Distance-Guided Search in Program Synthesis with Imperfect LLM Solutions

Hangyeol Cho
Hanyang University
Ansan, Republic of Korea
pigon8@hanyang.ac.kr

Jaehyung Lee
Hanyang University
Ansan, Republic of Korea
huna3869@hanyang.ac.kr

Woosuk Lee*
Hanyang University
Ansan, Republic of Korea
woosuk@hanyang.ac.kr

Abstract

Search-based program synthesis systematically explores a space of programs to find one that satisfies a given specification. While effective for small programs, it struggles with scalability due to the combinatorial explosion of the search space. In contrast, large language models (LLMs) can generate large programs but often produce solutions that are incorrect or fail to meet the specification. We propose a novel distance-guided search algorithm that leverages imperfect LLM-generated programs to guide both top-down and bottom-up synthesis. Using an anti-unification-based distance metric, we prioritize candidates in the top-down search that are structurally similar to the LLM output. For bottom-up synthesis, we generate components close to subexpressions of the LLM solution while preserving completeness and pruning efficiency. We implement our approach atop Trio, a bidirectional synthesizer for recursive functional programs, and evaluate it on 80 synthesis tasks. Our results show that distance-guided search effectively combines the strengths of LLMs and search-based methods, solving tasks beyond the reach of either technique alone.

CCS Concepts

• **Software and its engineering → Automatic programming**.

Keywords

synthesis, large language models, distance-guided search

ACM Reference Format:
Hangyeol Cho, Jaehyung Lee, and Woosuk Lee. 2026. Distance-Guided Search in Program Synthesis with Imperfect LLM Solutions. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787779

1 Introduction

Search-based program synthesis automatically constructs programs from specifications such as input-output examples or logical constraints. It has been applied successfully to tasks including data wrangling [6, 9], invariant inference [31], program optimization [21], end-user programming [39], and security [19]. While highly effective for small programs over domain-specific languages (DSLs), its scalability is limited due to the combinatorial explosion of the search space.

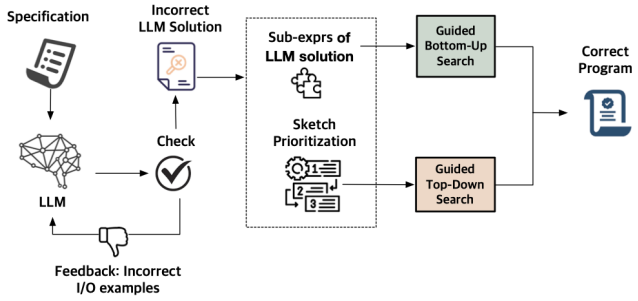

Figure 1: Overview of our distance-guided search algorithm.

In contrast, large language models (LLMs) have demonstrated remarkable scalability in generating large programs by predicting code directly from specifications. However, this scalability comes at the cost of reliability – LLM-generated programs often fail to satisfy the specification, particularly when targeting unfamiliar DSLs.

The complementary strengths and weaknesses of search-based synthesis and LLMs have motivated research that integrates the two [2, 13, 27, 28, 41]. Search-based synthesis (when with a verifier) ensures semantic correctness, while LLMs offer expressiveness power. [1] In such approaches, the synthesizer defines the DSL as the target language, and the LLM generates candidate programs in this DSL. Even if the LLM-generated solution is incorrect, it often contains useful structural hints and can be used to guide the search by prioritizing candidates close to the LLM output – under the hypothesis that the correct program lies in its vicinity. However, existing LLM-guided synthesis methods face three limitations.

First, there is no effective way to measure the similarity between candidate programs and the LLM-generated solution to guide the search. Based on the hypothesis that the LLM output is close to the correct program, prior methods have used edit distance metrics to explore candidates similar to the LLM solution first [26, 41]. However, computing edit distance metrics such as Levenshtein distance is computationally expensive, making them impractical for guiding search where many candidates (e.g., hundreds of thousands) must be explored quickly.

Second, they lack a mechanism to adjust the level of confidence in the LLM-generated solution. Existing approaches make one of two assumptions: either (1) the LLM output is mostly correct and can be directly reused without modification, or (2) it is largely

*Corresponding author

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787779

[1] If LLMs are with an external verifier, they can also ensure semantic correctness by re-generating code until it generates a correct one. However, this approach may require many attempts and thus can be inefficient when the LLM struggles with the target DSL, as we observed in our experiments.

incorrect but informative enough to train a surrogate model (e.g., a probabilistic grammar) that indirectly guides synthesis. There is no middle ground that allows flexible use based on confidence. Ideally, we would use the LLM output directly when confident and use it more loosely – e.g., as a search heuristic – when less certain.

Lastly, prior methods typically support only top-down or bottom-up synthesis, not both. Top-down synthesis starts with a partial program with missing parts (holes) and incrementally fills the holes, while bottom-up synthesis composes programs from smaller expressions. Each has strengths and weaknesses, and bidirectional synthesis – used in various systems [6, 14, 23, 24, 51] – has proven effective. In bidirectional synthesis, the top-down search generates partial program candidates with holes, while the bottom-up search generates components that can be used to fill in the holes in the partial program candidates. A unified LLM-guided approach supporting both strategies is thus desirable.

In this paper, we propose a novel distance-guided search algorithm that leverages imperfect LLM solutions to guide both top-down and bottom-up synthesis. As shown in Fig. 1, given a synthesis task that cannot be easily solved by a synthesizer, we prompt the LLM to generate candidate programs and iteratively refine them with counterexamples. If the LLM still fails after several iterations, its final output is used to guide the subsequent search.

In the top-down phase, we prioritize candidates structurally close to the LLM output, using an *anti-unification-based distance* metric. This metric is efficient to compute and captures structural similarity better than expensive alternatives like tree edit distance or Levenshtein distance, making it practical for guiding search over large candidate sets.

In the bottom-up phase, we generate components whose distance to some subexpression of the LLM output is within a threshold δ. This threshold controls how strictly the search should follow the LLM guidance. However, standard bottom-up enumeration relies on *observational equivalence* [46] to prune redundant components. This pruning could eliminate components similar to the LLM output, undermining the guidance. To address this, we introduce a novel bottom-up enumeration strategy that incorporates pareto optimality with respect to distances to LLM subexpressions. This approach retains components useful for LLM guidance while preserving the benefits of equivalence-based pruning.

We implement our method on top of Trio [24], a bidirectional synthesizer for recursive functional programs, and evaluate it on 80 synthesis tasks. Our experiments show that distance-guided synthesis effectively combines the strengths of LLMs and search-based techniques, solving problems that are challenging for either approach alone. We also demonstrate that our metrics are cost-effective and that our method outperforms previous approaches. The contributions of this paper are summarized as follows:

- We propose a novel anti-unification-based distance metric that efficiently measures structural similarity between candidate programs and imperfect LLM solutions to prioritize promising candidates during synthesis.
- We develop a new bottom-up enumeration algorithm that incorporates pareto optimality over distances to LLM subexpressions, while preserving observational equivalence-based pruning and enabling flexible confidence control via a distance threshold.
- We propose a unified approach to guide both top-down and bottom-up synthesis using LLM-generated solutions as flexible hints.
- We implement our approach in Trio and evaluate it on 80 synthesis tasks, demonstrating improved effectiveness over both standalone and existing LLM-guided methods.

2 Overview

In this section, we give an overview of our method using the problem of synthesizing a recursive function that manipulates lists of integers as an example.

Problem Specification. Given two lists of integers, we want to synthesize a function that compares the elements of the two lists at each index, gets the sum of the larger element at each index, and adds any remaining elements from the longer list if one list is longer than the other. For example, given the two lists `[1, 2, 3]` and `[0, 1]`, the function should return `6`.

The specification for the problem comprises inductive data types and helper functions that may be used by the target function to be synthesized, and input-output examples that the target function should satisfy. The following is the specification of the problem in OCaml-like syntax:

```
(* Inductive data types *)
type cmp = LT | EQ | GT

(* Helper functions *)
let compare (x1 : int) (x2 : int) : cmp =
    match x1 with
    | 0 -> (match x2 with 0 -> EQ | _ -> LT)
    | _ -> (match x2 with 0 -> GT
            | _ -> compare (x1 - 1) (x2 - 1))

(* Input-output examples *)
[] [0] -> 0,
[1, 2, 3] [0, 1] -> 6
```

Various approaches have been proposed to synthesize a recursive function from such a specification, enabling diverse applications such as end-user programming [18, 52], invariant inference [31]. and refactoring [16, 20].

Fig. 2a shows a solution to the problem. Because the function involves complex nested pattern matching and recursion, it is difficult to synthesize even for the state-of-the-art synthesizers for recursive programs [24, 29, 30] within 2 minutes.

We then use an LLM to generate a solution to the problem. [2] We prompt the LLM with the above specification and ask it to generate a solution. Whenever the LLM generates a syntactically invalid program or a program that does not satisfy the input-output examples, we ask it to regenerate a solution. After a few attempts, the LLM generates a solution as shown in Fig. 2b. However, a line-by-line inspection of the solution reveals that it is incorrect for the following reasons:

- It does not recursively call the function on the remaining elements when either list is empty (lines 6 and 9).

[2] In this example, we use the OpenAI GPT-4o-mini model. In the evaluation, we use more powerful models such as GPT-4o and o3-mini as well.

(a) A solution to the synthesis specification

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

(b) Incorrect solution generated by an LLM

```
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
```

Figure 2: Motivating example of our method. Incorrect lines in the LLM solution are highlighted in red.

- When both lists are non-empty, it causes a type error by adding a `cmp` value from `compare` to an integer from the recursive call. (line 11).

The LLM's solution is not directly repairable through simple edits, as two branches must be fixed and one added simultaneously due to their mutual dependence. Nevertheless, it captures partial structure, relevant operations, and key subexpressions, serving as a noisy yet informative signal that guides the synthesizer toward the correct solution.

We can observe that

- The overall structure of pattern matching is correct, as decisions are made based on whether the two given lists are non-empty.
- Key subexpressions such as `(compare n1 n2)` (that can be used as a scrutinee in the innermost pattern matching) and `(f rest1 rest2)` (that can be used for the recursive call) are present. Even though some desired expressions are not present, there are similar ones. For example, the expression `(f [] rest2)` in the correct solution is similar to `(f rest1 rest2)` in the LLM solution.

Based on these observations, we can guide the synthesis process to find a correct overall structure with missing parts and key subexpressions that can be used to complete the missing parts.

Existing Bidirectional Search-Based Synthesis. A popular search-based approach to program synthesis is bidirectional search [6, 14, 23, 24, 51], alternating between top-down and bottom-up search.

In the top-down search, the algorithm starts from an incomplete program with *holes* representing missing parts of the program, and gradually fills the holes with subexpressions or more concrete partial programs. During the search, input-output examples that should be satisfied by holes are inferred and used to guide the search. For example, in the top-down search for the above problem, the algorithm starts with a program skeleton such as

```
let rec f x y = □
```

and refines the hole with a more concrete program such as

Listing 1: Example of partial program candidate

```
let rec f x y =
    match x with
    | [] -> □1
    | n1 :: rest1 -> □2
```

and infers input-output examples that should be satisfied by the hole $\square_1$ as `[] [0] -> 0` and by the hole $\square_2$ as `[1, 2, 3] [0, 1] -> 6`. Then, each hole is further refined with another partial program with generating new holes associated with inferred input-output examples, or with a complete subexpression that satisfies the input-output examples, which is generated by the bottom-up search that will be described next. In addition to this refinement, the top-down search also often prunes infeasible candidates that cannot satisfy the specification no matter how the holes are filled.

The important aspect of the top-down search is that how to pick candidates for further refinement is crucial for the efficiency of the search. To tame the large search space, there have been various approaches to explore promising candidates first, such as using heuristics based on syntactic features of candidates [24, 33] or statistical models [1, 3, 25].

In the bottom-up search, the algorithm starts from the smallest expressions such as constants and variables and constructs larger expressions by combining smaller expressions. For example, in the bottom-up search for the above problem, the algorithm starts with constants such as `[]` and variables such as `f` and `x`, and constructs larger expressions such as `(f x [])`. Such expressions are used to fill the holes in partial programs generated by the top-down search.

The important aspect of the bottom-up search is that redundant expressions are pruned using the *observational equivalence* relation, which is a relation that two expressions are equivalent if they produce the same output for all input examples. For example, let variable `n2` denote the first element of the list `y`. Then, the expressions `0` and `n2` are observationally equivalent because they produce the same output for all input examples (`n2` always evaluates to `0` since the first element of the second input list is `0` in every case). When `n2` is generated after `0` is generated, the algorithm discards `n2`. In this manner, only one representative for each observational equivalence class is kept, which significantly reduces the number of expressions to be considered in the bottom-up search. This pruning is crucial not only for expediting the bottom-up search itself but also for improving the overall efficiency of the bidirectional search by reducing the number of component expressions for hole filling.

How the top-down and bottom-up searches interplay in the bidirectional search is various. For example, they may run in parallel [14] or the bottom-up search is first run to generate components and then the top-down search is run to fill holes with the generated components [23, 24].

Based on the observations on the LLM solution, we propose a distance-guided search algorithm that guides both the top-down and bottom-up search with the LLM solution.

Anti-Unification-Based Distance Metric for Guiding Top-Down Search. Since the LLM's incorrect solution often contains a correct overall structure despite some details being incorrect, we guide the top-down search to prioritize candidates having a similar structure to the LLM solution.

To this end, we use a distance metric based on anti-unification [40] that can measure the structural similarity between two programs. Anti-unification is a process that generalizes two programs by replacing their mismatched parts with variables and leaving only common parts. For example, the anti-unification of the LLM solution in Fig. 2b and the correct solution in Fig. 2a is as follows:

```
let rec f x y =
    match x with
    | [] ->
        match y with
        | [] -> 0
        | n2 :: rest2 -> X
    | n1 :: rest1 ->
        match y with
        | [] -> Y
        | n2 :: rest2 -> Z
```

where **X**, **Y**, and **Z** are variables that represent the mismatched parts (often called *pattern variables*). Then, the distance from program P_1 to another program P_2 is defined as the size of the expressions in P_1 corresponding to the pattern variables **X**, **Y**, and **Z** (i.e., the mismatched parts in P_1), which is measured by the number of nodes in the abstract syntax tree (AST) of the expressions.

A complication that arises here is that partial program candidates generated by the top-down search may have holes in contrast to the LLM solution, which is a complete program. Because holes are not concrete expressions, they cannot be directly compared with expressions in the LLM solution. To address this, we deal with holes by treating them as placeholders that can be filled with any expression. For example, the anti-unification of the partial program in Listing 1 and the LLM solution in Fig. 2b is the partial program itself. Because we optimistically assume that the holes can be filled in a way that minimizes the distance to the LLM solution, the holes are not replaced with pattern variables in the anti-unification.

During the top-down search, we use the distance metric to pick a candidate that is most similar to the LLM solution among candidates to be explored next. This way, we can guide the search to explore candidates that are structurally similar to the LLM solution first, which is likely to lead to a correct solution. In addition to preserving the common structure, another important benefit of this anti-unification-based metric is that it can be done efficiently with a linear time complexity with respect to the size of programs being compared, which is crucial for our purpose since millions of candidates are explored during the top-down search.

Similarity-Based Guidance for Bottom-Up Search. In the bottom-up search, we guide the search to generate components that are similar to the LLM solution.

We define a similar component as a component having an edit distance to a subexpression of the LLM solution that is smaller than a user-provided threshold δ. This threshold controls how confident we are about the LLM solution is correct. The distance is measured by the anti-unification-based distance metric described above.

Unfortunately, how to apply the observational equivalence pruning while generating only similar components is not straightforward. Suppose we perform the bottom-up search as usual with only difference that we discard any components that are not similar to the LLM solution. Then, the search may discard a component that may be used to construct a larger component that is similar to the LLM solution. For example, suppose δ is set to 1 (meaning that we only generate components that appear in the LLM solution) and we enumerate the component `0`. The LLM solution has a subexpression `0`. Therefore, the edit distance of `0` is 0, which is smaller than the threshold δ. Then, the component `0` is kept. Next, we enumerate the component `n2`. `n2` is discarded because it is observationally equivalent to `0`. Then, we never be able to construct `compare n1 n2` but just `compare n1 0` because `n2` was discarded. `compare n1 0` has an edit distance of 1 because the closest subexpression of the LLM solution is `compare n1 n2`, which has an edit distance of 1 from `compare n1 0`. Thus, we cannot construct `compare n1 n2` that should have been generated by the bottom-up search because its edit distance is smaller than the threshold δ.

To address this issue of search incompleteness, we propose a new bottom-up search algorithm based on the notion of pareto optimality. We first extract all subexpressions of the LLM solution: [`0`, `n2`, `n1`, `x`, `y`, ...] and whenever we enumerate a component, we compute edit distances of the component to all subexpressions of the LLM solution. For example, for the component `n2`, the list of edit distances to the subexpressions is $[1, 0, 1, 1, 1, \ldots]$ and for the component `0`, the list of edit distances is $[0, 1, 1, 1, 1, \ldots]$. When we enumerate a component, we keep it not only when it is unique (i.e., no other component is observationally equivalent to it), but also when it is pareto optimal with respect to the list of edit distances. A component is pareto optimal if there is no other component that has a smaller edit distance to all subexpressions of the LLM solution. For example, `n2` is kept because even though it is observationally equivalent to `0`, since there is no other component that has a smaller edit distance to all subexpressions of the LLM solution.

This way, we can ensure that the search never misses a component that is similar to the LLM solution while enjoying the benefits of the observational equivalence pruning, which we have formally proved in the paper.

With the incorrect LLM solution and the top-down and bottom-up search guidance methods based on the novel anti-unification-based distance metric, we can successfully synthesize a correct solution within 6 seconds.

3 Problem Definition

In this section, we define the problem of guiding program synthesis with incorrect LLM solutions. We first introduce preliminary concepts and then define a generic program synthesis algorithm

based on top-down search and bottom-up search, which is often called bidirectional search. Lastly, we define the problem of guiding this algorithm with LLM solutions.

3.1 Preliminaries

Terms. A signature Σ is a set of function symbols each of which has an arity. A constant is a special function symbol of arity 0 (i.e., a nullary function symbol). Let V be a set of variables. A term over Σ and V is inductively defined; a variable is a term, a constant symbol is a term, and if $f \in \Sigma$ is a function symbol of arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is also a term. The size of a term is the number of function symbols and variables in it, and we use $|t|$ to denote the size of a term t. We use $\mathrm{subterms}(t)$ to denote the set of all subterms of a term t (including t itself). We often use the term program or expression interchangeably with a term.

Context-Free Grammar. A context-free grammar G is a tuple (N, Σ, R, S) where N is a set of non-terminal symbols, Σ is a set of terminal symbols, R is a set of production rules, and $S \in N$ is the start symbol. Each production rule is of form $A, A_1, \cdots, A_k \in N$, $f \in \Sigma$ and $arity(f) = k$. [3] A sequence symbols containing at least one non-terminal symbol is called a *partial program*, and a sequence of terminal symbols is called a *complete program*. A program P' is derived from another partial program P in one step if there exists a production rule $A \longrightarrow \alpha$ in R such that P' can be obtained from P by replacing a non-terminal symbol A in P with α. We say that a partial program P' is derived from a partial program P if P' can be obtained from P by applying a sequence of production rules in R. We use $L(N)$ to denote the set of all complete programs that can be derived from a nonterminal symbol N in G. The language of G is defined as $L(G) = L(S)$, which is the set of all complete programs that can be derived from the start symbol S in G.

Program Synthesis. A program synthesis task is defined as a tuple (G, Φ) where $G = (N, \Sigma, R, S)$ is a context-free grammar and Φ is a specification of the desired program. A specification Φ is a logical formula that describes the properties that the synthesized program should satisfy. Input-output examples, pre- and post-conditions are common forms of such specifications. We say that a program P satisfies Φ (denoted $P \models \Phi$). A program synthesis algorithm takes a program synthesis task (G, Φ) and returns a complete program $P \in L(G)$ such that $P \models \Phi$ if such a program exists.

Anti-Unification. A substitution $\sigma = [x_1 \mapsto t_1, \cdots, x_n \mapsto t_n]$ is a mapping from variables in V to terms over Σ. We write $\sigma(t)$ to denote the term obtained by applying the substitution σ to term t.

A term t is more general than another term t', written $t \sqsubseteq t'$, if there exists a substitution σ such that $t' = \sigma(t)$. For example, $f(x, 1) \sqsubseteq f(2, 1)$ because we can apply the substitution $[x \mapsto 2]$ to $f(x, 1)$ to obtain $f(2, 1)$. The relation $\sqsubseteq$ is a partial order on terms, which is reflexive, transitive, and antisymmetric.

The anti-unification of two terms t_1 and t_2 (denoted $t_1 \sqcup t_2$) is a term t such that t is the least general term that is more general than both t_1 and t_2, i.e., $t_1 \sqsubseteq t$ and $t_2 \sqsubseteq t$. For example, the anti-unification of $f(x, 1)$ and $f(1, y)$ is $f(x, y)$, which is the least general pattern that is more general than both $f(x, 1)$ and $f(1, y)$.

[3] This grammar is actually a regular tree grammar, but we stick with the more familiar context-free grammar for simplicity of presentation.

Algorithm 1 Bidirectional Search-Based Synthesis

Require: A synthesis task $\langle G = \langle N, \Sigma, R, S \rangle, \Phi \rangle$
Ensure: A solution program $P \in L(G)$ that satisfies Φ

```
1:  procedure Synthesis(G, Φ)
2:      C := ∅                                              ▷ C : 𝒫(L(G))
3:      n := 1
4:      repeat
5:          C := BottomUp(G, n)
6:          Q := {S}
7:          while Q is not empty do
8:              P := Pick(Q)
9:              Q := Q \ {P}
10:             if P is partial then
11:                 Q := Q ∪ Refine(P, R, C)
12:             else
13:                 if P ⊨ Φ then return P
14:             end if
15:         end while
16:         n := n + 1
17:     until the budget is exhausted
18: end procedure
```

Computing the anti-unification of two terms can be done by a simple top-down traversal of the two terms, replacing any mismatched subterm by a new fresh variable.

3.2 A Generic Program Synthesis Algorithm

We consider a generic synthesis algorithm that uses top-down search and bottom-up search (often called bidirectional search) to synthesize a program from a context-free grammar G and a specification Φ.

Algorithm 1 describes the algorithm that takes as input a context-free grammar G and a specification Φ and returns a solution if it exists. The algorithm maintains a *component set* $\mathbf{C}$ containing complete terms that can be used to complete partial programs. The components are generated by the BottomUp function (line 5) which generates components of size $\leq n$ from the grammar G. The size limit n is initialized to 1 (line 3) and is increased by 1 at the end of each iteration (line 16). The inner loop from lines 7 to 15 performs the top-down search. At every iteration, a candidate program P is picked from the queue Q (line 8), and if P is partial, it is refined by the Refine function (line 11) which generates a set of candidate programs that can be explored in the next iteration. To guarantee termination, the Refine function returns the empty set if the size of P exceeds a certain limit. Bounding the search space is a common practice in synthesis [24, 29, 30]. If no complete program during the search satisfies the specification Φ, the algorithm increases the size limit n and repeats the process with more components of size $\leq n$.

Note that the BottomUp that performs bottom-up enumeration applies the *observational equivalence pruning*. It constructs the smallest components first and then combines them to form larger components. During the process, only either one of two components e and e' that are equivalent to each other with respect to the specification Φ (denoted $e \equiv_\Phi e'$) is kept in $\mathbf{C}$. For example, if Φ is a set of input-output examples, $e \equiv_\Phi e'$ if e and e' generate the same output for all input examples in Φ.

The algorithm is general enough to cover not only bidirectional algorithms [6, 14, 23, 51] but also the ones solely based on top-down [25, 29] or bottom-up search [3, 30, 46] by disabling either one of the two search strategies.

3.3 Guiding Program Synthesis with LLM Solutions

Now we define the problem of guiding the synthesis algorithm with incorrect LLM solutions. Suppose we have a synthesis task (G, Φ) which is challenging to solve with Algo. 1 within a given budget. We assume that we have access to a large language model (LLM) $\mathcal{L}$ that can generate a solution $P_{\mathcal{L}}$ to the task. Suppose $P_{\mathcal{L}}$ is incorrect. It is known that even if LLMs generate incorrect solutions, the correct solutions are often in the vicinity of the incorrect solutions. Based on this observation, we aim to guide the search of Algo. 1 using $P_{\mathcal{L}}$ as a hint by modifying the Pick function (line 8) and the BottomUp function (line 5).

Suppose we have a distance function dist that computes the distance between two programs and a similarity function sim that computes the similarity between two programs.

The Pick function for guiding the top-down search should satisfy the following condition:

- **Prioritization**: Let $P = \textsc{Pick}(Q)$ where Q is a set of partial or complete programs. Then, there exists a program P' that can be derived from P such that $\mathsf{sim}(P', P_{\mathcal{L}})$ is maximizes among all programs that can be derived from partial programs in Q.

The BottomUp function for guiding the bottom-up search should satisfy the following conditions:

- **Proximity**: Let $\mathbf{C} = \textsc{BottomUp}(G, n)$ for a size limit n. Then, for every component $e \in \mathbf{C}$, there exists a subexpression $e' \in \mathsf{subterms}(P_{\mathcal{L}})$ such that $\mathsf{dist}(e, e') \leq \delta$ where δ is a distance threshold that can be set by the user (denoted $e \approx_\delta P_{\mathcal{L}}$).
- **No redundancy**: No two components should be equivalent to each other with respect to the specification Φ (i.e., $\forall e, e' \in \mathbf{C}$, if $e \equiv_\Phi e'$, then $e = e'$).
- **Completeness**: For every complete program $e \in L(G)$ of size $\leq n$ that is similar to a subexpression of $P_{\mathcal{L}}$, there exists a component in $\mathbf{C}$ that is equivalent to e with respect to the specification Φ. Formally, $\forall e \in L(G).\ |e| \leq n \wedge e \approx_\delta P_{\mathcal{L}} \implies \exists e' \in \mathbf{C}.\ e \equiv_\Phi e'$.

In the next section, we present our algorithms for these two functions that satisfy the above criteria.

4 Our Algorithm

In this section, we present our design of the Pick and BottomUp functions that guide top-down and bottom-up search respectively using an incorrect LLM solution $P_{\mathcal{L}}$. Throughout this section, we assume that the LLM solution $P_{\mathcal{L}}$ is a global variable that is accessible to both functions.

4.1 The Pick Function for Top-Down Guidance

The Pick function is used to select a promising candidate program to be explored first from a set of candidates Q. The function is defined as follows:

$$\textsc{Pick}(Q) \stackrel{\mathsf{def}}{=} \arg\max_{P \in Q} \mathsf{sim}(P, P_{\mathcal{L}})$$

where $\mathsf{sim}(P, P_{\mathcal{L}})$ is a function that measures the similarity between a candidate program P and the LLM solution $P_{\mathcal{L}}$. The similarity function is defined as follows:

$$\mathsf{sim}(P, P_{\mathcal{L}}) = \frac{2 \times (|P_{\mathcal{L}}| - \mathsf{dist}(P, P_{\mathcal{L}}))}{|P| + |P_{\mathcal{L}}|}$$

where $\mathsf{dist}(P, P_{\mathcal{L}})$ is a distance function that measures the distance between the two programs P and $P_{\mathcal{L}}$. The distance function returns a value between 0 and $|P_{\mathcal{L}}|$, where 0 means that the two programs are identical and $|P_{\mathcal{L}}|$ means that they are completely different, thereby making the similarity function also return a value between 0 and 1.

There are many ways to measure the distance between two programs, such as tree edit distance [36, 37] or even string edit distance if the programs are represented as strings. However, these methods are not suitable for our purpose because they are not efficient enough to compute during the synthesis process. The dist function is invoked as many as the number of candidate programs explored during the synthesis process, which can be very large (e.g., millions). The tree edit distance has a cubic complexity in the worst case, and the string edit distance has a quadratic complexity in the worst case. These complexities are too high to be used in our synthesis algorithm as will be shown in Sec. 6.

We use anti-unification to measure the distance between two programs. Computing the anti-unification of two terms can be done in linear time with respect to the size of the two terms. The distance function is defined as follows:

$$\mathsf{dist}(P, P_{\mathcal{L}}) = \sum_{x \in V} |\sigma(x)|$$

where σ is a substitution such that $\sigma(P \sqcup^* P_{\mathcal{L}}) = P_{\mathcal{L}}$ and $\sqcup^*$ is the anti-unification operator that computes the anti-unification of two terms *considering nonterminal symbols*. Since nonterminal symbols are placeholders that can be replaced with any term, we should regard them as wildcards that can match any term. The new anti-unification operator $\sqcup^*$ is defined as follows:

$$t_1 \sqcup^* t_2 \stackrel{\mathsf{def}}{=} \begin{cases} f(t_{11} \sqcup^* t_{21}, \ldots, t_{1n} \sqcup^* t_{2n}) & (t_1 = f(t_{11}, \ldots, t_{1n}),\ t_2 = f(t_{21}, \ldots, t_{2n})) \\ t_1 & (t_1 \in N) \\ t_2 & (t_2 \in N) \\ \text{a fresh variable } x & (\text{otherwise}) \end{cases}$$

Theorem 1. *The Pick function satisfies the prioritization condition.*

Proof. Available in the supplementary material. □

4.2 The BottomUp Function for Bottom-Up Guidance

Algo. 2 describes our bottom-up enumeration algorithm. The algorithm takes as input a context-free grammar G and a size limit n and returns $\mathbf{C}$ containing components similar with respect to the LLM solution $P_{\mathcal{L}}$.

To define what similar components are, we begin with the following definition:

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

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

Algorithm 2 Our Bottom-Up Enumeration Algorithm

```
1:  procedure BottomUp(G = ⟨N, Σ, R, S⟩, n)
2:      C := ∅
3:      k := 1
4:      repeat
5:          for all A → β ∈ R do
6:              E := {β[e1/A1, ..., em/Am] | ei ∈ C, Ai ∈ N, Ai in β}
7:              for all e ∈ E do
8:                  if e ∉ L(A) ∨ |e| > k ∨ (e ≉δ P_A) then continue
9:                  if unique(e, C, Φ) ∨ pareto(e, C) then
10:                     C := C ∪ {e}
11:                 end if
12:             end for
13:         end for
14:         k := k + 1
15:     until k > n
16:     C := RemoveRedundant(C)
17:     return C
18: end procedure
```

where dist(t, t') is the distance between t and t' as defined in Sec. 4.1 and t_i is the i-th element of subterms$(P_{\mathcal{L}})$, which is the list of subterms of the LLM solution $P_{\mathcal{L}}$.

Definition 3 (Similar Components). *For a given δ as a maximum edit distance threshold, a component e is similar with respect to $P_{\mathcal{L}}$ (denoted $e \approx_\delta P_{\mathcal{L}}$) iff $\exists v \in profile(e).\ v \leq \delta$.*

We say a profile p is less than or equal to a profile p' (denoted $p \leq p'$) if for all i, $p_i \leq p'_i$. Based on this partial order, we define the notion of pareto-optimality as follows:

Definition 4 (Pareto-Optimal Components). *Given a set of terms $\mathbf{C}$, a component e is pareto-optimal (denoted pareto$(e, \mathbf{C})$) iff there is no other component $e' \in \mathbf{C}$ such that profile$(e') \leq$ profile(e).*

Lemma 5. *Suppose we have $e, e' \in L(G)$ such that profile$(e) \leq$ profile(e') and $|e| \leq |e'|$. Then, for every term $t \in L(G)$, profile$(t[e/e']) \leq$ profile(t) where $t[e/e']$ is the term obtained by replacing every occurrence of e' with e in t.*

Proof. Available in the supplementary material. □

The following theorem states that a non-pareto-optimal component e' that is equivalent to another pareto-optimal component e with respect to the specification Φ can be replaced by the pareto-optimal component if e is not larger than e'.

Theorem 6. *For a synthesis task with a grammar G and a specification Φ, suppose we have two terms $e, e' \in L(G)$ such that $e \equiv_\Phi e'$, profile$(e) \leq$ profile(e'), and $|e| \leq |e'|$. Then, for every $t \in L(G)$ containing e' such that $t \approx_\delta P_{\mathcal{L}}$, $t[e/e'] \approx_\delta P_{\mathcal{L}}$ and $t[e/e'] \equiv_\Phi t$ where $t[e/e']$ is the term obtained by replacing every occurrence of e' with e in t.*

Proof. Available in the supplementary material. □

Now we are ready to explain our BottomUp function described in Algo. 2. The algorithm starts with the initial set of components $\mathbf{C}$ which is empty at the beginning (line 2). The algorithm then iterates over the production rules of the grammar G (line 5) and generates a set E of new component candidates by replacing nonterminal symbols in the right-hand side of each production rule with components from $\mathbf{C}$ (line 6). If $e \in E$ cannot be derived from the nonterminal symbol A (i.e., $e \notin L(A)$), or if the size of e is larger than k which is the current size limit, or if e is not similar with respect to $P_{\mathcal{L}}$ (i.e., $e \not\approx_\delta P_{\mathcal{A}}$), then it is discarded (line 8). If e is unique with respect to the specification Φ (i.e., there is no other component $e' \in \mathbf{C}$ such that $e' \equiv_\Phi e$), or if e is pareto-optimal with respect to the components in $\mathbf{C}$ (line 9), then e is added to the set of components $\mathbf{C}$ (line 10). This process is repeated until the size limit n is reached (line 15). Finally, the algorithm removes redundant components from $\mathbf{C}$ (line 16) (i.e., among multiple components that are equivalent to each other with respect to the specification Φ, only one is kept) and returns the set of components $\mathbf{C}$ (line 17). The crux of the algorithm is to add pareto-optimal components even if they are not unique with respect to the specification Φ (line 9). This part is for the completeness condition, which ensures that we do not miss any component that is similar with respect to $P_{\mathcal{L}}$ during the bottom-up enumeration. The completeness condition is guaranteed by the theorem above. If the component e does not satisfy the condition at line 9, that is, there exists another component $e' \in \mathbf{C}$ such that $e \equiv_\Phi e'$ and profile$(e') \leq$ profile(e). Because we enumerate components in a bottom-up manner, e' that is generated before e must be smaller than or equal to e in size. By the previous theorem, we can guarantee that e' can be used in place of e in any program that contains e. Therefore, we can discard e and keep e' in the set of components $\mathbf{C}$ to ensure that we do not miss any component similar to the components of the LLM solution $P_{\mathcal{L}}$.

The following theorem states that the algorithm satisfies the three conditions mentioned above.

Theorem 7. *The BottomUp function satisfies the proximity, no redundancy, and completeness conditions.*

Proof. Available in the supplementary material. □

4.3 Working Example

In this section, we illustrate the interaction between the Pick and BottomUp functions through a simple example.

Given two integers `x` and `y`, suppose we wish to synthesize a function that returns their maximum. The specification Φ provided to Algorithm 1 consists of three input–output examples: `-1 0 -> 0`, `0 0 -> 0`, and `1 0 -> 1`. The desired solution is:

```
match compare x y with LT -> y | EQ -> x | GT -> x
```

Suppose the LLM-generated solution $P_{\mathcal{L}}$ is: `compare x y`

The input grammar G for Algorithm 1 includes production rules for generating `match` expressions, function calls to `compare`, the parameter variables `x`, `y`, and the constant `0`. We set the maximum edit-distance threshold δ to 1. The list of subterms of $P_{\mathcal{L}}$ is:

`[compare, x, y, compare x y]`.

We first illustrate how the BottomUp function operates. Suppose that, in line 5 of Algorithm 1, it is invoked to generate components for bottom-up guidance with $n = 3$. The function initializes $\mathbf{C}$ as an empty set (line 2 in Algorithm 2) and begins generating components of size 1 (line 3). The first component, `x`, has distance profile profile$(\texttt{x}) = [1, 0, 1, 1]$. Since some element is $\leq \delta$,

we have x $\approx_\delta P_{\mathcal{L}}$, and it is added to **C**. Next, `0` is generated with profile(`0`) = [1, 1, 1, 1] and added similarly. Then, y is generated with profile(y) = [1, 1, 0, 1]. Although y $\equiv_\Phi$ `0` (both return 0 for all inputs), it is Pareto-optimal since profile(`0`) is strictly greater and profile(x) is incomparable. Thus, y is also added to **C**. After completing the size-1 generation, **C** = {x, `0`, y}.

Next, k is incremented to 3 (line 14), and the BottomUp function generates `compare x x`. Its profile profile(`compare x x`) = [3, 3, 3, 1] has a value $\leq \delta$, where the last element is 1 since dist(`compare x y`, `compare x x`) = $\sum_{x \in V} |\sigma(x)|$ with σ = y $\mapsto$ x. Thus, `compare x x` $\approx_\delta P_{\mathcal{L}}$ and, being unique under Φ, is added to **C**. Then, `compare x y` is generated with profile(`compare x y`) = [3, 3, 3, 0]; it also satisfies $\approx_\delta$ and is added to **C**. The next candidate, `compare 0 x`, has profile(`compare 0 x`) = [3, 3, 3, 2] with no entry $\leq \delta$, so it is discarded (line 8). Continuing this process, the loop (lines 4–15) yields **C** = {x, y, `0`, `compare x x`, `compare x y`, `compare x 0`}. After applying RemoveRedundant (line 16), where `compare x y` $\equiv_\Phi$ `compare x 0` and y $\equiv_\Phi$ `0`, only `compare x y` and y remain. Hence, the final component set is

$$\mathbf{C} = \{\texttt{x}, \texttt{y}, \texttt{compare x x}, \texttt{compare x y}\}.$$

This final **C** is returned for bottom-up guidance in the subsequent top-down search (line 5).

Now suppose the Pick function (line 8 in Algorithm 1) is invoked to select a partial program from the queue Q during the top-down search. Assume Q contains two partial programs:

$$\{P_1 = \texttt{match compare x y with LT -> } \square_1 \texttt{ | EQ -> } \square_2 \texttt{ | GT -> } \square_3,\ P_2 = \texttt{compare } \square_1\ \square_2\}$$

P_2 is closer to the LLM solution $P_{\mathcal{L}}$ than P_1, with dist$(P_1, P_{\mathcal{L}}) = 3$ and dist$(P_2, P_{\mathcal{L}}) = 0$ ($\because$ $P_2 \sqcup^*$ `compare x y` = `compare` ($\square_1 \sqcup^* x$) ($\square_2 \sqcup^* y$) = `compare` $\square_1\ \square_2$, introducing no fresh variables).

Hence, Pick selects P_2 and refines it by filling $\square_1$ and $\square_2$ with components from **C** or other partial programs under grammar G. Suppose $\square_1$ is instantiated with x and y, producing P_{21} = `compare x` $\square_2$ and P_{22} = `compare y` $\square_2$. Their distances are dist$(P_{21}, P_{\mathcal{L}}) = 0$ and dist$(P_{22}, P_{\mathcal{L}}) = 1$. Since both are closer to $P_{\mathcal{L}}$ than P_1, they are prioritized in subsequent iterations of the top-down search.

As this process continues, all candidates derivable from P_2 are eventually explored without discovering the desired solution (after finitely many iterations, since the search space is bounded). At that point, the queue Q contains only P_1. The Pick function then selects P_1 and proceeds to refine it by filling the holes $\square_1$, $\square_2$, and $\square_3$ using components from **C** and other partial programs. Ultimately, the desired solution is obtained by instantiating these holes with y, x, and x, respectively.

Thus, the Pick function favors partial programs structurally similar to the LLM solution $P_{\mathcal{L}}$ (e.g., P_2), yet preserves exhaustiveness of the top-down search by eventually exploring structurally different candidates (e.g., P_1).

5 Implementation

5.1 The Underlying Synthesis Tool

We applied our approach to Trio, a tool that synthesizes recursive functional programs from input-output examples [24]. Trio takes algebraic data-type definitions, a library of usable functions, and input-output examples, and synthesizes a recursive program consistent with those examples.

The DSL underlying Trio resembles core ML, supporting algebraic data types, pattern matching, and higher-order functions (the full DSL appears in the prompt template later in this section). We implemented our Pick and BottomUp functions on top of Trio and built GuideSyn,[4] which guides Trio using LLMs. Trio normally ranks candidates using a syntactic cost function and performs standard bottom-up enumeration. We replace these mechanisms with our Pick function, which selects the candidate most similar to the LLM solution $P_{\mathcal{L}}$, and our BottomUp function, which generates components guided by $P_{\mathcal{L}}$ using a gradually increasing δ.

5.2 LLMs and Prompting

For LLM solutions, we use the OpenAI GPT-4o-mini [44], a lightweight general-purpose model, GPT-4o, a more capable general-purpose model, and o3-mini, a reasoning model that performs better on reasoning tasks such as programming, DeepSeek-Coder-V2 [53], an open-source model fine-tuned for code generation, and Gemini-2.5-Flash [11], a cost-efficient closed-source model. Our goal of using these models is understanding the impact of quality of LLM solutions on the performance of our algorithm.

When invoking the LLM, we use the following prompt template:

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}`

`Types and helper functions`, `Type`, and `Input-output examples` are filled with the algebraic data-type definitions, the target function's type, and the examples, respectively. We use a one-shot structured prompt [4] that assigns the model a code-generation role and constrains outputs to the Trio DSL, leveraging role-based [45] and grammar-based [48] prompting to ensure syntactic validity. A natural question is why we do not ask LLMs to generate code in more high-resource functional languages such as OCaml or Haskell and then translate it to the Trio DSL. The issue is that such translation is unreliable: in our experiments, GPT-4o-mini, GPT-4o, and o3-mini produced nontranslatable OCaml code for 28, 28, and 34 of the 80 tasks, often using unsupported operators, external libraries,

[4] Tool is available at https://github.com/pslhy/guide_syn_artifact

or syntax (e.g., nested function definitions). Thus, we ask LLMs to generate code directly in the Trio DSL, which works far more robustly in practice.

6 Evaluation

We evaluate GuideSyn to answer the following research questions:

RQ1: How effective is our approach in guiding program synthesis across different LLMs?
RQ2: How do our top-down and bottom-up guidance methods contribute to synthesis performance individually and in combination?
RQ3: How does GuideSyn perform if it is equipped with other possible alternative guidance methods?

All of our experiments were conducted on Linux machines with Intel Xeon 2.9GHz CPUS, 256GB of memory and NVIDIA RTX A6000x2 GPUs, running Ubuntu 20.04.6 LTS. We set the timeout limit to 120 seconds for each synthesis task.

6.1 Experimental Setup

Benchmarks. We use **80** tasks of synthesizing recursive functional programs. 60 out of 80 tasks are from the evaluation benchmark of Trio [24], and we devise 20 additional tasks. We added new tasks because the original 60 were too easy for unguided Trio (59/60 solved). To better evaluate GuideSyn on harder problems requiring LLM guidance, we selected 20 additional tasks from a functional programming course at our institution, choosing only assignments without publicly available solutions to avoid contamination. These tasks involve recursion, higher-order functions, and data-structure manipulation (lists, trees, automata). The detail can be found in the supplementary material.

Tool Setup. We have implemented GuideSyn on top of Trio [10, 24]. [5] For each task, we run Trio for a second, and if it does not finish, we stop it considering the problem possibly challenging. [6] We then invoke an LLM to generate a solution candidate. We prompt each LLM until it generates a correct solution up to a maximum of 3 attempts. [7] Whenever an LLM generates a syntactically wrong program, we ask the LLM to regenerate a solution by pointing out the syntax error. If the LLM generates a syntactically correct program but it does not satisfy the input-output examples, we provide input-output examples violated by the program as feedback to the LLM asking it to regenerate a solution. This feedback resembles counterexample-guided inductive synthesis (CEGIS) except that we do not use a synthesizer but rather use the LLM to generate a solution candidate at each iteration. There is no guarantee that the feedback will lead to a better solution. So we use the best solution generated during the 3 attempts. As LLMs, as already mentioned in Section 5, we use GPT-4o-mini, GPT-4o, and o3-mini, DeepSeek-Coder-V2, and Gemini-2.5-Flash as LLMs. [8] Lastly, we set the threshold δ to 5, meaning that only components with a distance of 5 or less from the LLM solution are considered during the bottom-up search.

[5] Trio employs a block-based pruning technique that discards infeasible partial program candidates. This pruning is *eventually sound*: it may initially prune feasible candidates with insufficient components, but preserves all feasible ones once enough components are generated. We disable this pruning in our implementation to avoid interactions with our guidance methods and focus on their pure effect.
[6] Prior synthesis work shows most successes occur quickly: Trio 91% [24], Burst 80% [30], Smyth 87% [29] within 1s. Raising the cutoff (e.g., to 30s) barely changes the "challenging" set but significantly increases runtime. Switching to LLM-guided synthesis after 1s is therefore cost-effective.
[7] We set the maximum number of feedback iterations to be 3, which we found to offer the best trade-off between performance and cost. Further iterations yield marginal benefit, consistent with prior findings [12, 15].

Methods	Models	# Solved (Variance)	# LLM Calls	Time (s)
LLM Only	gpt-4o-mini	48.8 (0.2)	152.2	2.8
	gpt-4o	62.4 (2.3)	133.6	2.3
	o3-mini (gpt)	68.2 (6.2)	139.0	18.3
	DeepSeek-Coder-V2	39.6 (5.2)	168.8	9.3
	gemini-2.5-flash	67.8 (12.7)	129.2	26.0
ThinkRepair	w/ gpt-4o-mini	51.4 (2.8)	289.2	18.7
	w/ gpt-4o	66.8 (4.2)	247.0	24.4
	w/ o3-mini (gpt)	71.2 (1.9)	233.2	100.2
	w/ DeepSeek-Coder-V2	41.4 (2.3)	303.6	52.6
	w/ gemini-2.5-flash	69.2 (3.2)	266.6	149.1
GuideSyn	w/ gpt-4o-mini	71.6 (1.3)	58.4	2.9
	w/ gpt-4o	74.0 (3.5)	42.5	3.6
	w/ o3-mini (gpt)	78.4 (0.8)	45.0	8.4
	w/ DeepSeek-Coder-V2	66.0 (1.0)	65.4	8.4
	w/ gemini-2.5-flash	76.6 (1.3)	43.4	10.8
Unguided	–	68.0 (–)	–	3.7

Table 1: Evaluation of LLM only, GuideSyn, and Unguided methods (averaged over 5 runs, 80 benchmarks).

Baselines. We compare GuideSyn with the following baselines:

- LLM-Only-`model`: These baselines use LLMs as standalone synthesizers. We prompt each model up to three times per task, and tasks unsolved after three attempts are marked as failures. Solving time is measured as the latency for each model to generate a solution, but since this depends on provider-specific serving conditions, the reported times are approximate.
- Unguided: We use the Trio program synthesizer as is without any LLM guidance.
- ThinkRepair: We use ThinkRepair [50], a state-of-the-art LLM-based automated program repair tool. We use it to repair incorrect programs generated by LLM-Only for each synthesis task. Because ThinkRepair usually takes long time to repair a program, we let it run until it exhausts five repair attempts per task without a time limit.

6.2 Effectiveness of GuideSyn

Table 1 summarizes the results. Each experiment is repeated 5 times, and we report the average number of solved tasks with variance in parentheses. The results show that the synergistic combination of the existing synthesizer and LLMs is more effective than using either alone. Unguided solves **68** out of **80** tasks, while LLM-Only with gpt-4o-mini, gpt-4o, o3-mini, DeepSeek-Coder-V2, and gemini-2.5-flash solves **48.8**, **62.4**, **68.2**, **39.6**, and **67.8** tasks on average, respectively. In contrast, GuideSyn with the same LLMs solves **71.6**, **74.0**, **78.4**, **66.0**, and **76.6** tasks, consistently outperforming both LLM-Only and Unguided across most models. The only exception is DeepSeek-Coder-V2 (relatively outdated), where GuideSyn is slightly less effective than Unguided due to the lower quality of LLM outputs. Compared to Unguided, GuideSyn solves **3.6**, **6.0**, **10.4**, and **8.6** more tasks with gpt-4o-mini, gpt-4o, o3-mini, and

[8] We use models with the temperature set to 1.0, which is the default value.

gemini-2.5-flash, respectively. Furthermore, GuideSyn significantly surpasses the state-of-the-art LLM-based repair tool ThinkRepair, solving **20.2**, **7.2**, **7.2**, **24.6**, and **7.4** more tasks with the five LLMs, respectively. These results indicate that simple repair of LLM outputs is insufficient: LLM-generated incorrect programs are often too far from the target to fix directly. By contrast, GuideSyn 's synthesis engine bridges this gap, discovering correct programs beyond the reach of repair-only methods.

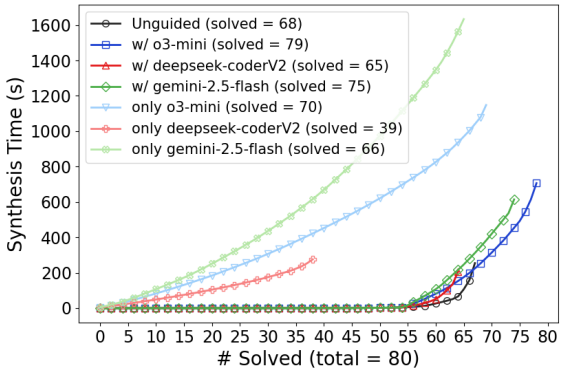

Figure 3: Comparison between GuideSyn, LLM-Only, and Unguided (with GPT-4o-mini and 4o omitted for clarity).

Fig. 3 shows the *cactus plot* of the cumulative solving time of each synthesizer, with the x-axis representing the number of problems solved and the y-axis representing cumulative solving time in seconds. A line closer to the x-axis indicates better performance. GuideSyn with LLMs outperforms LLM-Only in both the number of solved tasks and solving time, demonstrating synergy between the synthesizer and LLMs: easy tasks are solved quickly by the synthesizer, while harder ones are asked to LLMs. When LLMs fail, our guided synthesizer can still solve most tasks using their incorrect solutions. GuideSyn also surpasses Unguided, except with o3-mini, where it is slightly slower due to longer response time.

Answer to Q1: Our approach is effective in guiding program synthesis across different LLMs. It enables to solve more tasks than the synthesizer alone, the LLMs alone, and the state-of-the-art LLM-based program repair tool no matter which LLM is used.

6.3 Ablation Study

We evaluate how much benefit GuideSyn gains from the top-down and bottom-up guidance methods using OpenAI models. With only top-down guidance, GuideSyn solves **2**, **4**, and **9** more tasks than Unguided using 4o-mini, 4o, and o3-mini, respectively. With only bottom-up guidance, the improvements are **0**, **3**, and **8** tasks. When both methods are combined, GuideSyn achieves gains of **5**, **8**, and **11** tasks, respectively.

Answer to RQ2: Both of the guidance methods are crucial for the performance of GuideSyn, and the synergy between the two guidance methods is effective in solving more tasks.

6.4 Qualitative Analysis of LLM Guidance

To see how GuideSyn benefits from LLM guidance, we examine two representative examples that highlight the effectiveness of each guidance method.

Methods	# Solved			Time (s)		
	4o-mini	4o	o3-mini	4o-mini	4o	o3-mini
Unguided	68	68	68	3.7	3.7	3.7
+ Top-Down Guidance	70 (↑ **2**)	72 (↑ **4**)	77 (↑ **9**)	2.7	1.5	7.9
+ Bottom-Up Guidance	68 (↑ **0**)	71 (↑ **3**)	76 (↑ **8**)	2.0	1.6	17.1
+ Both Guidance	73 (↑ **5**)	76 (↑ **8**)	79 (↑ **11**)	3.1	2.4	8.9

Table 2: Ablation Study on Guidance Methods.

Top-down Guidance. For the `expr_b` task where the goal is to synthesize a function `f` that evaluates logical formulas, the LLM (GPT-4o-mini) generates the following incorrect program:

```
let rec f (x : formula) : bool =
  match x with
  | BOOL b -> b
  | NOT p -> not (f p)
  | ANDALSO (p, q) -> (f p) && (f q)
  | ORELSE (p, q) -> (f p) || (f q)
  | IMPLY (p, q) -> (f p) || (f q)
```

The incorrect line (in red) should be `not (f p) || (f q)`. While the unguided version of GuideSyn fails to find the solution within the time limit, However, with top-down guidance, GuideSyn quickly converges to a partial program candidate with the `IMPLY` branch as a hole and the rest filled in as per the LLM output because the candidate is structurally very similar to the LLM-generated program. Then, GuideSyn fills the hole to produce intended solution.

Bottom-up Guidance. Even when the LLM-generated solution is structurally dissimilar to the intended program, GuideSyn can still synthesize the correct solution through bottom-up guidance. For example, in the `list_zigzag` task, where the goal is to synthesize a function `f` that generates a list by alternating elements from two input lists `x` and `y`, LLM (GPT-4o-mini) produces an incorrect program with a wrong recursive call: `x::(f ys xs)` instead of the correct `x::y::(f xs ys)`, where `xs` and `ys` are the tails of `x` and `y`, respectively. Our bottom-up guidance enables GuideSyn to generate the correct recursive call by mutating the LLM-generated subexpression (the distance between `x::(f ys xs)` and `x::y::(f xs ys)` is 3, within the similarity threshold δ=5). Thus, GuideSyn efficiently generates necessary components that are absent in the LLM output but are close variants. In the unguided version, GuideSyn should have generated all the components up to the certain size to discover the correct recursive call, which makes the overall synthesis inefficient due to the large number of components.

Methods	# Solved	
	4o-mini	4o
GuideSyn	73	76
GuideSyn-RTED	62 (↓ **11**)	70 (↓ **6**)
GuideSyn-PCFG	67 (↓ **6**)	67 (↓ **9**)
GuideSyn-NoMutation	59 (↓ **14**)	64 (↓ **12**)

Table 3: Evaluation of Other Possible Alternatives.

6.5 Other Possible Alternatives

In this section, we evaluate alternative guidance methods to justify our design choices. Table 3 summarizes the results. To assess the cost-effectiveness of the anti-unification-based distance metric, we compare GuideSyn with GuideSyn-RTED, which uses RTED [36], a well-known tree edit distance algorithm. RTED has a worst-case complexity of $O(n^3)$, whereas our method runs in $O(n)$ (n is the size

of the trees). Although RTED provides more accurate distance estimation, it is computationally expensive. As shown in Table 3, with GPT-4o-mini and GPT-4o, GuideSyn-RTED performs worse than GuideSyn and even than Unguided when using GPT-4o-mini. This confirms that our anti-unification-based metric achieves a better balance between accuracy and efficiency, effectively guiding synthesis. Next, to test whether previous LLM-guided synthesis methods work in our setting, we compare GuideSyn with GuideSyn-PCFG, which uses a probabilistic context-free grammar (PCFG) trained on 100 LLM-generated solutions for each task [27, 28]. The trained PCFG guides the top-down search by always selecting the highest-probability candidate. However, GuideSyn-PCFG performs worse than even Unguided. Most of its time (about 69 seconds per task) is spent generating 100 candidate programs. In addition, the PCFG approach is not effective in our setting because PCFGs cannot capture contextual information around a production position and thus cannot provide accurate guidance. They give a fixed probability to each production, no matter where it is used in the program, which is not suitable in practical synthesis tasks where the context matters. Lastly, we study the impact of mutating LLM-generated components during bottom-up search. We compare GuideSyn with a variant of setting δ=0 (GuideSyn-NoMutation) with GPT-4o-mini, GPT-4o, and o3-mini. It solves 59, 64, 75 tasks−14, 12, and 4 fewer than with δ=5. Hence, mutation is critical.

Answer to RQ3: Our anti-unification-based distance metric is more cost-effective than the widely used tree edit distance, and our method is more effective than prior approaches that guide synthesis using PCFGs trained on multiple LLM-generated solutions or by directly reusing LLM-generated components without mutation.

7 Related Work

Distance Metrics in Program Synthesis & Repair. Distance metrics have been used in program synthesis and repair to measure program similarity. GiantRepair [26] constructs patch skeletons from LLM-generated patches and instantiates them with in-scope elements (e.g., variables, methods), prioritizing candidates close to the LLM patches using edit distance. Its patch generation is structure-preserving-it may change the content within a skeleton (e.g., the condition of an if-statement) but not the overall structure. In contrast, GuideSyn can discover solutions structurally different from LLM outputs (e.g., Fig. 2a introduces a new innermost match absent in Fig. 2b). This is enabled by synthesizing from scratch while using distance to the LLM solution as a soft guide, rather than modifying the LLM output directly. Rahmani et al. [41] extracts components from LLM-generated solutions and constructs larger programs by combining them using the Hamming, Euclidean, and Levenshtein distances to discard candidates too far from the LLM solutions during its beam search. Because those distance metrics are used for measuring similarity of operator usages only (i.e., which operators are used and how many times), they cannot capture structural similarity between programs as we do. SyMetric [17] and Bester [38] also use distance metrics for program synthesis, but only to cluster or rank candidates, not to guide search.

LLM-Guided Program Synthesis. Recently, LLMs have been used to guide the search in program synthesis [2, 13, 27, 28]. Li et al. [27, 28] infer a PCFG from multiple LLM-generated solutions and use it to guide the search. However, as shown in our evaluation, this approach is ineffective in our setting because PCFGs cannot capture contextual information around production sites and thus cannot provide accurate guidance. Moreover, generating multiple LLM solutions incurs significant computational cost. Hysynth [2] adopts a similar approach by learning a PCFG from multiple LLM solutions, but applies it to bottom-up enumeration, suffering from the same limitation. DryadSynth [13] asks an LLM to generate useful subexpressions and uses them as reusable components during its bottom-up search. Rahmani et al. [41] mine useful initial components from LLM-generated solutions to guide its component-based synthesis. All these approaches rely directly on multiple LLM outputs, making them less robust to noise or deviations from the intended solution. Our bottom-up guidance, in contrast, generates variants of LLM subexpressions guided by distance to the LLM solution, making it more robust to noise and more effective, as shown in our experiments with δ=0 and δ=5 in Section 6.5. For synthesizing *semantic regular expressions* from examples, Smore [8] combines LLM-generated sketches with search-based completion, re-invoking the LLM when a sketch is infeasible, whereas our approach calls the LLM only once at synthesis start, reducing LLM queries. For synthesizing robot policies from demonstrations, Prolex [35] applies LLM guidance only to top-down synthesis, whereas our method guides bidirectional, which are equally important as shown in our evaluation. Our synthesis approach from imperfect LLM solutions can be viewed as a generate-and-repair approach [49], where the LLM provides an initial candidate and the synthesizer repairs it.

Anti-Unification for Structural Similarity. Anti-unification [40, 42] has been widely used to analyze structural similarity in code clone detection [5, 22, 47] and program repair [43]. To our knowledge, our work is the first to employ anti-unification to guide search in program synthesis, whereas Babble [7] uses it only to cluster similar programs for library learning.

8 Threats to Validity

We discuss potential threats to the validity of our approach and strategies to mitigate them.

First, our method assumes LLM-generated programs are reasonably close to the correct solution; its effectiveness may drop when they deviate significantly. In this case, increasing the distance threshold δ may help explore more diverse candidates, at the cost of a larger search space and longer synthesis time.

Second, LLMs sometimes produce invalid DSL programs in terms of syntax or types. We allow up to three retries, but if invalid code persists, the system falls back to unguided synthesis. We expect this issue of invalid outputs can be mitigated via constrained decoding techniques that ensure syntactic [34] and type [32] correctness.

Acknowledgments

We thank the reviewers for their insightful comments that helped us improve the paper. This work was supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (No. RS-2021-NR060080) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (Nos. 2022-0-00995, RS-2024-00341722, RS-2024-00423071).

References

[1] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[2] Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. 2024. Hysynth: Context-free llm approximation for guiding program synthesis. *Advances in Neural Information Processing Systems* 37 (2024), 15612–15645.

[3] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[5] Peter Bulychev and Marius Minea. 2009. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*. 54–55.

[6] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (Jan. 2023), 30 pages. doi:10.1145/3571226

[7] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 396–424.

[8] Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Işıl Dillig. 2023. Data extraction via semantic regular expression synthesis. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1848–1877.

[9] Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. 2023. Fast and Reliable Program Synthesis via User Interaction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 963–975. doi:10.1109/ASE56229.2023.00129

[10] HANGYEOL CHO and WOOSUK LEE. 2025. Inductive synthesis of structurally recursive functional programs from non-recursive expressions. *Journal of Functional Programming* 35 (2025), e17. doi:10.1017/S0956796825100063

[11] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[12] Dekun Dai, MingWei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. 2025. FeedbackEval: A Benchmark for Evaluating Large Language Models in Feedback-Driven Code Repair Tasks. *arXiv preprint arXiv:2504.06939* (2025).

[13] Yuantian Ding and Xiaokang Qiu. 2024. Enhanced enumeration techniques for syntax-guided synthesis of bit-vector manipulations. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2129–2159.

[14] Yuantian Ding and Xiaokang Qiu. 2025. A Concurrent Approach to String Transformation Synthesis. *Proc. ACM Program. Lang.* 9, PLDI, Article 233 (June 2025), 25 pages. doi:10.1145/3729336

[15] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering* (2024).

[16] Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 244–259. doi:10.1145/3519939.3523726

[17] Jack Feser, Işıl Dillig, and Armando Solar-Lezama. 2023. Inductive program synthesis guided by observational program similarity. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 912–940.

[18] Qiantan Hong and Alex Aiken. 2024. Recursive Program Synthesis using Paramorphisms. *Proc. ACM Program. Lang.* 8, PLDI, Article 151 (June 2024), 24 pages. doi:10.1145/3656381

[19] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. doi:10.1145/1806799.1806833

[20] Ruyi Ji, Yuwei Zhao, Nadia Polikarpova, Yingfei Xiong, and Zhenjiang Hu. 2024. Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis. *Proc. ACM Program. Lang.* 8, PLDI, Article 185 (June 2024), 26 pages. doi:10.1145/3656415

[21] Dongkwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2023. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Time-bounded Exhaustive Search. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 16 (Sept. 2023), 37 pages. doi:10.1145/3591622

[22] Hyo-Sub Lee and Kyung-Goo Doh. 2009. Tree-pattern-based duplicate code detection. In *Proceedings of the ACM first international workshop on Data-intensive software management and mining*. 7–12.

[23] Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

[24] Woosuk Lee and Hangyeol Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. *Proc. ACM Program. Lang.* 7, POPL, Article 70 (jan 2023), 31 pages. doi:10.1145/3571263

[25] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. *ACM Sigplan Notices* 53, 4 (2018), 436–449.

[26] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2025. Hybrid automated program repair by combining large language models and program analysis. *ACM Transactions on Software Engineering and Methodology* 34, 7 (2025), 1–28.

[27] Yixuan Li, José Wesley de Souza Magalhães, Alexander Brauckmann, Michael FP O'Boyle, and Elizabeth Polgreen. 2025. Guided tensor lifting. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1984–2006.

[28] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding enumerative program synthesis with large language models. In *International Conference on Computer Aided Verification*. Springer, 280–301.

[29] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[30] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. doi:10.1145/3498682

[31] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-driven inference of representation invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3385412.3385967

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

[33] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

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

[35] Noah Patton, Kia Rahmani, Meghana Missula, Joydeep Biswas, and Işıl Dillig. 2024. Programming-by-demonstration for long-horizon robot tasks. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 512–545.

[36] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: a robust algorithm for the tree edit distance. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 334–345. doi:10.14778/2095686.2095692

[37] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.* 40, 1, Article 3 (March 2015), 40 pages. doi:10.1145/2699485

[38] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the enemy of good: Best-effort program synthesis. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2–1.

[39] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1114–1124. doi:10.1145/3180155.3180189

[40] Gordon D Plotkin. 1970. A note on inductive generalization. *Machine intelligence* 5, 1 (1970), 153–163.

[41] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[42] John C Reynolds. 1970. Transformational systems and the algebraic structure of atomic formulas, Machine Intelligence 5.

[43] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2018. Learning quick fixes from code repositories. *arXiv preprint arXiv:1803.03806* (2018).

[44] John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, et al. 2022. Chatgpt: Optimizing language models for dialogue. *OpenAI blog* 2, 4 (2022).

[45] Murray Shanahan, Kyle McDonell, and Laria Reynolds. 2023. Role play with large language models. *Nature* 623, 7987 (2023), 493–498.

[46] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 287–296. doi:10.1145/2491956.2462174

[47] Wim Vanhoof and Gonzague Yernaux. 2019. Generalization-driven semantic clone detection in clp. In *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 228–242.

[48] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. 2023. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems* 36 (2023), 65030–65055.

[49] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[50] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1274–1286.

[51] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI, Article 174 (June 2023), 25 pages. doi:10.1145/3591288

[52] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 860–883.

[53] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931* (2024).