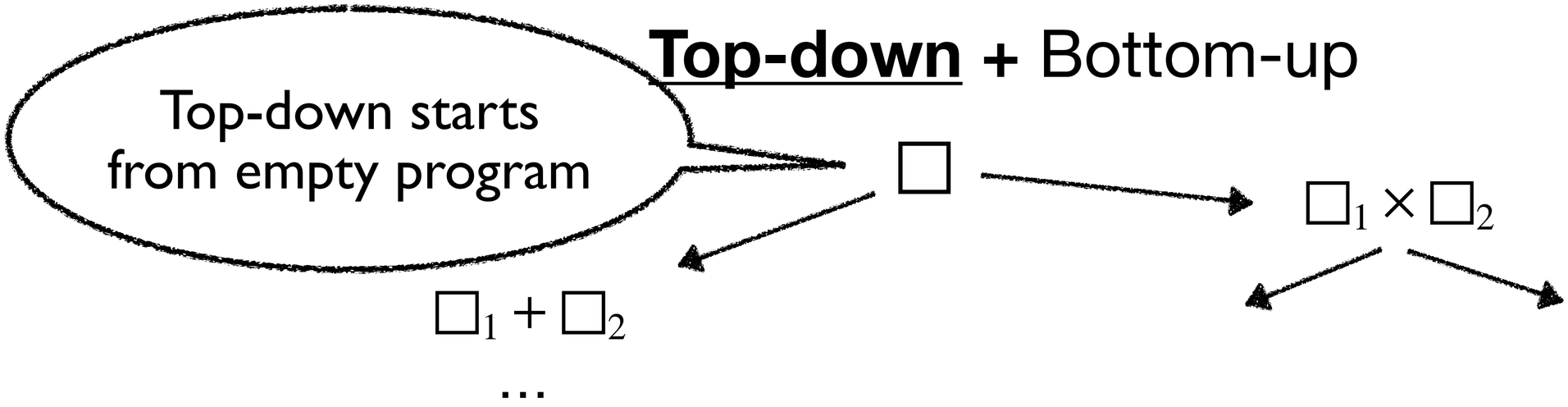
Top-down + **Bottom-up**



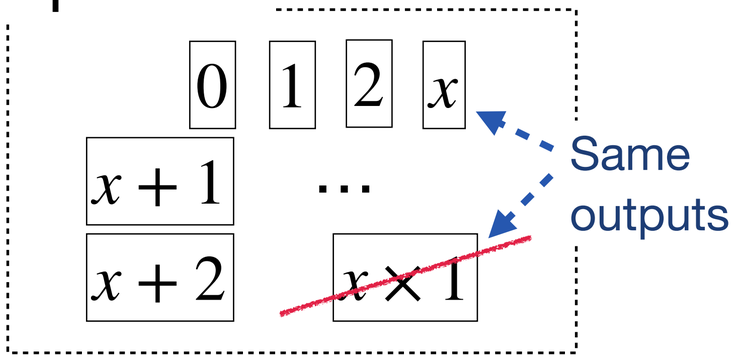
Search-based Synthesis : Bidirectional

Function f x = □ (spec: $1 \to 4,\quad 2 \to 6$)

Top-down + Bottom-up



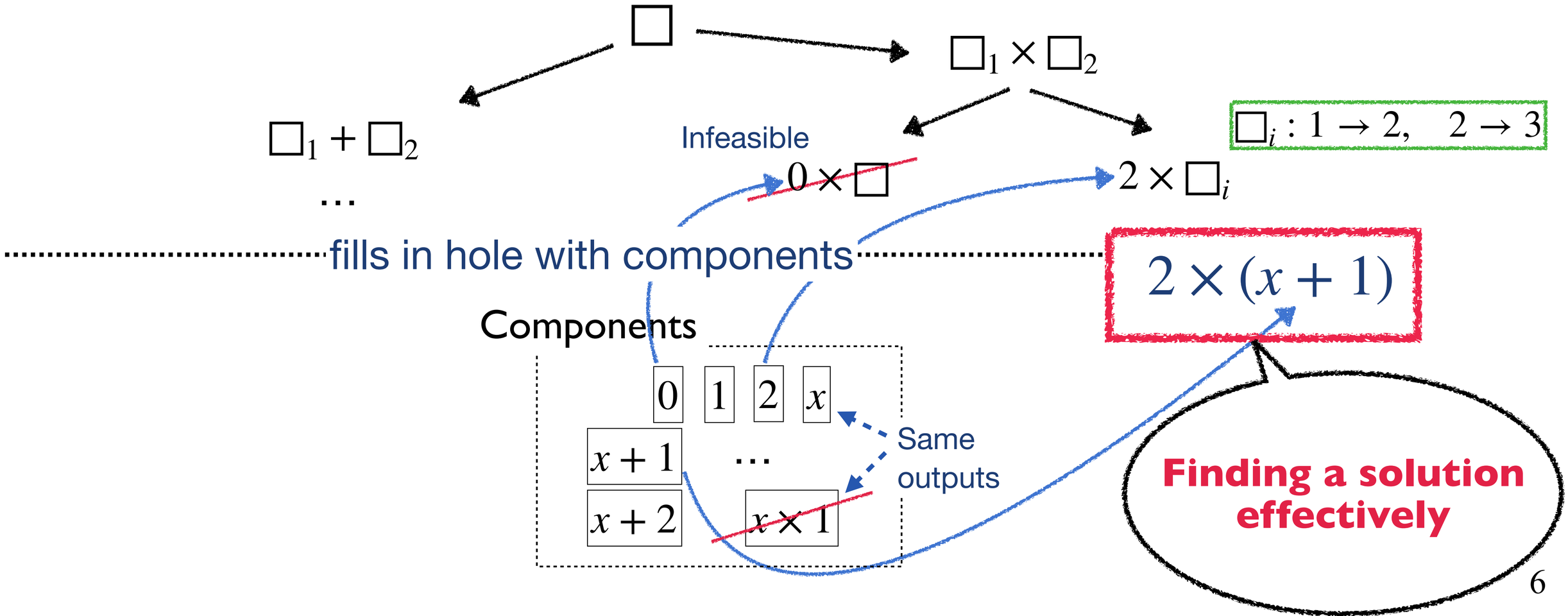
Components



Search-based Synthesis : Bidirectional

Function f x = □ (spec: $1 \to 4, \quad 2 \to 6$)

Top-down + Bottom-up



Recursion Synthesis from I/O examples w/ LLM #1

- GPT-4o-mini generated a program, but the solution was incorrect.

```
let rec f n t ->
      match t with
      | Leaf -> True
      | Node(x, left, right) ->
        match (compare n x) with
        | EQ -> False
        | GT -> (f n right)
        | LT -> (f n left)
```

LLM solution is wrong

```
let rec f n t ->
      match t with
      | Leaf -> True
      | Node(x, left, right) ->
        match (compare n x) with
        | EQ -> False
        | GT -> (f n left)∧(f n right)
        | LT -> (f n left)∧(f n right)
```

Desired Solution

Recursion Synthesis from I/O examples w/ LLM #2

- GPT-4o-mini generated a program, but the solution was incorrect.

```
let rec f x y ->
      match x with
      | [] ->
         match y with
         | [] -> 0
         | n2 :: rest2 -> n2
      | n1 :: rest1 ->
         match y with
         | [] -> n1
         | n2 :: rest2 ->
             (compare n1 n2) + (f rest1 rest2)
```

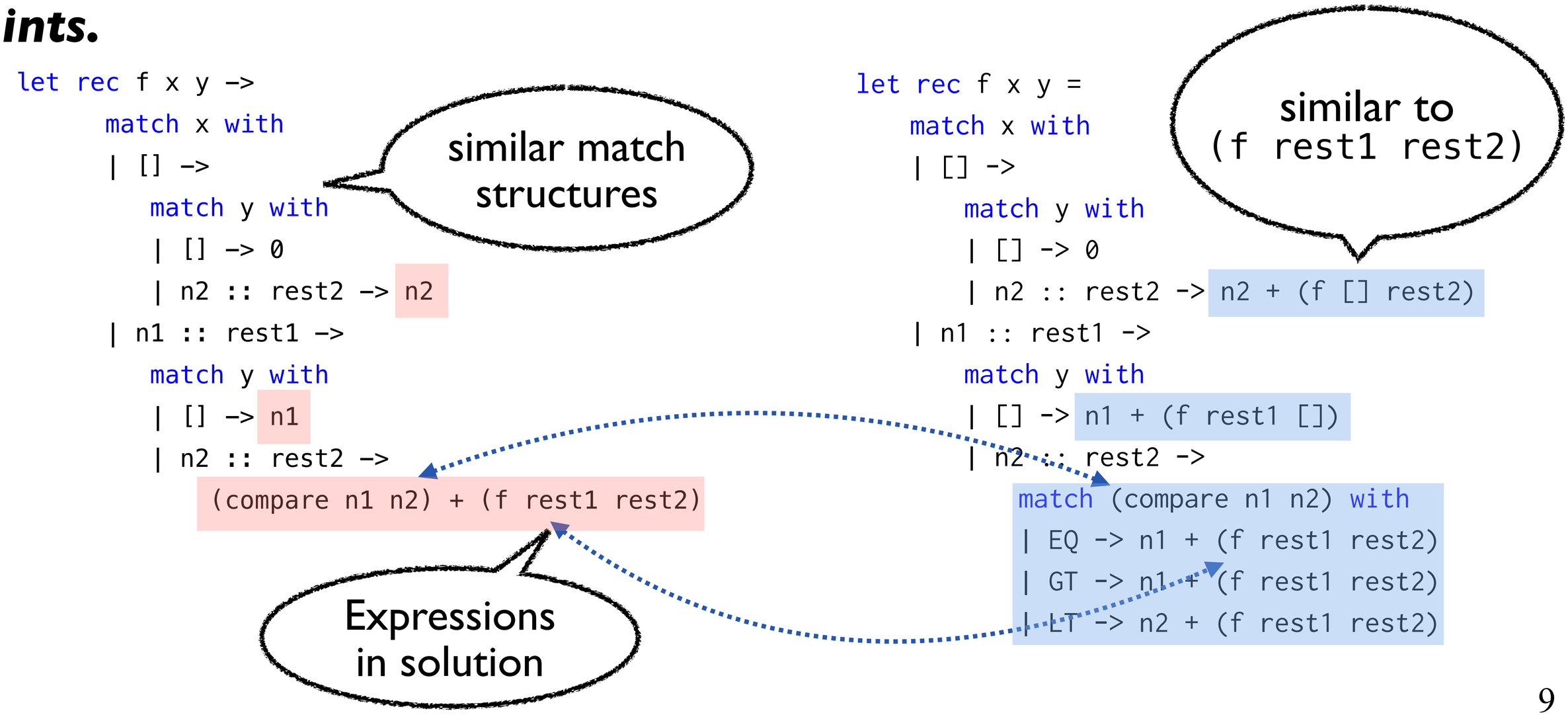
LLM solution is wrong

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> n2 + (f [] rest2)
    | n1 :: rest1 ->
        match y with
        | [] -> n1 + (f rest1 [])
        | n2 :: rest2 ->
            match (compare n1 n2) with
            | EQ -> n1 + (f rest1 rest2)
            | GT -> n1 + (f rest1 rest2)
            | LT -> n2 + (f rest1 rest2)
```

Desired Solution

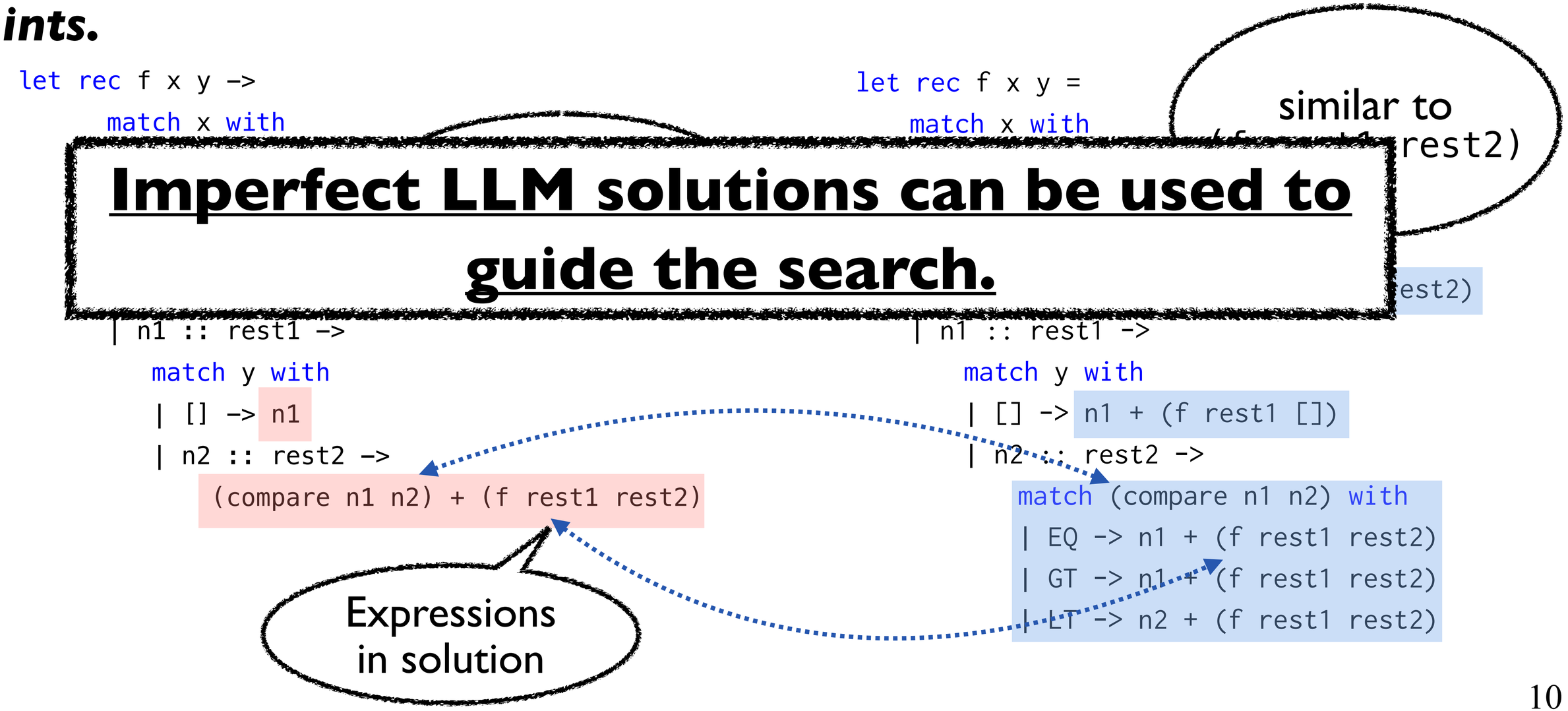
Observation

- Even though LLM solutions are incorrect, they often contain ***useful structural hints.***



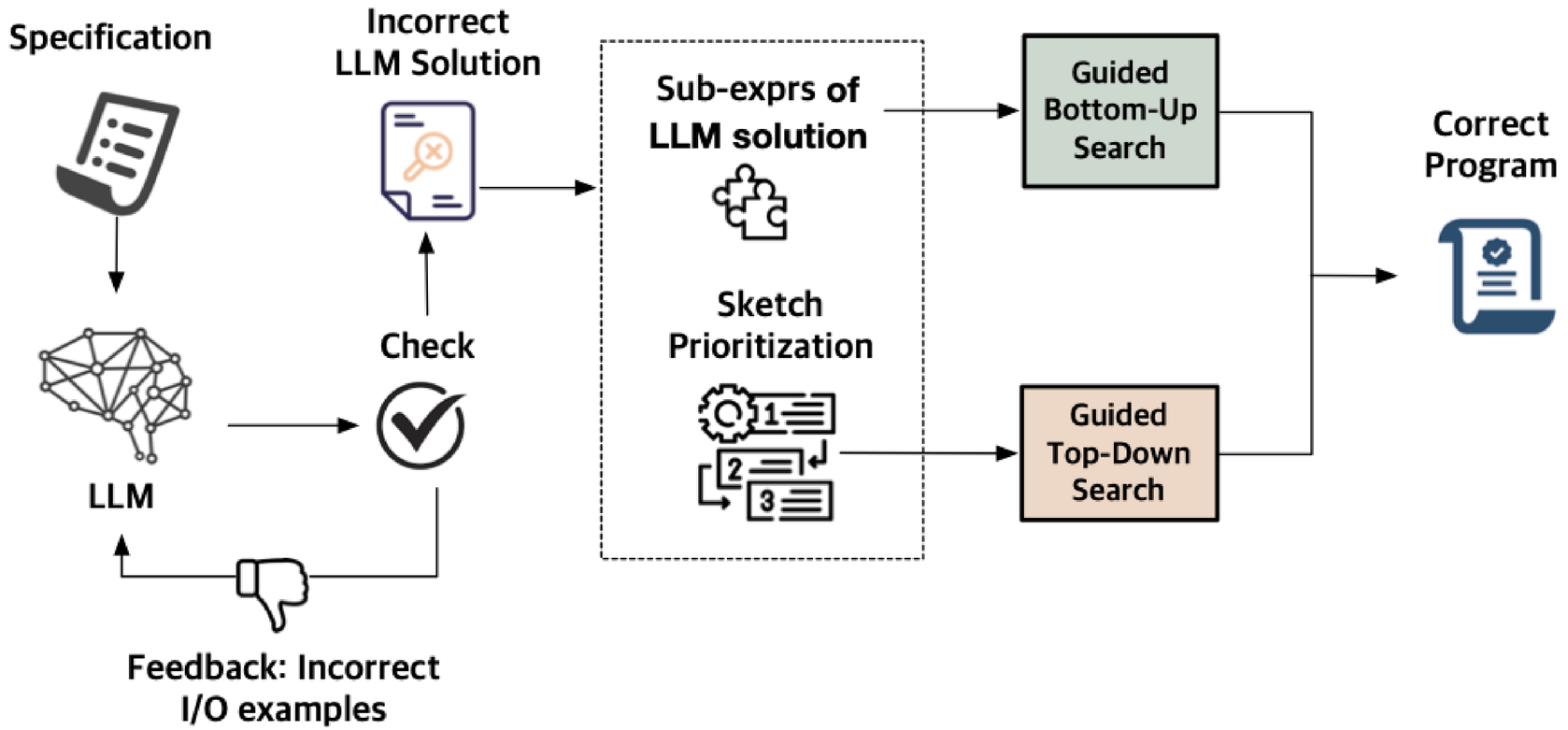
Observation

- Even though LLM solutions are incorrect, they often contain ***useful structural hints.***

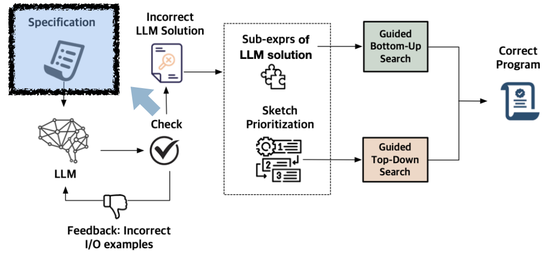


Key Idea

- Applying LLM guidance to each direction in bidirectional search
  - Top-down guidance prioritizes candidates close to the LLM solution using an effective distance metric
  - Bottom-up guidance enumerates components close to LLM subexpressions.



Problem Example



- Target domain: ***recursive functional programs*** from input-output examples
- Generate a function that takes two lists of integers and returns the sum of the larger elements at each position.
- Spec: `f :` int list → int list → int

  `f [] [0]` → `0`

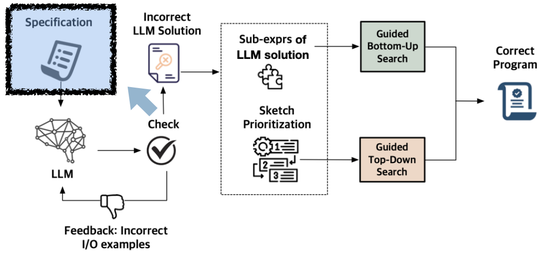
  `f [1, 2, 3] [0, 1]` → `6`

  using

  (+) , compare

  …

Problem Example (1/3)

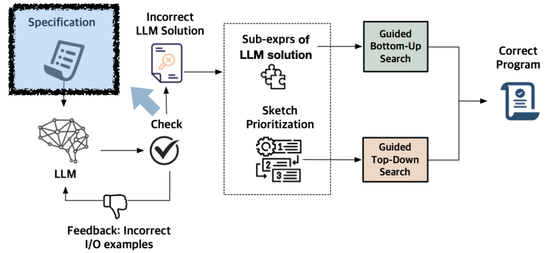


- Generate a function that takes two lists of integers and returns the sum of the larger element at each corresponding index.

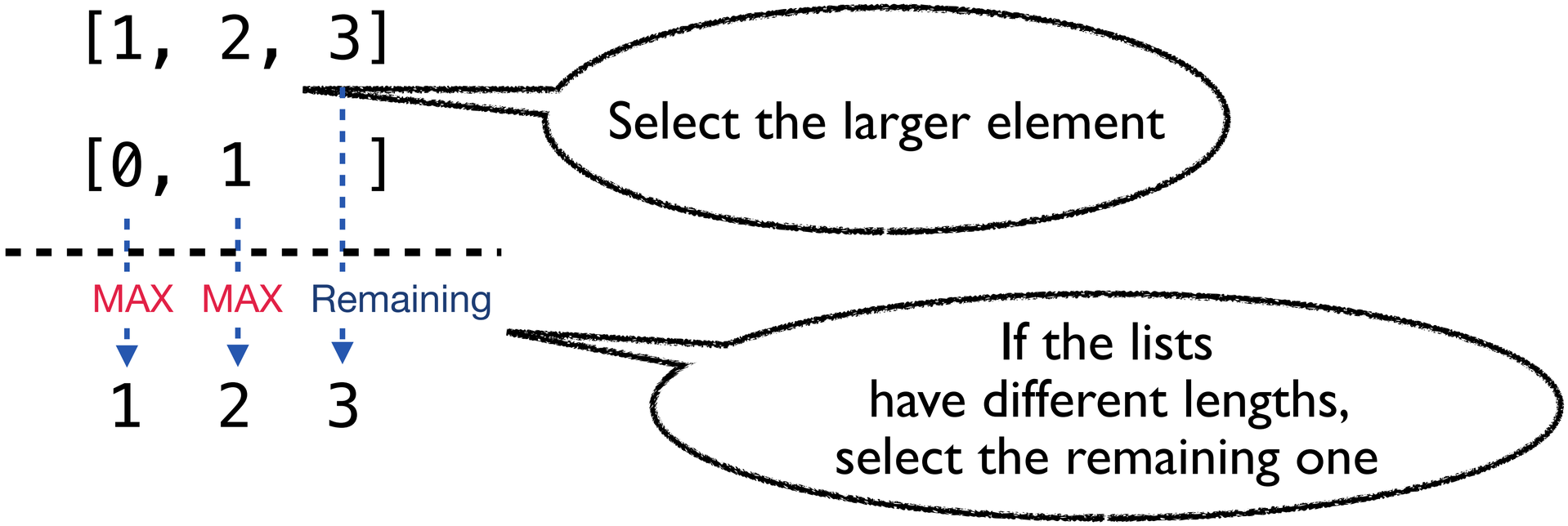
```
[1, 2, 3]

[0, 1   ]
```

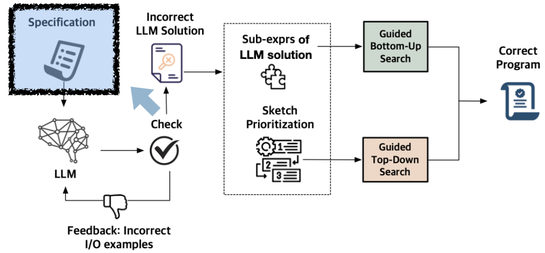
Problem Example (2/3)



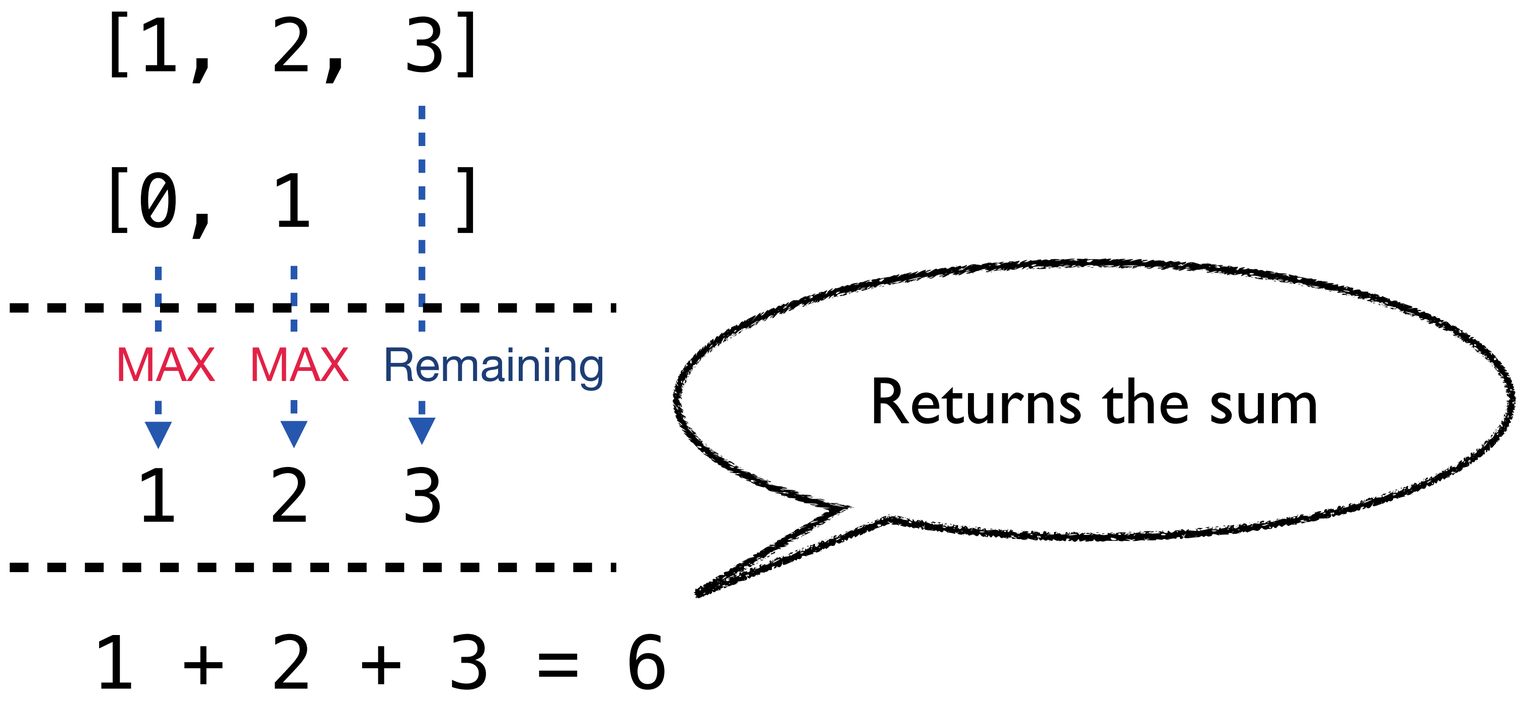
- Generate a function that takes two lists of integers and <u>returns the sum of the larger element</u> at each corresponding index.



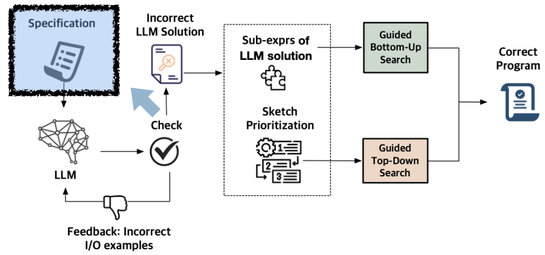
Problem Example (3/3)



- Generate a function that takes two lists of integers and returns the sum of the larger element at each corresponding index.



Problem Example



- Generate a function that takes two lists of integers and returns the sum of the larger element at each corresponding index.

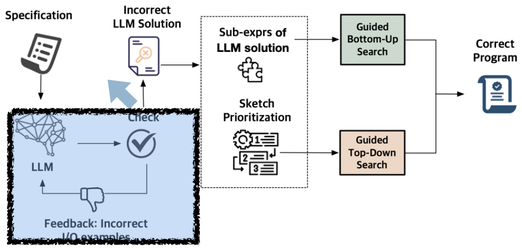
SOTA* synthesizers fail to synthesize within 2 minutes.

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> n2 + (f [] rest2)
    | n1 :: rest1 ->
        match y with
        | [] -> n1 + (f rest1 [])
        | n2 :: rest2 ->
            match (compare n1 n2) with
            | EQ -> n1 + (f rest1 rest2)
            | GT -> n1 + (f rest1 rest2)
            | LT -> n2 + (f rest1 rest2)
```

Complex even for SOTA

*Trio - Lee et al. 2023
*Burst - Miltner et al. 2022
*Smyth - Lubin et al. 2022

Problem Example

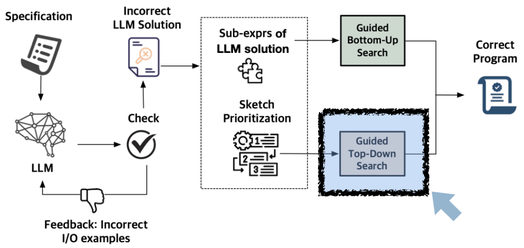


- Generate a function that takes two lists of integers and returns the sum of the larger element at each corresponding index.

```
1     let rec f x y =
2         match x with
3         | [] ->
4             match y with
5             | [] -> 0
6             | n2 :: rest2 -> n2
7         | n1 :: rest1 ->
8             match y with
9             | [] -> n1
10            | n2 :: rest2 ->
11                (compare n1 n2) + (f rest1 rest2)
12
13
14
```

LLM returns an incorrect result.

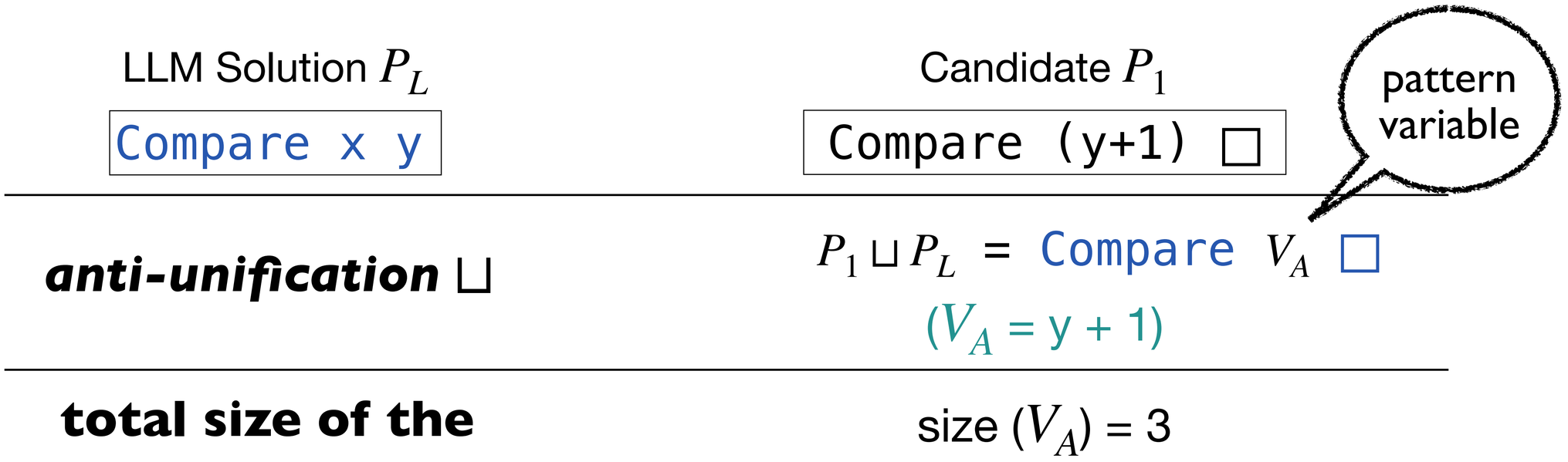
Top-down Guidance



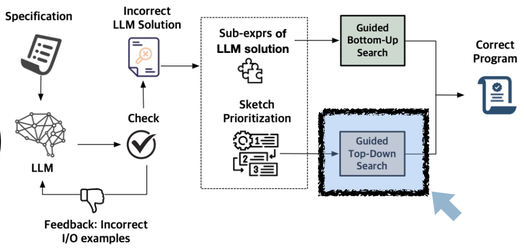
- **Prioritize** candidates **close to the LLM** solution
- Use *anti-unification* to compute structural distance
  - The distance using anti-unification can be computed in $O(n)$ time.
  - More efficient than string or tree edit distance ($\geq O(n^2)$)

Distance Metric for Guiding Search

- Compute distance between a candidate program P and the LLM solution P_L via ***anti-unification*** — replace each mismatched part with pattern variables
- Distance = total size of the mismatched parts
- Holes can be filled with anything, so they are not counted in the distance.



Top-down Guidance w/ Example (1/3)




```
match x with
| [] ->
    match y with
    | [] -> 0
    | n2 :: rest2 -> n2
| n1 :: rest1 ->
    match y with
    | [] -> n1
    | n2 :: rest2 ->
          (compare n1 n2)
          + (f rest1 rest2)
```

LLM Solution P_L

```
let rec f x y = □
```

```
Compare □1 □2
```

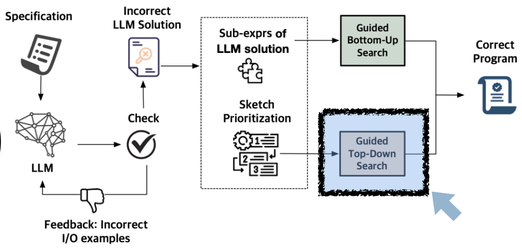
Candidate P_1

```
match x with
| [] -> □1
| n1 :: rest1 -> □2
```

Candidate P_2

Start with an empty program and expand it by one-step derivation

Top-down Guidance w/ Example (2/3)



Top-level structure of P_L

```
match x with
| [] -> ...
| n1 :: rest1 -> ...
```

```
match x w
| [] ->
    match y with
    | [] -> 0
    | n2 :: rest2 -> n2
| n1 :: rest1 ->
    match y with
    | [] -> n1
    | n2 :: rest2 ->
        (compare n1 n2)
        + (f rest1 rest2)
```

LLM Solution P_L

```
let rec f x y = □
```

```
Compare □1 □2
```

Candidate P_1

```
match x with
| [] -> □1
| n1 :: rest1 -> □2
```

Candidate P_2

anti-unification ␣

(V_B = compare $\square_1$ $\square_2$)

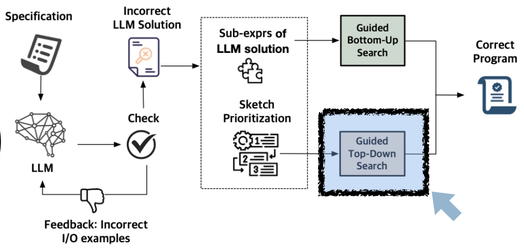
No pattern variable

Compute distance

size (V_B) = 1

No pattern variable = 0

Top-down Guidance w/ Example (3/3)




```
match x with
| [] ->
    match y with
    | [] -> 0
    | n2 :: rest2 -> n2
| n1 :: rest1 ->
    match y with
    | [] -> n1
    | n2 :: rest2 ->
          (compare n1 n2)
          + (f rest1 rest2)
```

LLM Solution P_L

```
let rec f x y = □
```

Distance = 1

```
Compare □1 □2
```

Candidate P_1

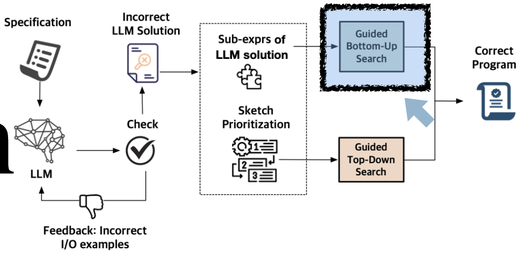
Distance = 0

```
match x with
| [] -> □1
| n1 :: rest1 -> □2
```

Candidate P_2

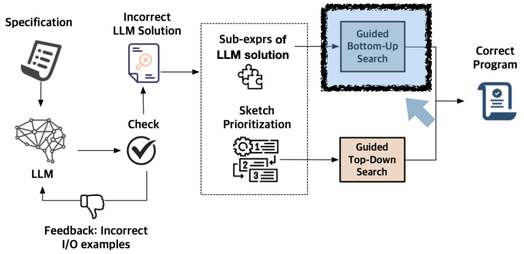
Pick candidate P_2 which is closest to the P_L

Bottom-up Guidance: Motivation



- Bottom-up enumerates components to fill holes in partial programs
- Components = sub-expressions that may be used in a solution
- **More components → higher cost**
- Observational Equivalence (OE) pruning removes semantic duplicates, but components that are never used in the correct solution still remain.

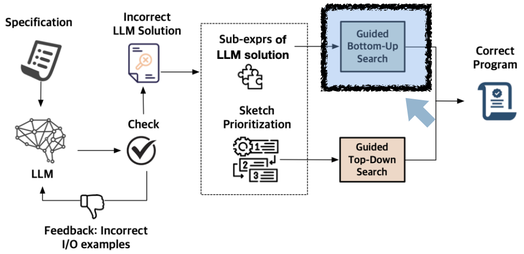
Bottom-up Guidance



- Enumerate only **components close to LLM** sub-expressions
- Discard components that are far from the LLM solution
- OE pruning removes duplicates + LLM guidance removes distant ones

complementary

δ Filtering

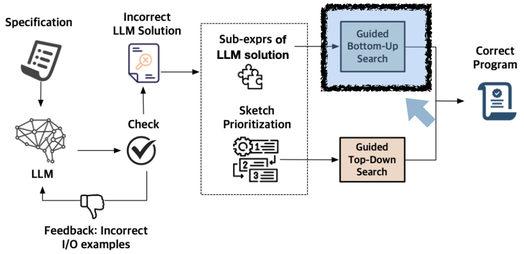


- Keep components if its distance to an LLM sub-expression is $\leq \delta$
  (Distance measured via *anti-unification)*
- Distance threshold δ: how much we *trust the LLM*
  (small $\delta \rightarrow$ keep only similar ones, large $\delta \rightarrow$ allow different ones too)
- Example: components filtering with $\delta = 1$

LLM sub-expressions: `[0, x, y, n1, n2, compare n1 n2]`

```
compare n1 n2 → min distance = 0 ≤ δ → kept
compare n1 0  → min distance = 1 ≤ δ → kept
n1 + n2       → min distance = 3 ≰ δ → discarded
```

Challenge & Solution



- Correct program requires "`compare n1 n2`"

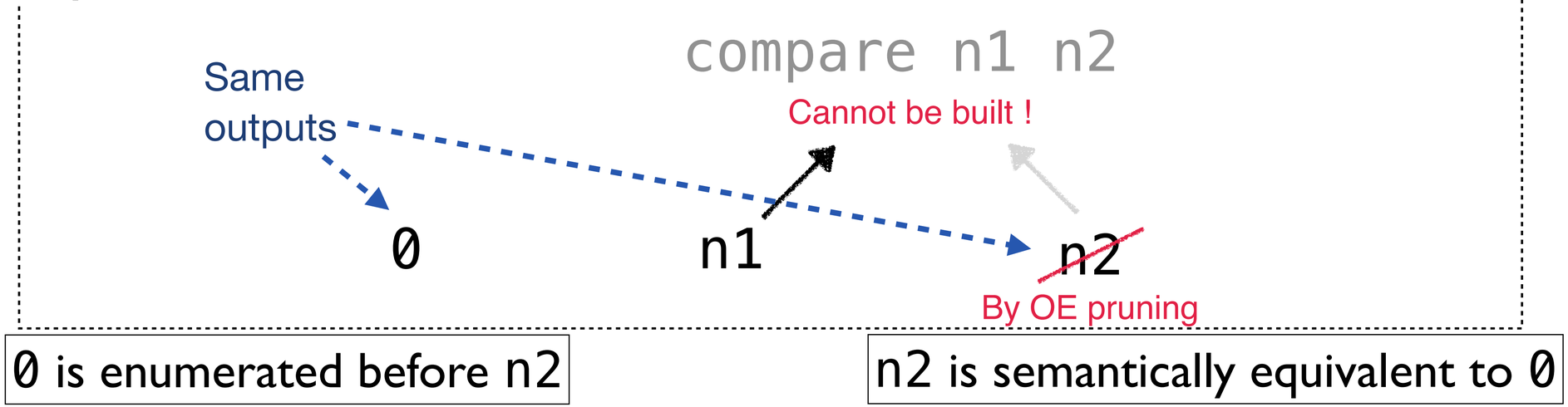
(n1, n2: first elements of each input lists)

LLM sub-expressions: `[0, x, y, n1, n2, compare n1 n2]`

`Spec : [] [0] → 0, [1,2,3] [0,1] → 6`

n2 n2

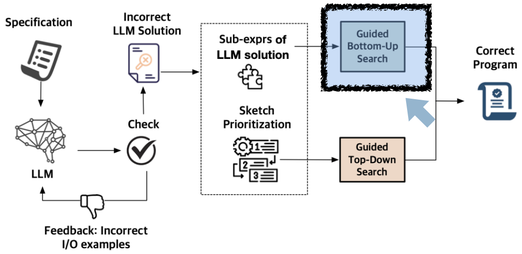
Components



`0` is enumerated before n2

n2 is semantically equivalent to `0`

Challenge & Solution



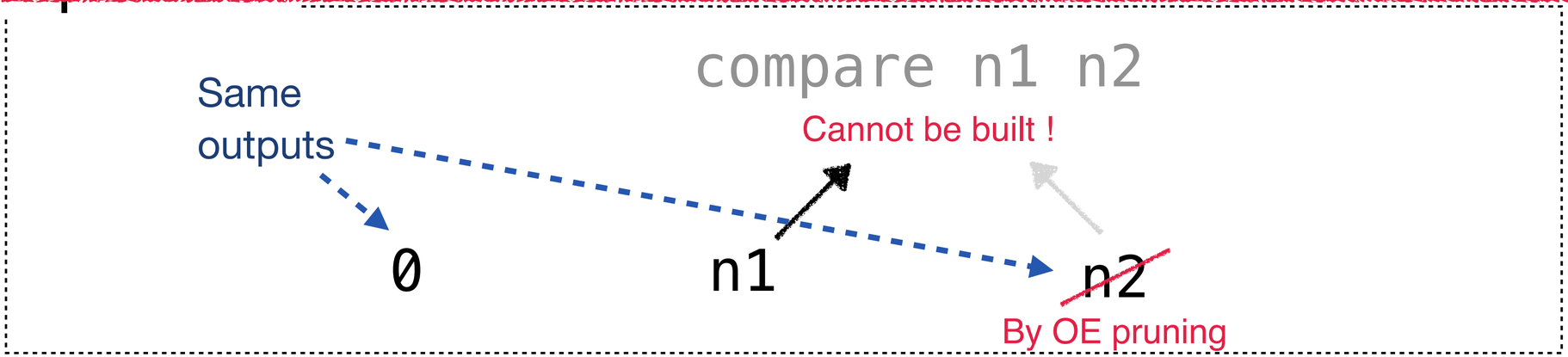
- Correct program requires "`compare n1 n2`"

(n1, n2: first elements of each input lists)

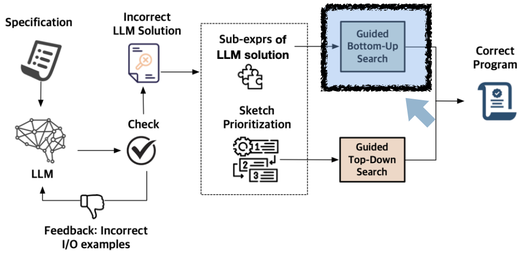
LL

Sp

OE pruning can discard components similar to the LLM output, thereby undermining the guidance by δ filtering.

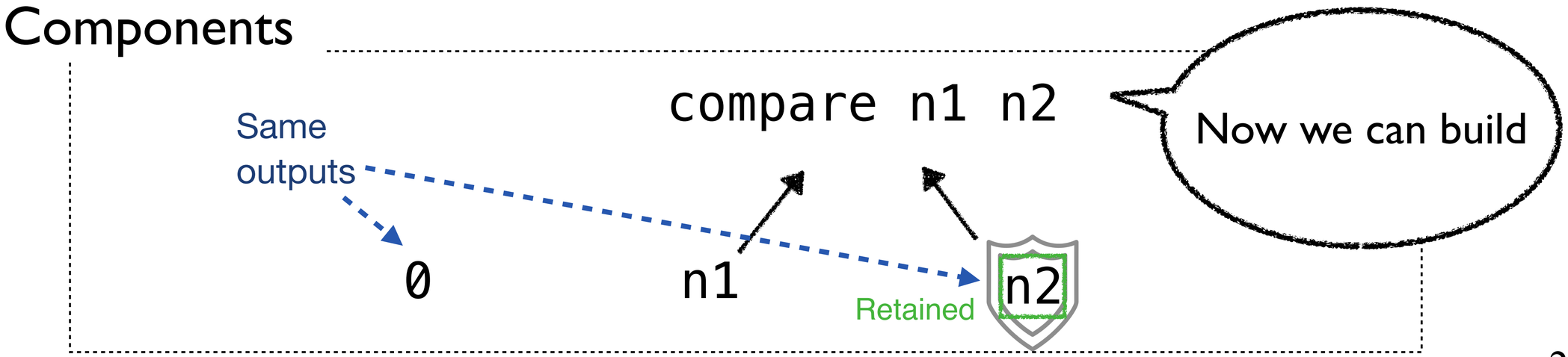


Challenge & Solution



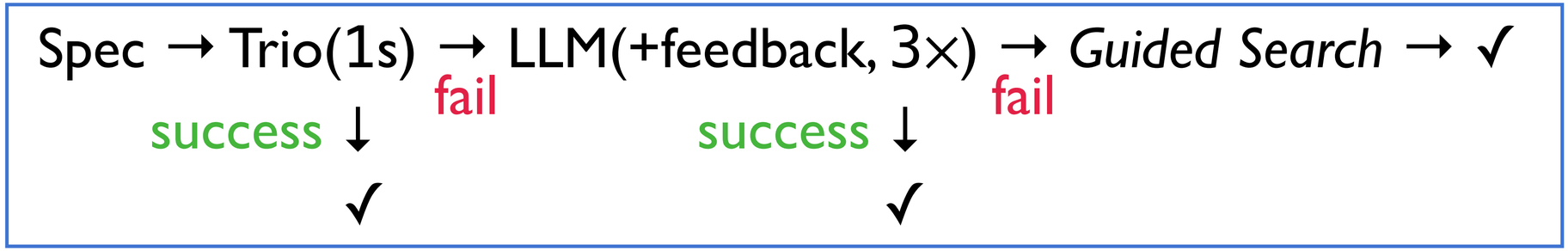
- *Pareto optimality* — retaining components that are closest to some LLM sub-expression
  - n2 is in LLM sub-expressions (distance = 0) → *Pareto optimal* and retained

LLM sub-expressions: `[0, x, y, n1, n2, compare n1 n2]`

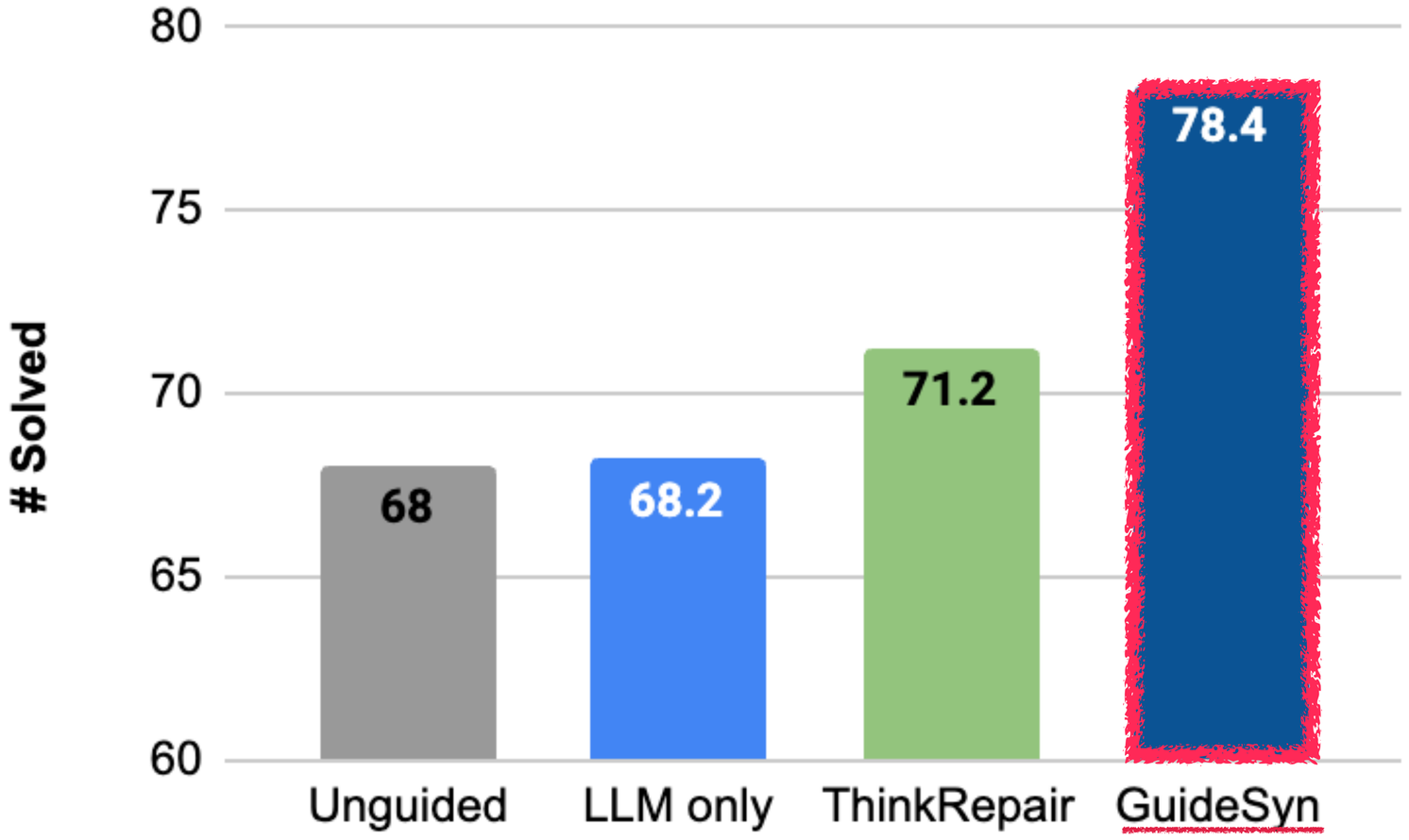


Evaluation Setup

- Our tool — GuideSyn implemented on top of Trio (POPL'23)
- Benchmark Suite (80 Recursive functional programs)
  - 60 from Trio benchmark + 20 from a functional programming course
- Baselines
  - ThinkRepair(ISSTA'24) : A state-of-the-art LLM-based program repair tool
  - LLMs: GPT-4o-mini, GPT-4o, o3-mini, DeepSeek-Coder-V2, Gemini-2.5-flash
- GuideSyn setup :



Evaluation Results (o3-mini, averaged over 5 runs)

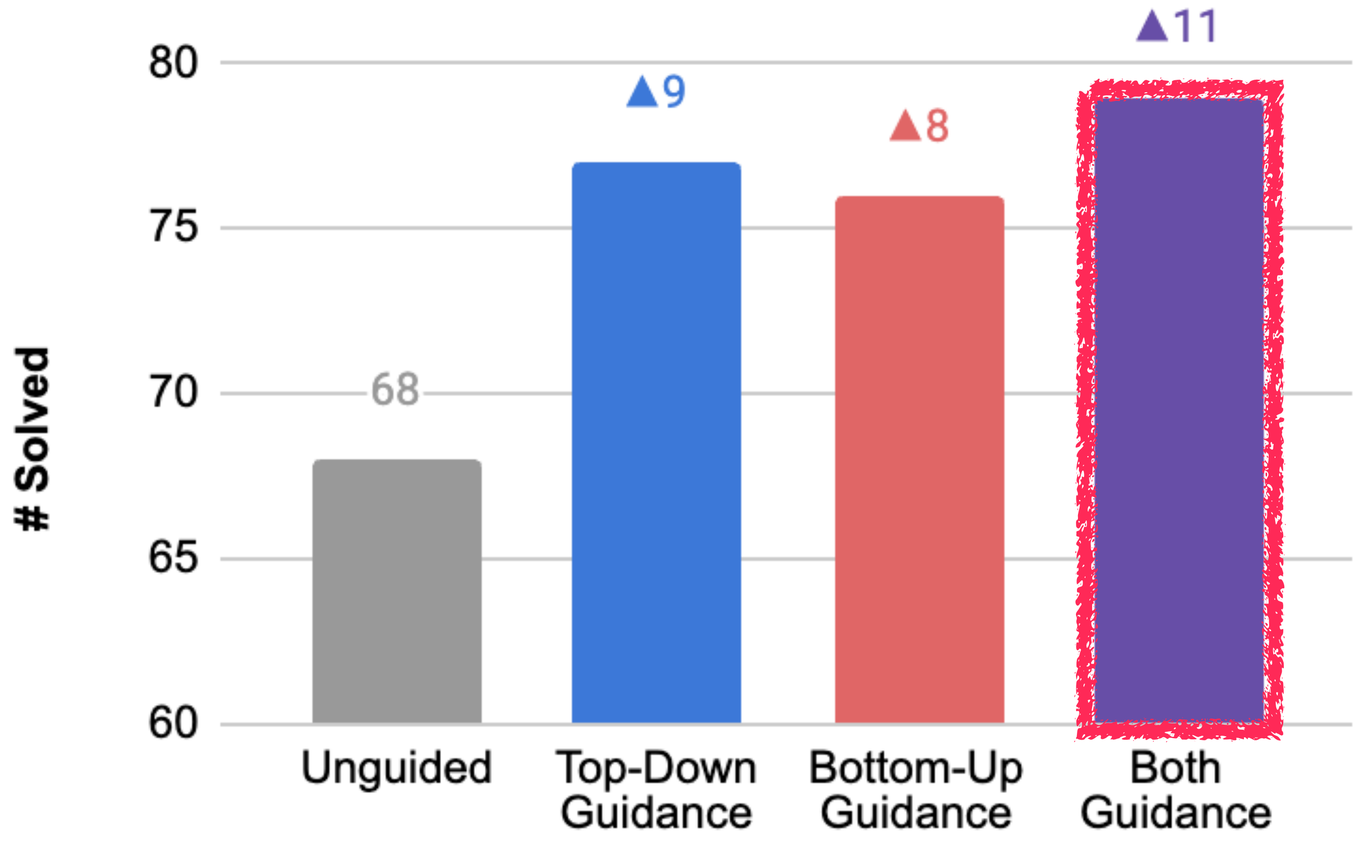


o3-mini is shown as a representative example; similar results hold for other models.

GuideSyn outperformed the others

Ablation Study (o3-mini, 1 run)

- GuideSyn performs better using Both Guidance

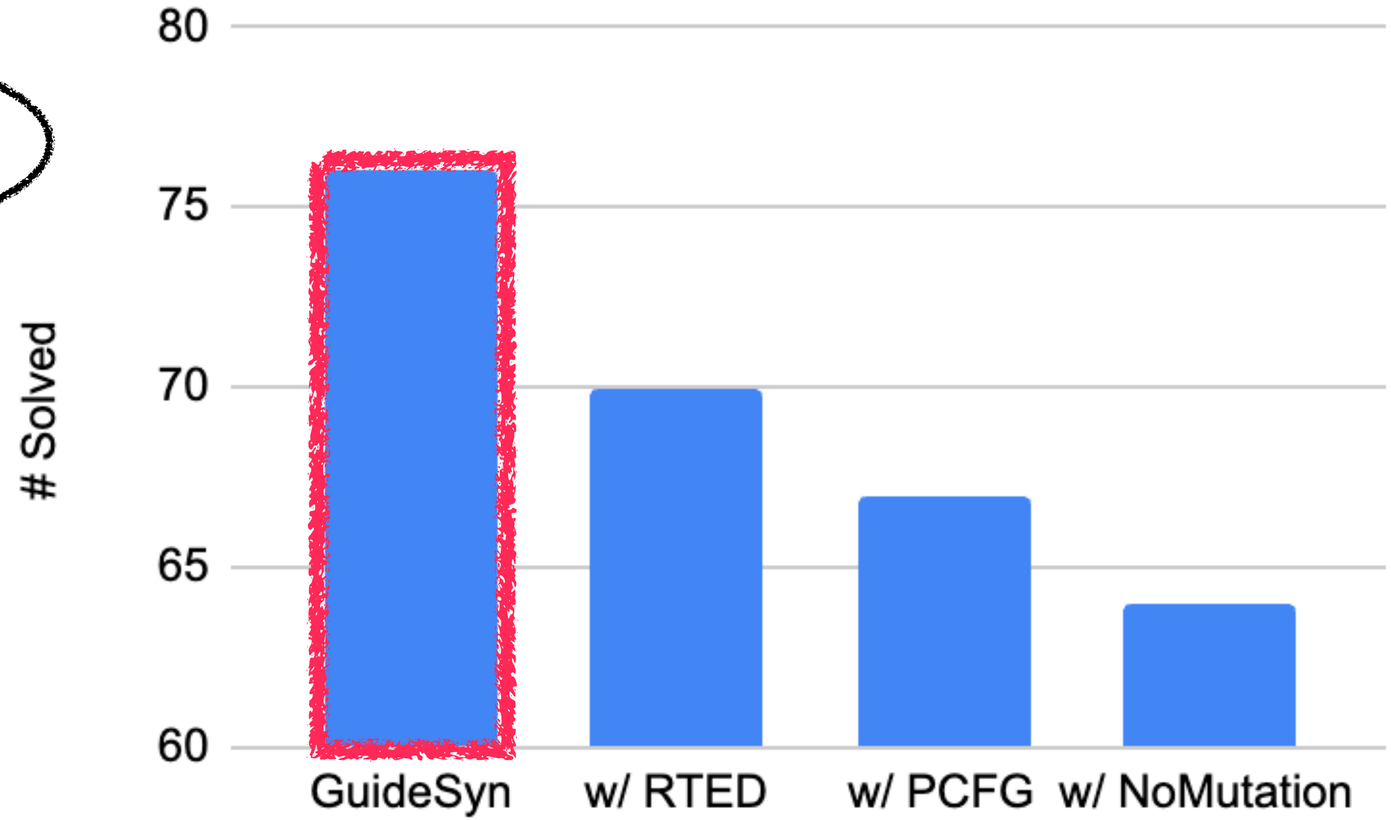


Evaluation of Other Alternatives (GPT-4o, 1 run)

- GuideSyn outperforms the approaches based on tree edit distance (RTED), PCFG-guided synthesis[*‡], and direct reuse of LLM-generated components[§].

Probabilistic Context-Free Grammar

Set $\delta = 0$



*LI, Yixuan, et al. Guided tensor lifting. PLDI'25

‡LI, Yixuan, et al. Guiding enumerative program synthesis with large language models. CAV'24

§Ding, Yuantian, and Xiaokang Qiu. Enhanced enumeration techniques for syntax-guided synthesis of bit-vector manipulations.. POPL'24

In conclusion..

- We proposed an anti-unification-based metric that efficiently measures structural similarity to the LLM solution
- Top-down search prioritizes candidates closest to LLM
- Bottom-up search enumerates LLM-close components, preserving OE pruning
- GuideSyn outperforms LLM-only, search-only, and existing LLM-guided approaches (e.g., PCFG-based)

Thank you !!

Backup Slides

How do we set δ ?

Our approach

- Max δ = 5, increased gradually during synthesis
- Start tight (close to LLM solution) → expand as needed

Why δ = 5? (empirically determined)

- δ < 5 → misses useful variants (δ = 0 : -14 tasks)
- δ > 5 → more components, no gain

prompting strategies

1. One-shot
2. Role-based
3. Grammar-based

Prompt Design

Prompt Template

You are a code generator for an ML-like functional language. You must strictly follow all the constraints below.

Constraints:

- Output only code. No comments or explanations.
- …

Here's the ML-like language.

```
P ::= let rec (f:t) = fun (x:t) -> e
e ::= x | (e e) | k(e, ..., e) | Un_k(e, ..., e) | e.n
   | (e,...,e) | match e with bs
bs ::= p -> e | bs '|' p -> e
```

where `f` is a function name, `t` ranges over data types, `k` ranges over data type constructors, `Un_k` denotes a destructor which extracts all the subcomponents of a constructor application of `k` as a tuple, ...

Give me a single recursive function `f` in the above language that uses the following types and helper functions

`{Types and helper functions}`

and of type `{Type}` that satisfies the following input-output examples:

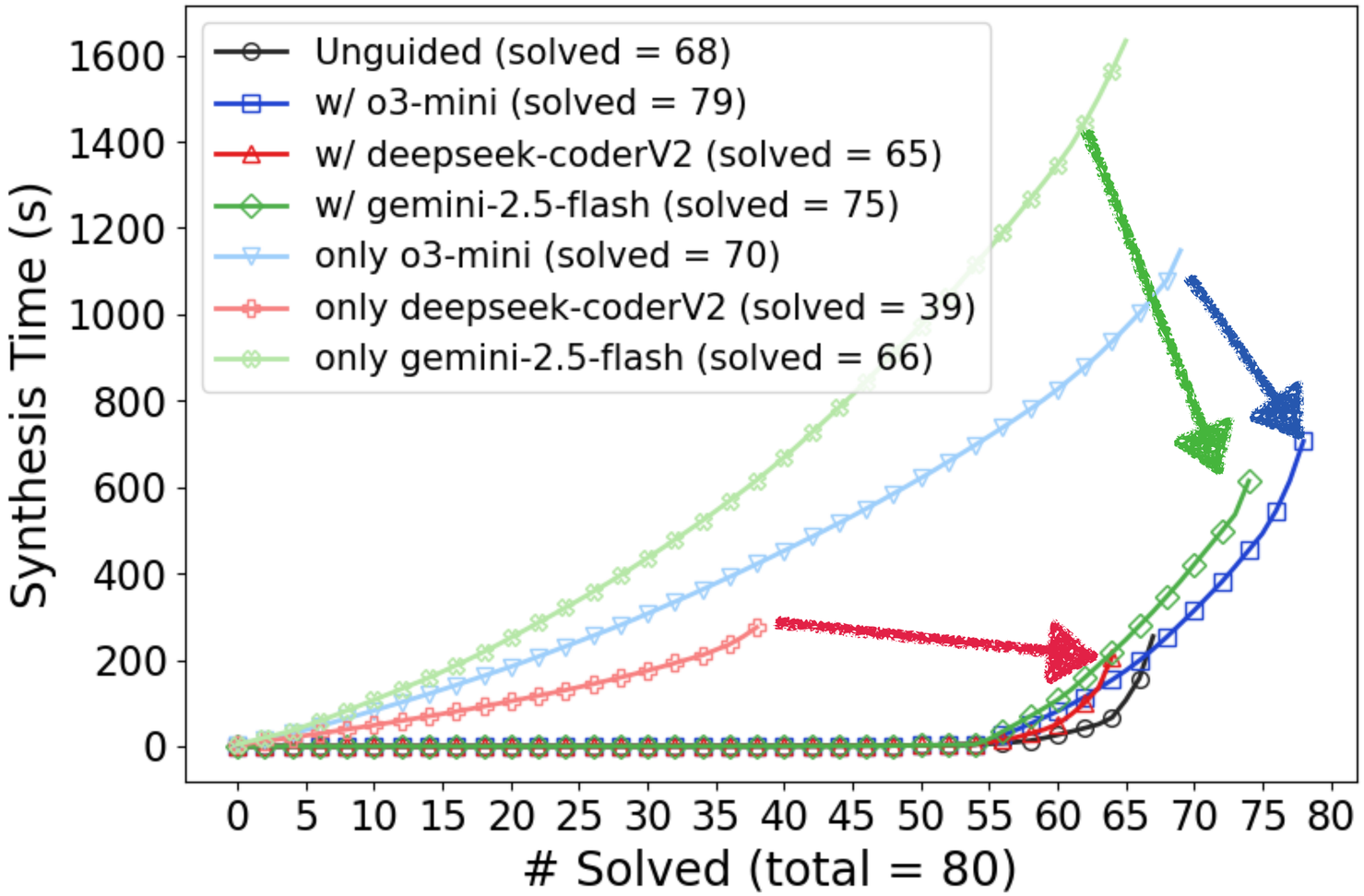
`{Input-output examples}`

- LLM role: code generator for an **ML-like functional language** (the Trio DSL)
- Direct generation outperforms "generate-then-translate" (OCaml/Haskell → Trio)
  (unreliable in practice: GPT-4o-mini, GPT-4o, and o3-mini produced untranslatable output on 28, 28, and 34 of 80 tasks, respectively)

Feedback (CEGIS-style, up to 3 attempts)

- Syntax error → retry with error message as feedback
- Wrong output → retry with failing I/O examples as counterexamples
- Best solution among 3 attempts is used for guided search

Cactus Plot: LLM-Only vs GuideSyn



- A line **closer to the x-axis** = better performance
- Solid color = GuideSyn (w/ LLM)
- Hollow color = LLM-Only

- GuideSyn solves **more tasks** in **less time** than LLM-Only

Failure Cases of LLM Guidance (1/2)

- Guidance can **misdirect the search** when the LLM's solution is far from correct solution — wasting time on wrong candidates

- Case: DeepSeek-Coder-V2
  - GuideSyn (w/ DeepSeek): 66 solved — fewer than Unguided (68)
  - DeepSeek fails to adapt to the Trio DSL → low-quality outputs
  - Wrong LLM solutions mislead both top-down and bottom-up search

Failure Cases of LLM Guidance (2/2)

- When the LLM solution is too far from the correct solution, guidance misdirects

Why GuideSyn Still Matters

- **Correctness guaranteed** — GuideSyn always returns a verified solution
- **Completeness preserved** — Guidance may mislead, but never excludes the correct solution.

- Incorrect LLM solution misleads both top-down and bottom-up search

GuideSyn with Frontier LLMs

- list_nth_occur (challenging) : Given a list and a number, return the index of its n-th occurrence (or the list's length if fewer than n occur).

LLM	LLM Time	LLM Result	+ GuideSyn	Total Time
Claude Opus 4.6	84.5s	Incorrect	28.0s	112.5s
GPT-5.4	218.0s	Correct	—	218.0s
GPT-4o-mini	3.8s	Incorrect	15.2s	19.0s

Note: Each LLM is queried only once per task here (large models take significant time per generation)

- Even frontier LLMs miss subtle spec details (e.g., 0- vs 1-indexed)
- GPT-4o-mini (3.8s) + GuideSyn (15.2s) = 19s total — **11.5X** faster than GPT-5.4 alone

δ Filtering in Detail

Definition 2 (Distance Profile). The distance profile of a term t is defined as:

$$\text{profile}(t) = [\text{dist}(t_1, t),\ \text{dist}(t_2, t),\ \ \ldots,\ \text{dist}(t_m, t)]$$

where t_i is the i-th element of subterms(P_L)

Definition 3 (Similar Components). For a given δ, a component e is similar w.r.t. P_L (denoted $e \approx_\delta P_L$) iff:

$$e \approx_\delta P_L \iff \exists\, v \in \text{profile}(e)\,.\ v \leq \delta$$

A component e is discarded if $e \not\approx_\delta P_L$.

Pareto Optimality in Detail

Partial order on profiles. $p \leq p' \iff \forall i,\ p_i \leq p'_i$

Definition 4 (*Pareto-Optimal* Components).

Given a set C, a component e is *pareto-optimal (denoted* pareto(e, C)*)* iff:

$$\text{pareto}(e, C) \iff \nexists\, e' \in C \text{ s.t. } \text{profile}(e') \leq \text{profile}(e)$$

Condition to retain a component (Algorithm 2, line 9): A component e is added to C iff:

$$\text{unique}(e, C, \Phi) \vee \text{pareto}(e, C)$$

- unique: there is no $e' \in C$ s.t. $e' \equiv_\Phi e$
- pareto: guarantees completeness — components closest to are never discarded by observational-equivalence pruning