Question Selection for Multi-Modal Code Search Synthesis using Probabilistic Version Spaces

Jiarong Wu, Yanyan Jiang, Lili Wei *Member, IEEE*, Congying Xu, Shing-Chi Cheung *Fellow, IEEE*, Chang Xu *Senior Member, IEEE*

***Abstract*—Searching the occurrences of specific code patterns (code search) is a common task in software engineering, and programming by example (PBE) techniques have been applied to ease customizing code patterns. However, previous PBE tools only synthesize programs meeting the input-output examples, which may not always align with the user intent. To bridge this gap, this paper proposes EXCALIBUR, a multi-modal (example and natural language description) and interactive synthesizer for code search. EXCALIBUR ensures that the generated programs are correct for the provided examples (soundness) and include the user-intended program (bounded completeness). Furthermore, EXCALIBUR helps the user identify the user-intended program through question-answer interaction. To minimize the required interaction efforts, question selection is crucial. To improve question selection for code search, we propose probabilistic version spaces (ProbVS), in which the user-intended program's probability is high and others are low. ProbVS combines traditional version spaces for compactly representing extensive programs and large language models (on the user-provided natural language description) for adjusting programs' probabilities to align with users' intents. Extensive experiments on a benchmark of 44 tasks demonstrated the effectiveness of EXCALIBUR and ProbVS and demystified how ProbVS affects probability distributions and how the configurable parameters affect ProbVS.**

***Index Terms*—Program synthesis, question selection problem, large language model**

I. INTRODUCTION

CODE search/linting tools such as CODEQL [1], Semgrep, and Spoon [2] are popular nowadays. They aim to detect the occurrences of specific code patterns, especially those related to code smells, bugs, or security issues, in a given piece of code. In this paper, we call such an occurrence detecting task *code search*, which identifies code fragments matching a target code pattern. Despite the success of existing code search/linting tools, many code patterns are application-specific and may not be readily supported. For example, preparing a code search program that identifies invocations of a later-written erroneous API is impossible. In most cases, extra implementation effort is required.

Jiarong Wu and Congying Xu and Shing-Chi Cheung are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China. Email: {jwubf, cxubl}@connect.ust.hk, scc@cse.ust.hk.

Lili Wei is with the Department of Electrical and Computer Engineering, McGill University, Montreal, Canada. Email: lili.wei@mcgill.ca.

Yanyan Jiang and Chang Xu are with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China. Email: {jyy, changxu}@nju.edu.cn.

Yanyan Jiang and Shing-Chi Cheung are the corresponding authors.

While some code search tools provide libraries to facilitate the detection of new code patterns, the implementation is still non-trivial because it demands expertise in both the target programming language (e.g., Java) and the code search libraries (e.g., CODEQL and Spoon).

Previous Work. ALICE [3] and SPORQ [4] support the use of programming by example (PBE) [5], [6] to help users find code snippets that match a target pattern in a codebase (e.g., a repository). Specifically, a user needs to provide a codebase and a few code snippet examples matching the target pattern. A code search program is then synthesized and refined in multiple iterations. Each iteration consists of three steps: (a) the synthesizer finds a tentative program that implements a pattern conforming to the examples provided by the user. (b) the program is executed to find code snippets from the codebase as speculative examples. (c) the user labels these examples as positive or negative, which are used to refine the tentative program in the next iteration. As the labeled examples accumulate, the refined program approaches the intended one.

However, there is no guarantee that the synthesis procedure will find the target program. For example, in the worst case of ALICE's evaluation, only 66% of target programs are found. Our replication of SPORQ also produced overfitting results in our benchmark. Consider the motivating example in Figure 1 that aims to identify `toString` and `hashCode` calls on array-type objects in Java, which are common pitfalls (behaving based on memory locations rather than element values). A program overfitting the provided example is extracting method calls without arguments, which is by chance example-consistent but deviates from the user intent. When such overfitting code search programs are incorporated as subroutines of code analysis [7], [8], the soundness or completeness of the outer analysis could be compromised.

Interactive Synthesis. A promising approach to addressing the overfitting issue is interactive synthesis via a question-answer (QA) loop [9], [10], [11]. At each iteration/round of the loop, a question disambiguating the target program from others is generated, and the user as the oracle provides the target program-pertinent answer, which filters out answer-inconsistent programs. For example, a question to distinguish $\lambda x.x + 2$ from $\lambda x.x * 2$ can be the expected output of $x = 3$. The QA interaction loops until only the target program or equivalent ones are left.

Challenge. The major challenge in QA-based interactive synthesis is to reduce the number of required interaction rounds. As the number of candidate programs could be

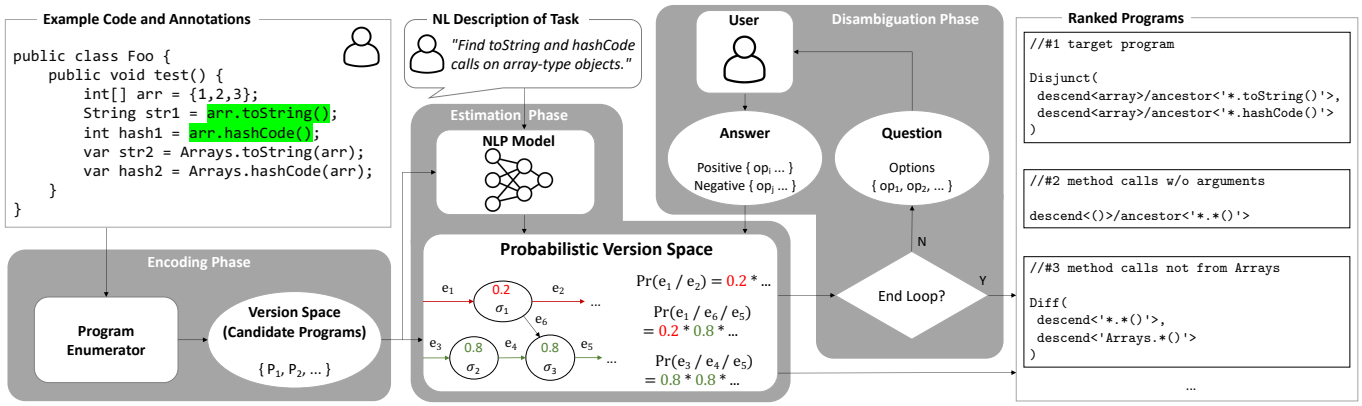


Fig. 1: Workflow of EXCALIBUR on an example code search task that aims to extract all `toString` and `hashCode` calls on array-type objects. Frames in shade denote three phases of the EXCALIBUR workflow. The encoding phase enumerates example-consistent candidate programs in a data structure called a version space. Then, the estimation phase leverages an NLP model to estimate which programs in the version space align with the user-provided NL description and produces a probabilistic version space (ProbVS), which describes a probability distribution of candidate programs. Lastly, the disambiguation phase embodies a QA-based interactive synthesis procedure. Candidate programs are ranked after each interaction round.

tremendous (e.g., up to hundreds of thousands in our benchmark), without appropriate treatment, it may take an overwhelming number of interaction rounds, hindering the deployment of interactive synthesis in practice.

The key factor determining the number of interaction rounds is the questions being asked, and finding questions to minimize the expected number of interaction rounds (probabilities are assigned to candidate programs) is known as the *question selection* problem [10], [11]. The state-of-the-art solutions [10], [12] encode the program operators and the question domain into constraints and leverage an SMT solver [13] to efficiently find the optimal question. However, encoding the ASTs in general-purpose programming languages (e.g., Java) presents a significant challenge in efficiency due to the enormous choices of productions and the complex constraints encoded from the recursive and expressive grammar. In addition, an encoding for general Java programs' semantics is also not available off-the-shelf.

Our Solution. This paper proposes a code search synthesizer called EXCALIBUR, which is *sound*, *bounded-complete*, and supports QA-based *interaction*.

We take Figure 1 to illustrate the key features of EXCALIBUR. EXCALIBUR is multi-modal, i.e., the inputs to EXCALIBUR include two forms of specifications:

1) an input-output example, where the example input is a code snippet and the example output is a set of code fragments annotated in the input code snippet, e.g., highlighted parts in Figure 1.
2) a natural language (NL) description of the target code pattern, e.g., "`toString` and `hashCode` calls on array-type objects".

The example specification ensures the synthesized programs can pass this "testcase", and the NL specification provides extra information to help identify the target program.

Given such specifications, EXCALIBUR first synthesizes all example-consistent programs (or simply consistent programs) within a given search budget (e.g., an upper bound of program length), including the target program and overfitting programs (the encoding phase in Figure 1). EXCALIBUR is *sound* since only example-consistent programs would be synthesized. Moreover, because *all* consistent programs are considered, EXCALIBUR is *bounded-complete* [1]. These properties cannot be guaranteed by end-to-end code generation from large language models (LLMs), particularly, when generating code in domain-specific-languages (DSLs) [14] or domain-specific libraries [15], [16].

EXCALIBUR supports QA-based interactive synthesis in code search (disambiguation phase in Figure 1). We work around the limitation of an SMT solver by asking questions based on programs' runtime behaviors. Such questions are easier to synthesize with no dependence on SMT solvers. EXCALIBUR models a code search program as a step-by-step matching of code fragments. For example, the target program in Figure 1 should match array-type objects in the first step, and then their associated `toString` calls or `hashCode` calls in the next step. Therefore, programs' runtime behaviors (i.e., which code fragments are matched) can serve as questions for disambiguation. Consider the motivating example, a question can be whether an empty argument list `()` is matched/searched by the target program. With "No" answered, the overfitting program "method calls without arguments" can be excluded.

Probabilistic Version Spaces. The absence of an SMT solver also poses an efficiency challenge. In response, EXCALIBUR first adopts the SOTA question selection strategy, which optimizes the interaction efficiency given a probability distribution of candidate programs. Moreover, we observe that a strategy's input, the estimated probability distribution of programs, also has a great impact on the interaction efficiency. The previous works [10], [17] use a priori probabilities

[1]That means EXCALIBUR will not miss the target program if it is within the bounded search budget.

that are agnostic of the user tasks and may result in suboptimal question selection performance, while a user-aligned distribution (i.e., the target program is assigned a higher probability while others are lower) is expected to reduce the number of interaction rounds. In light of this observation, we propose a new methodology called *probabilistic version space* (ProbVS) to acquire a more user-aligned probability distribution.

Specifically, ProbVS requires a natural language description of the target code pattern, e.g., "Find toString and hashCode calls on array-type objects" (Figure 1). Then, the in-context learning capability of a large language model (LLM) is leveraged to estimate which code fragments are related to the task description. Combined with version spaces (a succinct representation of candidate programs), ProbVS can efficiently compute a user-aligned probability distribution of candidate programs (the estimation phase in Figure 1). How to model the programs' probabilities conditioned on their intermediate results in version spaces is unexplored before and one of our technical contributions.

To evaluate the effectiveness of EXCALIBUR, we implemented a prototype tool for Java. Evaluation results on our collected tasks show that EXCALIBUR without using ProbVS solves all 44 tasks with 4.5 rounds of interaction on average, and the target programs of 30 tasks (68.18%) are ranked top. As a comparison, the baseline that replicates SPORQ only solved at most 26 out of the 44 tasks (59.09%), with several being overfitted. The effectiveness of ProbVS is also confirmed. The best result of EXCALIBUR with ProbVS and GPT4o takes only 3.68 average rounds, reducing up to 0.82 (18.22%) average interaction rounds on top of the SOTA question selection technique. We also conducted experiments to demystify how ProbVS improves the performance and the impacts of configurable factors.

In summary, this paper's contribution is as follows.

1) We propose a multi-modal synthesizer EXCALIBUR for code search with the bounded-complete guarantee and the QA-based disambiguation.
2) We propose a methodology ProbVS that combines LLMs and version spaces to compute a probability distribution of candidate programs, which aims to align with the user's intent and improve the performance in the question selection problem.
3) We evaluate EXCALIBUR on a set of code search tasks and showed the overall effectiveness of EXCALIBUR in solving code search tasks as well as the effectiveness of ProbVS in reducing interaction rounds.

II. OVERVIEW

Figure 1 presents the workflow of EXCALIBUR, which takes a user-provided input-output example and natural language description as input and outputs a ranked list of programs, including the target one. EXCALIBUR works in three phases:

- *(Encoding Phase)*. EXCALIBUR encodes all programs consistent with the example into a succinct automaton [18] representation called a version space, which is constructed based on dynamic programming and avoids inefficient brute-force enumeration.

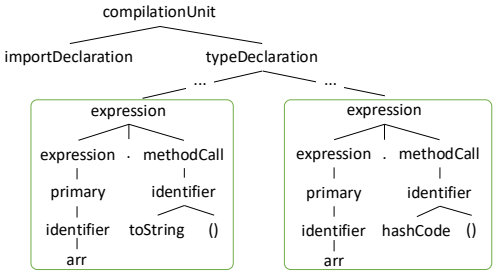


Fig. 2: The AST of the example code in Figure 1. The two framed subtrees are the ASTs of the example output code fragments `arr.toString()` and `arr.hashCode()`.

- *(Estimation Phase)*. An LLM is utilized to estimate the probabilities of specific program *features* in the target program, based on the natural language description provided by the user. Such a *probabilistic version space* (ProbVS) assign higher probabilities to programs with more features matching the LLM's estimations. These probabilities are used as guidance for question selection.
- *(Disambiguation Phase)*. Finally, to help identify the target program in the user's mind, EXCALIBUR iteratively synthesizes a question for the user to provide an answer to reduce the version space. When the programs in the reduced version space are no longer distinguishable by any question [2], the remaining programs are ranked and returned.

A. Encoding Phase: Encoding Consistent Programs

Example. Like other PBE techniques, EXCALIBUR requires a user to provide a code snippet as an example input to the target program and highlight code fragments as example outputs in the snippet. EXCALIBUR assumes the user provides all occurrences of the code fragments that will be outputted by the target program [3]. For instance, in Figure 1, the user provides a code snippet containing both the example outputs `arr.toString()` and `arr.hashCode()`.

ExPath Programs. EXCALIBUR is designed to synthesize programs for searching/matching code fragments in specific patterns, e.g., "toString calls on an array object" or "method definitions with non-empty parameter list". The matched code fragments in an input code snippet are typically [19], [20] subtrees of its abstract syntax tree (AST), as shown in Figure 2. In the remainder of this paper, we use the terms "code fragment" and "AST" interchangeably.

To express code patterns, we designed a simplified dialect of the XPath [21] query language called ExPath (EXCALIBUR's XPath). The basic construct of an ExPath program P is an XPath-like *path program* represented as a sequence of slash-separated path expressions e_i:

$$P = e_1/e_2/\ldots/e_n.$$

[2]A question can distinguish two programs if they correspond to different answers, e.g., different sizes when program length is asked. Given a question and a user answer, a version space is reduced by retaining programs whose answers match the user's.

[3]In the setting of EXCALIBUR, the input code example is usually short enough for the user to inspect all the occurrences.

Each e_i (path expression) performs a "single-step" pattern matching process. Like in XPath, a path expression e_i takes an AST subtree as input and selects its ancestor/descendant nodes (or even semantically related nodes) that match the pattern specified by e_i. For example, when evaluating the expression `descend<()>` in Figure 3, the `descend` keyword specifies matching against the input subtree's descendant nodes, and the code fragment `()` inside `<·>` specifies selecting nodes that are empty parentheses.

Path expressions can also be applied to multiple AST subtrees. In this case, the output is the union of performing the matching process to each AST subtree, e.g., evaluating `ancestor<'*.*()'>` in Figure 3 returns the two brackets' respective method calling ancestors. Without loss of generality, we let the input to a path expression be a set of AST subtrees (e.g., σ_i in Figure 3) in this paper.

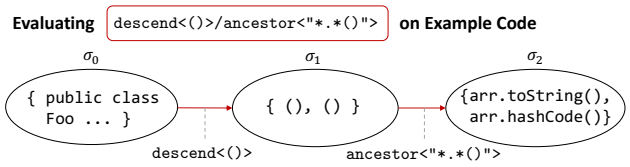


Fig. 3: A path program $P = e_1/e_2$ and its evaluation, where e_1 is `descend<()>` and e_2 is `ancestor<'*.*()'>`. Each σ_i denotes a set of AST subtrees.

Like in XPath, the slashes "/" chain a sequence of path expressions e_i for "multiple-step" pattern matching. Specifically, the evaluation of $e_1/\ldots/e_n$ can be understood as the lambda expression

$$\lambda\sigma_0.\ e_n(e_{n-1}(\ldots(e_1(\sigma_0)))),$$

where σ_0 denotes the input and is by default a singleton set containing only the example code snippet (the AST root). The step-by-step evaluation process can also be viewed as a cascading transition between different σ_i:

$$\sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \sigma_n.$$

(ExState). Hence, each set of code fragments σ_i represents an ExPath program's running state. We called such states ExStates. All code fragments in an ExState match a specific code pattern, which is specified by e_i and its predecessors $e_1, \ldots, e_{i-1}$.

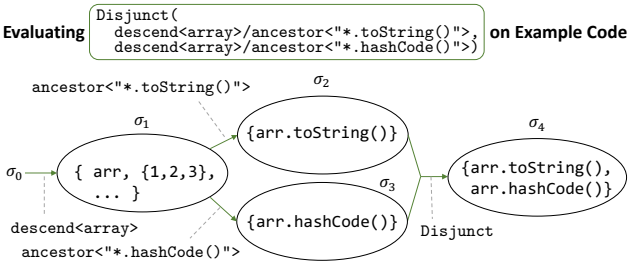


Fig. 4: Example set operation and its evaluation

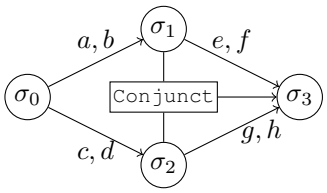
Set Operations. Furthermore, ExPath allows set operations `Conjunct` (∩), `Disjunct` (∪), and `Diff` (\) to be applied to path programs. This allows the expression of composite code patterns like "hashCode or (∪) toString calls". When evaluating a set operator, the subsidiary programs are first evaluated separately into the resulting ExStates, which are then fed to the set operator (e.g., union) to yield the final ExState, as shown in Figure 4.

Encoding Consistent Programs with Version Spaces. Given an input-output example, Excalibur encodes all programs (within a bound) consistent with the example via a structure called a version space [22], [5].

A version space in Excalibur is a graph with vertices denoting ExStates (e.g., vertices in Figure 4) and edges labeled by path expressions. Specifically, there must be an initial state denoting the example input and a target state denoting the example output. Path expressions on the same edge are *observationally equivalent* in evaluating the incoming state to the outgoing state, i.e., $e_1(\sigma_{in}) = e_2(\sigma_{in}) = \sigma_{out}$ for an edge like

$$\sigma_{in} \xrightarrow{e_1, e_2} \sigma_{out}.$$

The version spaces in Excalibur also allow directed hyper-edges $\langle \sigma_{in_1}, \sigma_{in_2} \rangle \rightarrow \sigma_{out}$ that take two positioned incoming states and one outgoing state. Such hyper-edges are labeled with set operators (e.g., `Conjunct`) to denote set operations in ExPath. For example, the below version space specifies that σ_3 is the intersection of σ_1 and σ_2.



A version space can be constructed in a breadth-first search way and is initialized with only the initial state. Iteratively, Excalibur enumerates all path expressions and set operators on all the states in the graph to find new states and update the states/edges on the graph, until a bound of the iterations is reached. More details are in Section IV-A.

The graph representation of a version space succinctly encodes multiple programs that input a specific initial state and output a specific target state. Consider the above version space example, each path from σ_0 to σ_3 represents a set of programs, i.e., all Cartesian products of expressions in different edges. For example, the path $\{a, b\} \rightarrow \{e, f\}$ represents four programs a/e, a/f, b/e, and b/f. As such, Excalibur avoids explicitly enumerating all example-consistent programs. The size of a version space (number of edges/states) is logarithmically smaller than the size of programs represented by the graph.

A version space can also accelerate the operations on the represented programs. For example, a common operation during the QA interaction of Excalibur is to follow the user's answer to mark an ExState as irrelevant to the target program and remove the programs relevant to the state. Removing all such programs can be achieved by simply removing the state from the graph because the paths pertinent

to the state and the programs constituted by these paths are automatically removed.

For the program synthesis problem in this paper, an algorithm is *sound* if all the synthesized programs satisfy the given input-output examples; an algorithm is *bounded-complete* if, for any program that satisfies the input-output examples and is within the bound of search budget, the program is included in the synthesized results.

Theorem II.1. *The synthesis algorithm in* EXCALIBUR *is sound and bounded-complete. The proof can be derived from the properties of version spaces and is detailed in the supplementary material [23].*

B. Estimation Phase: Assigning Probabilities to Programs

Thus far, all example-consistent programs in the version space are treated as equally likely to be the target program, but this does not align with the user's perspective. Most example-consistent programs exhibit behaviors that diverge from the natural language description of the target program. Therefore, we propose assigning probabilities to programs based on their alignment with the user's intent (NL description) to facilitate distinguishing the target program from others.

Probabilistic Version Spaces. Recall that each EXSTATE in version spaces encodes not only example-consistent programs but also the matched code fragments. Different code fragments have different correspondence with the user's NL description and can thus serve as features (like in decision tree) to estimate the corresponding programs' probabilities.

Specifically, EXCALIBUR uses code fragments in the states as features and exploits LLMs' in-context learning capability [24], [25] to classify whether a feature is related. Each feature is assigned a probability based on the classification result and the LLM's "confidence". Then, these probabilities propagate via the version space through states containing the features and along the state-pertinent edges/paths to estimate their constituted programs' probabilities. We exemplify the calculation instincts below and leave the details to Section IV.

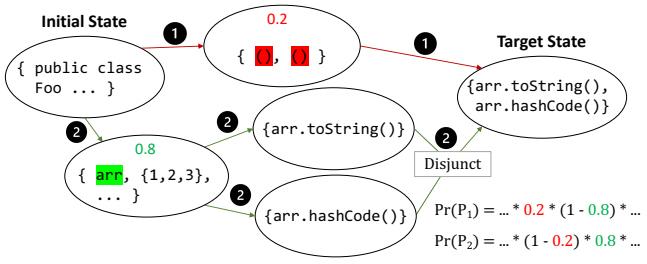


Fig. 5: Basic idea of ProbVS. ❶ and ❷ edges correspond to the program P_1 and P_2, respectively.

Take Figure 5 for example, suppose an empty argument list `()` and an array-type expression `arr` are the features. Because `()` is not related to the user intent, this code fragment is assigned a low probability like 0.2, while `arr` is relevant and assigned a high probability like 0.8. Then, the code fragments' probabilities propagate via the version space to calculate the consistent programs' probabilities. The basic idea is to model a program's probability in proportion to each code fragment's probability $\mathbb{P}$ (resp., $1-\mathbb{P}$) if the code fragment is ever (resp., never) matched by the program. For example, the overfitting program P_1, which matches `()` but not `arr`, is modeled with a proportion $0.2 \times (1-0.8)$. Likewise, the target program P_2, which matches `arr` but not `()`, is modeled with a proportion $(1-0.2) \times 0.8$.

Since the probabilities are propagated via the version space, we call this approach *probabilistic version spaces* (ProbVS). ProbVS exploits the correspondence between the NL description and the matched code fragments in an explicit task, so we considered ProbVS more fine-grained and user-aligned than simple heuristics like preferring shorter programs or assigning a priori probabilities at the grammar level [26] without considering the explicit context.

Estimating Code Fragments' Probabilities. EXCALIBUR estimates a code fragment's probability by prompting a large language model (LLM) a binary classification problem: whether the code fragment matches any part of the NL description. A simplified prompt[4] for the binary classification problem is shown below.

```
```
public class Foo {
    public void test() {
        int[] arr = {1, 2, 3};
        ...
    }
}
```
Is `Foo` related to the task "toString and hashCode calls on array-type objects" ? Just answer Yes or No.
```

The LLM's response contains a probability distribution on the LLM's token vocabulary, and EXCALIBUR assigns the probability of the label token (e.g., "Yes") to the code fragment. This process repeats for all/selected code fragments in the version space. Then, as shown in Figure 5, these probabilities propagate via ProbVS to yield a probability distribution of the example-consistent programs.

C. Disambiguation Phase: Interactive Program Synthesis

Although ProbVS provides a probability distribution of the candidate programs, the user's target program may not be ranked top due to the ambiguity in the NL specification. For example, in "toString and hashCode calls on arrays", it is unclear whether "arrays" refers to array-type objects or a variable named `arrays`. LLMs may also make wrong predictions, such as mistaking a statement like `foo.bar();` as a call expression due to overlooking the semicolon.

To further assist the user in efficiently identifying the target program, EXCALIBUR adopts the question-answer-based interactive synthesis framework [11], [10], which iteratively asks a question q to which different programs correspond to

[4]In practice, the prompt we adopt is more complex (available at [23]) and tries to guide the LLM to solve the task in a chain-of-thought way [27]. We also squeeze multiple code fragments to query in a single prompt in order to reduce the cost of querying LLMs.

different answers, and the user answer α (based on the target program) filters out inconsistent programs.

Question Space. In QA-based interactive synthesis, *questions* are drawn from a set of question candidates $\mathbb{Q}$ called the *question space*. Given a programming-by-example task to synthesize a program P, a natural way to construct the question space $\mathbb{Q}$ is to take the program inputs x as questions, where the user answers are the expected outputs $y = P(x)$.

Unfortunately, for code search tasks, finding inputs x (code snippets in industrial programming languages like Java) satisfying complex constraints (for distinguishing various ExPath programs) remains difficult, as discussed in Section I. Alternatively, EXCALIBUR asks questions in the following template (for novice users):

> When localizing your target code pattern step by step, is `<code fragment> <exactly | nested in>` an intermediate result? Each step takes in an intermediate code fragment f and returns its related code fragments that are:
> - AST ancestors or descendants with a specific syntactic structure, e.g., expression, statement;
> - AST ancestors or descendants with a specific semantic property, e.g., array-type, string-type, inherited methods; or
> - semantically related to f, e.g., callers/callees of f.

Or formally (for proficient users):

> When evaluating your target ExPath program step by step, would any EXSTATE contain any code fragment `<exactly | nested in> <code fragment>`?

This question template disambiguates ExPath programs by exploiting their runtime behaviors, i.e., whether `<code fragment>` or its nested fragments occur during the evaluation. To answer such questions, the user needs to "simulate" a step-by-step evaluation of the path program but does not need to know the exact path expressions in ExPath.

Consider the motivating example to search `toString` calls and `hashCode` calls on array-type objects. The users can simulate a matching procedure to first localize array-type objects (i.e., `arr` and `{1, 2, 3}`), followed by toString calls (i.e., `arr.toString()`) and hashCode calls (i.e., `arr.hashCode()`), even if the users do not know the exact path expressions. Then, when asked a question substituting an empty argument list "`()`" for `<code fragment>`, the answer is clearly "No", which then distinguishes the overfitting program (Figure 3) and the target program (Figure 4).

Question Selection. The question-answer procedure in EXCALIBUR iteratively narrows the range of the target program. Intuitively, the more "relevant" questions are selected for the user, the fewer rounds are expected for disambiguation. However, given a probability distribution of candidate programs (e.g., ProbVS), selecting the questions minimizing the expected number of rounds is shown NP-hard [10]. Hence, EXCALIBUR follows the SOTA approximation strategy called minimax-branch [10], [11] (detailed below).

EXCALIBUR asks the user k questions (e.g., 3 in our evaluation) in each round of interaction. Because each question $q \in \mathbb{Q}$ has a binary answer (yes or no), k questions as a *cluster* (denoted as Q) has at most k-bit information (2^k different choices of answers) for disambiguation.

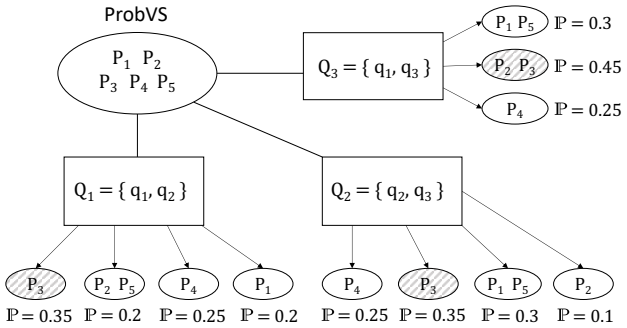


Fig. 6: An example of minimax-branch in EXCALIBUR. Each rectangle denotes a question cluster (Q_i). Each small oval denotes a branch (i.e., a subset of answer-consistent programs), and a shaded one denotes the max-branch under each corresponding question cluster.

(Branch). Each answer α has a corresponding subset of candidate programs consistent with α, and this subset is called a *branch*. A branch's probability $\mathbb{P}(\alpha)$ is the sum of the probabilities of programs in the branch, indicating how likely the answer α would be given by the user. Figure 6 depicts an example scenario selecting a question cluster (Q_i) comprising $k = 2$ questions (q_i) for 5 candidate programs (P_i), which fall in different branches with corresponding probabilities ($\mathbb{P}$).

Given all possible $\binom{|\mathbb{Q}|}{k}$ question clusters, the minimax-branch approximation strategy identifies the question cluster with a minimized "worst case answer":

$$Q^* = \operatorname*{argmin}_{Q} \max_{\alpha} \mathbb{P}(\alpha)$$

where $\max_\alpha \mathbb{P}(\alpha)$ denotes the probability of Q's max-branch, i.e., the branch with the highest probability. This branch corresponds to the worst-case answer because choosing a branch with a higher probability generally means taking more subsequent interaction rounds for disambiguation.

Take Figure 6 for example, the choice of Q_3 can be first discarded because it has a higher max-branch probability (0.45) than those of Q_1 (0.35) and Q_2 (0.35). When the max-branch probabilities are equal, the comparison continues with the second-largest branch, third-largest branch, etc. In this case, the second-largest probability under Q_1 (0.25) is smaller than that of Q_2 (0.3), so the final selection is Q_1.

We also exemplify how ProbVS improves the question-answer efficiency. Suppose ProbVS makes correct estimations, the target program is expected to have a higher probability than others. Then, the minimax-branch strategy is inclined to find a question (cluster) where the target program is in a branch with fewer other programs, so the reduction of remaining programs converges more quickly. For example, suppose P_2 is the target program and has a high probability 0.5. Both Q_1 and Q_3 put P_2 in a branch with other programs, so the max-branch probabilities are greater than 0.5; while Q_2 puts P_2 alone in one branch corresponding to a max-branch probability of 0.5. The minimax-branch strategy thus selects Q_2, allowing the user to identify the target program with only one interaction.

III. Preliminaries

We formulate the preliminary concepts in EXCALIBUR in this section, including the specification of the synthesis task and our DSL ExPath.

A. Multi-Modal Specification

The specification Φ in EXCALIBUR is multi-modal, i.e., $\Phi = \langle \Phi_{\mathcal{E}}, \Phi_{\mathcal{N}} \rangle$, where $\Phi_{\mathcal{E}}$ is an input-output example pair, and $\Phi_{\mathcal{N}}$ is a natural language (NL) description of the user intent, e.g., "Find toString and hashCode calls on array objects". The first design principle of EXCALIBUR is *soundness*, i.e., the synthesized programs must satisfy the input-output example in $\Phi_{\mathcal{E}}$. EXCALIBUR only uses the informal $\Phi_{\mathcal{N}}$ to estimate the probabilities of candidate programs instead of generating programs directly from NL.

Example Input. The *example input* is a code snippet I, which could be a Java class, method, or any element that can be parsed as an abstract syntax tree (AST) like Figure 2. Let V denote the set of all AST nodes. An AST t is represented as

- a *leaf* node $v \in V$,
- or $v(t_1, \ldots, t_n)$, where $v \in V$ is an *internal* node, and $t_1, \ldots, t_n$ are subtrees.

Wd also use v to denote the subtree rooted at $v \in V$ for simplicity. Each AST node $v \in V$ is associated with a sort name $v.sort$ denoting a syntactic construct, e.g., `expression` or `typeDeclaration`. Each AST node also has a pointer to its parent $v.parent$ to support the operation of retrieving ancestors. We use r to denote the root of the input code, e.g., `compilationUnit` in Figure 2.

Example Output. The *example output* is a set of user-specified code fragments in the input example, e.g., the highlighted method calls in Figure 1. Each output code fragment o is a subtree in the input AST I, and the example output is denoted as $O = \{o_1, o_2, \ldots\}$.

Hence, both the example input I and the example output O are denoted as sets of ASTs, i.e., $I = \{r\} \subseteq V$ and $O = \{o_1, \ldots, o_n\} \subseteq V$, which correspond to the EXSTATEs we illustrate in Section II-A.

Consistent Program. In the remainder of this paper, we refer to *consistent programs* $\mathbb{P}_c$ as those synthesized programs that output O when running on I, i.e., $\mathbb{P}_c = \{P \mid [\![P]\!](I) = O\}$. The previous synthesizers for code search (e.g., SPORQ [4]) may return an arbitrary consistent program, while EXCALIBUR aims to find *all* consistent programs (ensuring bounded completeness) within a search budget and help users identify the target program.

B. Domain-Specific Language

As illustrated in Section II, programs in EXCALIBUR include XPath-like *path programs* and set operations-based *set programs*. Specifically, they are described in a domain-specific language (DSL) called ExPath in Figure 7.

Like other DSLs for program synthesis, the design principle of ExPath is to be expressive enough to handle a wide range of code search tasks while also restrained enough to make synthesis and disambiguation tractable. We base ExPath on XPath for its expressiveness in describing hierarchical patterns in tree-like structures, e.g., code patterns on ASTs. Such XPath-like languages are widely adopted by source-code analysis libraries (e.g., Spoon [2]) and program synthesizers for refactoring [19], [20].

$$
\begin{aligned}
\langle program \rangle &::= \langle pathProg \rangle \mid \langle setProg \rangle \\
\langle pathProg \rangle &::= \langle pathExpr \rangle \ / \ \langle pathProg \rangle \mid \langle pathExpr \rangle \\
\langle setProg \rangle &::= \text{Conjunct} (\langle program \rangle , \langle program \rangle) \\
&\quad \mid \text{Disjunct} (\langle program \rangle , \langle program \rangle) \\
&\quad \mid \text{Diff} (\langle program \rangle , \langle program \rangle)
\end{aligned}
$$

Fig. 7: Context Free Grammar of ExPath

Path Program	$[\![e \ / \ P]\!](\sigma) = [\![P]\!]([\![e]\!](\sigma))$
Disjunction	$[\![\mathsf{Disjunct}(P_1, P_2)]\!](\sigma) = [\![P_1]\!](\sigma) \cup [\![P_2]\!](\sigma)$
Conjunction	$[\![\mathsf{Conjunct}(P_1, P_2)]\!](\sigma) = [\![P_1]\!](\sigma) \cap [\![P_2]\!](\sigma)$
Difference	$[\![\mathsf{Diff}(P_1, P_2)]\!](\sigma) = [\![P_1]\!](\sigma) - [\![P_2]\!](\sigma)$

Fig. 8: Evaluation rules of ExPath. $[\![\cdot]\!]$ denotes evaluating an expression or a program. Each P denotes an ExPath program. Each e denotes a path expression. Each σ denotes an EXSTATE.

ExPath further enhances the expressiveness with set-based operations. Typically, previous works [19], [20] do not support disjunctive patterns (e.g., "pattern A or B"), possibly because it burdens synthesis efficiency and users are assumed to be able to divide such patterns manually. For users' convenience, ExPath supports disjunction by managing the efficiency burden via the version spaces representation and a bounded search budget, e.g., the number of set operators.

EXSTATE. An EXSTATE σ is a set of code fragments ($\sigma \subseteq V$) that match a specific code pattern (specified by a ExPath program) in the input code snippet. All ExPath operators (path expressions and set operators) take EXSTATEs as inputs and output an EXSTATE, so EXSTATEs can also be viewed as running states of ExPath programs.

Path Program. A *path program* is denoted by a sequence of *path expressions* e separated by slashes "/", and these expressions are evaluated successively, as shown in Figure 8. Each *path expression* takes the current EXSTATE as input and transforms it into the next EXSTATE. The initial EXSTATE (by default $\{r\}$) is transformed successively into the final output EXSTATE, which contains code fragments matched by the whole path program.

Path Expression. A *path expression* is an EXSTATE-to-EXSTATE function. Specifically, EXCALIBUR adopts three categories of path expressions: node-based, semantic-based, and pattern-based. They cover simple (node-based) and complex (pattern-based) AST patterns and simple semantic patterns (semantic-based). Table I lists their representations and evaluation rules, which we illustrate as follows.

(Node-based Expression). A node-based expression consists of an indicator of the relative position in the AST and

TABLE I: Path Expressions and their evaluation rules in EXCALIBUR. RP denotes a relative position relation in ASTs, e.g., descendant or ancestor. $sort$ denotes an AST sort name. UR (resp., BR) denotes a unary (resp., binary) relation on AST nodes. pt denotes a wildcard pattern on AST nodes, and $v \models pt$ denotes v matches pattern pt. e denotes a path expression. σ denotes the input EXSTATE. v denotes an AST node.

Category	Representation	Example	Evaluation Rules
Node-based	RP<$sort$>	`descendant<expression>`	$[\![e]\!](\sigma) = \{v' \mid (v', v) \in RP \wedge v'.sort = sort, v \in \sigma\}$

Semantic-based	RP<UR>
	`forward`<BR>
	`backward`<BR>
		`ancestor<stringTypeExpr>`
		`forward<taintFlow>`
		`backward<methodCall>`
			$[\![e]\!](\sigma) = \{v' \mid (v', v) \in RP \wedge v' \in UR, v \in \sigma\}$
			$[\![e]\!](\sigma) = \{v' \mid (v, v') \in BR, v \in \sigma\}$
			$[\![e]\!](\sigma) = \{v' \mid (v', v) \in BR, v \in \sigma\}$

Pattern-based	RP<pt>	`descendant<* + *>`	$[\![e]\!](\sigma) = \{v' \mid (v', v) \in RP \wedge v' \models pt, v \in \sigma\}$

an indicator of the target AST sort. For example, in `descendant<parameter>`, `descandant` indicates the output AST nodes are descendants of the input ASTs, and `parameter` indicates the output AST nodes have the sort name `parameter`. Evaluating a node-based expression returns AST nodes matching these indicators. Given an input example code snippet, the node-based expressions are obtained by enumerating a fixed set of position indicators and the AST sorts presented in the example code.

(Semantic-based Expression). When a user task relies on semantic information (e.g., finding array-type objects in our motivating example), the example input snippet is compiled and analyzed with a set of predefined rules to extract unary or binary [5] *semantic relations*. A unary relation UR is a set of AST nodes ($UR \subset V$) that share the same property, eg, expressions of array type. A binary relation is over $V \times V$, and each pair $\langle v_1, v_2 \rangle$ in it satisfies a specific relation, e.g., v_1 is a method definition and v_2 is its call site. The rules to extract such relations are adopted based on some common and lightweight semantic analyses (e.g., expression types, control flow, data flow) and are extendable as long as the output format is a unary or binary relation on AST nodes.

A semantic-based expression encapsulates an extracted relation and an indicator of the relative position (for unary relation) or an indicator of flow direction (for binary relation). Evaluating an expression based on a unary relation enumerates AST nodes in the specified relative positions and then returns the nodes satisfying the relation. Unary relation-based expressions can express patterns like array-type expressions or methods inherited from parental classes.

Evaluating an expression based on binary relation enumerates the tuples in the relation and returns those nodes pertinent to the input nodes in the forward or backward direction. Binary relation-based expressions can express patterns like call sites of a focal method.

(Pattern-based Expression). A pattern-based expression extends node-based expression by specifying not a single AST node type but a subtree with wildcard matching.

(Wildcard Pattern). A wildcard pattern pt and whether an AST t matches the pattern ($t \models pt$) is defined recursively. A wildcard pattern pt can be

- a concrete AST t', and $t \models t'$ only if $t = t'$.
- a wildcard symbol $*$, and $t \models *$ for any t.
- a tree structure $st(pt_1, \ldots, pt_k)$ where the root is a nonterminal with sort name st and the children are patterns $pt_1, \ldots, pt_k$. An AST $t = v(t_1, \ldots, t_k)$ matches if $v.sort = st \ \wedge \ t_1 \models pt_1 \ \wedge \ \ldots \ \wedge \ t_k \models pt_k$.

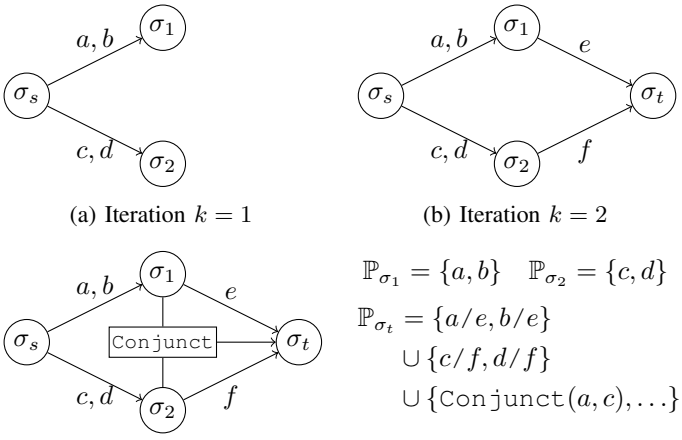
Using pattern-based expressions shortens the required program size of the target program and thus reduces the search space in the synthesis/disambiguation stage. For example, to express "expressions calling the `toString` method", using the wildcard pattern `*.toString()` is more concise than expressing it as a conjunction of several path programs. However, the number of wildcard patterns in a code snippet can be enormous due to the exponential number of combinations. Therefore, EXCALIBUR uses the "most specific" pattern as a representative if two patterns are observationally equivalent, i.e., they match the same AST subtrees in the example code. For example, if both `arr.toString()` and `*.toString()` match the only occurrence `arr.toString()`, the more specific `arr.toString()` in the subsumption lattice [28] is used as the representative. Using the most specific pattern allows users to further edit the pattern (replacing subtrees with $*$) to obtain the intended level of abstraction. EXCALIBUR guarantees the mined wildcard patterns are sound (most specific) and complete, which is detailed in the supplement [23].

Set Program. ExPath adopts set-based operations to express conjunction/disjunction/difference of code patterns described by the subsidiary path programs. Such programs with set-based operators are called *set programs* in ExPath. Specifically, a set program is applying a set-based operator (`Disjunct`/`Conjunct`/`Diff`) on two ExPath programs. The evaluation result is given by applying the set operation on the EXSTATEs returned by the subsidiary ExPath programs, as shown in Figure 8. For example, the pattern "toString and hashCode calls on array objects" can be described as a disjunction of two path programs: (a) matching from array object nodes to their direct ancestor `expressions` calling `toString`, and (b) matching from array object nodes to their direct ancestor `expressions` calling `hashCode`.

IV. PROBABILISTIC VERSION SPACE

This section starts with a brief illustration of the traditional version spaces data structure customized by EXCALIBUR, followed by the introduction of probabilities to this data structure by ProbVS.

[5]This format and our implementation are based on CODEQL.



(a) Iteration $k = 1$ (b) Iteration $k = 2$

(c) Iteration $k = 2, m = 1$ (d) Enumerated Programs

Fig. 9: Example of constructing a version space (a to c) and enumerating the consistent programs (d)

A. Version Spaces

A *version space* [22], [5] is a graph representation of a set of programs. This paper formulates a version space as a graph structure $\langle \Sigma, \sigma_s, \sigma_t, \Pi \rangle$ where

- Σ is a set of ExStates (vertices). Each represents a *running state* of one or more consistent ExPath programs executed on the example input I.
- $\sigma_s, \sigma_t \in \Sigma$ are the source and sink ExStates. The source state $\sigma_s = I = \{r\}$ denotes the input example code, and the target state $\sigma_t = O$ denotes the output examples.
- Π is a set of directed hyper-edges denoting the possible *transitions* between the running states, each of which can be either:
  1) $\sigma_1 \to \sigma_2$ associated with the set of path expressions that can evaluate σ_1 to σ_2;
  2) $\langle \sigma_1, \sigma_2 \rangle \to \sigma_3$ associated with the set of set operators that can evaluate $\langle \sigma_1, \sigma_2 \rangle$ to σ_3.

Constructing Version Spaces. The set of path expressions E is constructed by applying the templates in Table I to the example code. Given E and an example input I, the following procedure constructs the version space of ExPath programs within the *search budget*, i.e., the number of path expressions up to k and the number of set operations up to m:

1) Maintain the set of currently reachable states Σ (initialized as $\{\sigma_s\}$) and the set of edges Π (initially empty) recording how the states are reached.
2) Iterate k times: enumerate all path expressions $e \in E$ and all states $\sigma \in \Sigma$, update Σ with the new state $\sigma' = [\![e]\!](\sigma)$ and add edge $\sigma \xrightarrow{e} \sigma'$ to Π.
3) Iterate m times: enumerate a set operator op and two distinct states $\sigma_1, \sigma_2 \in \Sigma$, update Σ with the new state $\sigma' = op(\sigma_1, \sigma_2)$ and add edge $\langle \sigma_1, \sigma_2 \rangle \xrightarrow{op} \sigma'$ to Π.

Example IV.1. Figure 9 exemplifies the construction of a version space with k up to 2 and m up to 1.

Version Spaces to Programs. Let $\mathbb{P}_\sigma$ denote the set of programs that evaluate σ_s to σ. Given a state $\sigma_o \in \Sigma$ of a constructed version space, $\mathbb{P}_{\sigma_o}$ can be constructed by recursively applying:

- For all edges $\sigma_1 \xrightarrow{e} \sigma_o$, add P/e to $\mathbb{P}_{\sigma_o}$ for all $P \in \mathbb{P}_{\sigma_1}$.
- For all edges $\langle \sigma_1, \sigma_2 \rangle \xrightarrow{op} \sigma_o$, add $op(P_1, P_2)$, to $\mathbb{P}_{\sigma_o}$ for all $P_1 \in \mathbb{P}_{\sigma_1}$ and $P_2 \in \mathbb{P}_{\sigma_2}$.

In particular, example-consistent programs are those that sink at σ_t. Figure 9d examplifies the intermediate $\mathbb{P}_{\sigma_1}$, $\mathbb{P}_{\sigma_2}$, and the final $\mathbb{P}_{\sigma_t}$ in Example IV.1. The set of all consistent programs can be quite large in practice. To avoid the overhead of exhaustive enumeration, Excalibur directly updates the version space (Section V) to be consistent with the user's question answers during the interaction, after which the number of consistent programs is typically reduced to a tractable size, e.g., 1,000. The complete algorithms for constructing version spaces and enumerating programs can be found in the supplementation [23].

Visited Code Fragments. When matching code patterns, programs in the version space "visit" the intermediate code fragments (AST nodes) in ExStates. Take Figures 3 and 4 for example, a user can easily tell `arr.toString()` is one of the code fragments visited by the target program searching for `toString` and `hashCode` calls on arrays, while `()` is not. We formulate the *visited code fragments (VCF)* τ of an ExPath program P on a specific input σ as the set of code fragments occurred during the execution of $P(\sigma)$. The rules to yield τ (denoted as $[\![P]\!](\sigma) \Downarrow \tau$) are given in Figure 10.

$$\frac{\sigma' = [\![e]\!](\sigma)}{[\![e]\!](\sigma) \Downarrow \sigma \cup \sigma'} \qquad \frac{[\![e]\!](\sigma) \Downarrow \tau_1 \qquad [\![P]\!]([\![e]\!](\sigma)) \Downarrow \tau_2}{[\![e/P]\!](\sigma) \Downarrow \tau_1 \cup \tau_2}$$

$$\frac{[\![P_1]\!](\sigma) \Downarrow \tau_1 \qquad [\![P_2]\!](\sigma) \Downarrow \tau_2 \qquad \sigma' = [\![op(P_1, P_2)]\!](\sigma)}{[\![op(P_1, P_2)]\!](\sigma) \Downarrow \tau_1 \cup \tau_2 \cup \sigma'}$$

Fig. 10: Rules for yielding τ. op denotes a set operator.

Example IV.2. Consider the version space in Example IV.1. Let $\sigma_1 = \{v_1, v_2\}$, $\sigma_2 = \{v_2, v_3\}$, and $\sigma_t = \{v_2\}$, the programs a/e and b/e yield $\tau_1 = \{v_1, v_2\}$; c/f and d/f yield $\tau_2 = \{v_2, v_3\}$; and the `Conjunct` programs yield $\tau_3 = \{v_1, v_2, v_3\}$. The AST root in σ_s is omitted for simplicity.

Questions for Disambiguation. As introduced in Section II-C, the questions in Excalibur disambiguate programs based on their visited code fragments. Specifically, a question is asking whether there is a code fragment during execution that is "`exactly` v" or "`nested in` v". Formally, an `exactly` question asks whether the user's target program's visited code fragments τ^* satisfies

$$v \in \tau^*, \tag{1}$$

while a `nested-in` question asks whether

$$\exists v' \in \tau^*.\ v' \in v.\text{descendants}. \tag{2}$$

The set of all possible questions (denoted as the question space $\mathbb{Q}$) is defined by Equations (1) and (2), where v ranges over all code fragments in the example code.

Example IV.3. Consider the code fragments v_1, v_2, v_3 in Example IV.2. Suppose the target program is c/f and $\tau^* = \{v_2, v_3\}$, the question space $\mathbb{Q}$ contains three questions, $v_1 \in \tau^*$, $v_2 \in \tau^*$, and $v_3 \in \tau^*$ according to Equation (1). We omit Equation (2) in examples for simplicity.

B. Probabilistic Model

For efficient disambiguation among consistent programs, our goal is to estimate $\mathbb{P}(P|\Phi)$, the probability of a program P conditioned on the task's specification Φ (both the examples and NL description). ProbVS uses the code fragments visited by P as the *features* of P to help estimate its probability.

A feature ω is a binary event (random variable) indicating whether a program's execution meets a specific condition in Equation (1). Its probability $\mathbb{P}(\omega|\Phi)$ is modeled as how likely (determined by an LLM) the event is true for the target program. A program is assigned a higher (resp., lower) probability if its features are more (resp., less) consistent with the LLM's estimation. The set of code fragments as features can be all the AST subtrees of the user-given example code or its subset.

Probabilities via Features. Suppose a candidate program P is associated with a *feature vector* $\vec{\omega} = \langle \omega'_1, \ldots, \omega'_n \rangle$, following the Bayes' law, the probability of P being the target program can be decomposed as

$$\mathbb{P}(P|\Phi) = \mathbb{P}(\vec{\omega}|\Phi) \cdot \mathbb{P}(P|\vec{\omega}, \Phi). \tag{3}$$

$\mathbb{P}(\vec{\omega}|\Phi)$ is the probability of $\vec{\omega}$ in the joint distribution of features $\{\omega_1, \ldots, \omega_n\}$. To simplify the computation of the joint distribution, ProbVS assumes independence of these random variables [6].

$$\mathbb{P}(\vec{\omega}|\Phi) = \prod_i \mathbb{P}(\omega_i|\Phi). \tag{4}$$

Each ω_i is 0 or 1, and ProbVS estimates $\mathbb{P}(\omega_i|\Phi)$ by encoding Φ in a prompt and asking the LLM to classify the value of ω_i for the target program. $\mathbb{P}(\omega_i = l|\Phi)$ can be computed from the probability of the first token of the classification label l in an LLM's autoregressive generation.

Note that the questions for LLMs ($\vec{\omega}$) is a subset of the questions for users (τ). The user is assumed to know the exact answer because he/she knows the target program's expected behaviors, while an LLM is expected to make user-aligned estimations (i.e., assigning high probabilities when the user answers are positive). Intuitively, a correct estimation "answers" a question in a probabilistic way, thus waving the need for users to answer.

Example IV.4. Suppose the used features are $\langle \omega_1, \omega_2 \rangle$, which correspond to v_1 and v_2 in Example IV.2, respectively. Let $\mathbb{P}(\omega_1|\Phi) = 0.2$ and $\mathbb{P}(\omega_2|\Phi) = 0.9$. The programs with $\tau_1 = \{v_1, v_2\}$ and $\tau_3 = \{v_1, v_2, v_3\}$ have a feature vector $\vec{\omega}_1 = \langle 1, 1 \rangle$, and $\mathbb{P}(\vec{\omega}_1|\Phi) = 0.18$. Programs with $\tau_2 = \{v_2, v_3\}$ corresponds to $\vec{\omega}_2 = \langle 0, 1 \rangle$ and $\mathbb{P}(\vec{\omega}_2|\Phi) = 0.72$. Since $\langle 1, 0 \rangle$ and $\langle 0, 0 \rangle$ are absent from the version space, $\mathbb{P}(\vec{\omega}_1|\Phi)$ and $\mathbb{P}(\vec{\omega}_2|\Phi)$ are normalized to be 0.2 and 0.8, respectively.

[6]Features may be correlated in practice. However, this simplified model still provides a better estimation than not using the execution information, as shown in the evaluation.

Computing Probabilities. Maintaining the probability for each program P for computing the answers' probabilities in question selection is costly. Recall that all questions $q \in \mathbb{Q}$ can be answered by the visited code fragments τ of a program. $\tau \models (q, \alpha)$ if τ answers α. This enables the use of τ to partition the programs in the version space and maintain the cumulative probability per VCF τ.

$$\mathbb{P}(\alpha) = \sum_{P \models (q,\alpha)} \mathbb{P}(P|\Phi) = \sum_{\tau \models (q,\alpha)} \sum_{[\![P]\!](\sigma_s) \Downarrow \tau} \mathbb{P}(P|\Phi) \tag{5}$$

Since each τ determines a unique feature vector $\vec{\omega}_\tau$, the cumulative probability $\mathbb{P}(\tau)$ can be expanded as

$$\mathbb{P}(\tau) = \mathbb{P}(\vec{\omega}_\tau|\Phi) \sum_{[\![P]\!](\sigma_s) \Downarrow \tau} \mathbb{P}(P|\vec{\omega}_\tau, \Phi). \tag{6}$$

EXCALIBUR models $\mathbb{P}(P|\vec{\omega}, \Phi)$ with a heuristic scoring function $s(P)$ over all programs consistent with the example Φ and with the feature vector $\vec{\omega}$:

$$s(P) = \begin{cases} 1/(n+1) & \text{if } P = e_1/\ldots/e_n, \\ s(P_1) \cdot s(P_2) & \text{if } P = op(P_1, P_2) \end{cases},$$

which prioritizes programs with fewer path expressions and set operators. $s(P)$ is then normalized to give the probability distribution of

$$\mathbb{P}(P|\vec{\omega}, \Phi) = \frac{s(P)}{\sum_{P \models \vec{\omega}, \Phi} s(P)}.$$

Note that the definition of $s(P)$ also enables a divide-and-conquer way to sum scores of programs with set operators:

$$\sum_{P_1 \in \mathbb{P}_1, P_2 \in \mathbb{P}_2} s(op(P_1, P_2)) = \sum_{P_1 \in \mathbb{P}_1} s(P_1) \cdot \sum_{P_1 \in \mathbb{P}_1} s(P_2).$$

Example IV.5. Since only τ_2 has the feature vector $\vec{\omega}_2$, we have $\mathbb{P}(\tau_2) = \mathbb{P}(\vec{\omega}_2|\Phi) = 0.8$ from Equation (6). τ_1 and τ_3 share the feature vector $\vec{\omega}_1$, so EXCALIBUR needs to compute program scores. For τ_1, the programs a/e and b/e are both scored $1/3$. For τ_3, the four `Conjunct` programs are all scored $1/4$. After normalization, $\mathbb{P}(\tau_1) = 0.08$, and $\mathbb{P}(\tau_3) = 0.12$. The VCF τ_2 of the target program (c/f) is assigned the highest probability.

V. INTERACTIVE QUESTION-ANSWERING FOR DISAMBIGUATION

This section introduces the operations in a ProbVS to support question-answer interactions, including selecting questions with the minimax-branch strategy and updating the ProbVS according to the user-given answers.

A. Question Selection

During the interaction, EXCALIBUR maintains for each consistent program its visited code fragments (VCFs) τ and the probability of τ, i.e.,

$$\mathrm{T} = \{\tau \mapsto \mathbb{P}(\tau) \mid [\![P]\!](\sigma_s) \models \tau, P \in \mathbb{P}_c\}.$$

Algorithm 1 Beam search

```
1: function BeamSearch(ℚ, b, k)
2:     B ← Map()                          ▷ Initialize the beam set
3:     for all i = 1, ..., k do           ▷ Iterate tuple arity
4:         B′ ← Map()
5:         for all Q ∈ B.Keys(), q ∈ ℚ do
6:             B′.Insert(Q+q, MinimaxBranch(Q+q))
7:         B ← B′.MinByValue(b)           ▷ Keep only top b
8:     return B.MinByValue(1)
```

The initial T can be derived by traversing the version space with the rules in Figure 10, and the probabilities can be computed correspondingly by summing the scores $s(P)$ of programs P that yield τ and then normalizing over $\vec{\omega}_\tau$.

To reduce the number of rounds to finish the question-answer loop, EXCALIBUR leverages the minimax-branch strategy following existing work [10], [11]. In each round, EXCALIBUR presents the user with a *question cluster* Q consisting of exactly k questions, i.e., $Q = \langle q_1, \ldots, q_k \rangle \in \mathbb{Q}^k$. The user gives the answer $\alpha \in \{0, 1\}^k$ for all k questions to eliminate the unintended programs from the version space. The selected question cluster is the one that minimizes the maximum branch probability:

$$Q^* = \operatorname*{argmin}_{Q} \max_{\alpha} \mathbb{P}(\alpha), \tag{7}$$

where each branch's probability is defined in Equation (5).

Instead of calculating Q^* by computing $\max_\alpha \mathbb{P}(\alpha)$ for all $\binom{|\mathbb{Q}|}{k}$ possible question clusters of size k, EXCALIBUR approximates the optimization by the beam-search [29] algorithm in Algorithm 1.

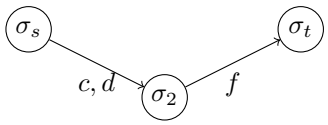
In the algorithm, a beam (window) B of size b is used to maintain the best b intermediate $i-$ary question clusters (keys) and their maximal-branch probabilities (values) after the i-th iteration. When $b = 1$, this algorithm becomes a greedy search, which is the default strategy because the experiment results (Section VII-E) show that increasing the beam size does not significantly improve the selected questions but slows the search efficiency.

For simplicity, the loss function in Equation (7) lists only the maximal branch probability $\max_\alpha \mathbb{P}(\alpha)$. In practice, the MINIMAXBRANCH function compares not only the maximal branch probabilities but also the second largest if the max branches' equal, and the third largest, etc.

B. Updating ProbVS with User Feedback

When a question is answered true or false by the user, EXCALIBUR updates the version space and VCFs T accordingly to maintain only the programs consistent with the user feedback. To ease presentation, we say an EXSTATE σ *witnesses* a question q if σ is an state that makes the existential statement Equation (1) or Equation (2) true.

For question q received a negative answer (0), T is updated by removing all the VCFs whose bit corresponding to q is 1. To update the version space, EXCALIBUR traverses all the states and removes all those who witness q. Because a consistent program must have all its paths in the version space, removing such states (and the pertinent edges) naturally prunes the programs inconsistent with the user feedback $(q, 0)$. For example, answering $v_1 \in \sigma_1$ (Figure 9c) is not visited will prune the version space as follows.



For question q received a positive answer (1), T is updated by removing all the bitvectors whose corresponding bit is 0. Updating the version space, however, cannot be achieved by simply removing states like the negative feedback, because a program can have multiple sub-programs (e.g., Disjunct), and q only needs to be witnessed by one of them to be positive. EXCALIBUR takes a lazy approach in pruning the version space. Specifically, the positive questions are only added to a set $\mathbb{Q}^+$ during the interaction, while the actual pruning process happens when enumerating the consistent programs. When enumerating programs, the associated VCFs are used for checking, and only those that witness all the positive questions $\mathbb{Q}^+$ are returned.

VI. IMPLEMENTATION

Parsing and Semantic Analysis. Parsing the user-given code snippets into ASTs is based on Antlr 4 [30]. Currently, EXCALIBUR targets Java for evaluation but can be extended to other languages. As for the semantic relations as the building blocks of the path expressions, most analysis rules are written in CODEQL queries [1], which return unary or binary AST tuples as relations. The semantic analysis rules employed by EXCALIBUR can be found on its homepage [23]. Note that these rules can be extended to relations represented as 1-ary or 2-ary AST tuples. Although the increase in relations can add overhead to the encoding and disambiguation phases, we observe the overhead is mild because a simple example code does not contain massive relation tuples.

Selecting Code Fragments for Estimation. LLMs cannot handle excessive requests (e.g., all code fragments) within a reasonable time, so our implementation only selects part of the code fragments for estimation. Also, code fragments may "duplicate" and confuse LLMs. For example, a variable `foo` has different semantics when being passed as an argument or being assigned a value. Providing extra context for a code fragment costs extra prompt space and may lead to suboptimal performance. As such, we adopt a workaround: *only code fragments whose normalized texts do not duplicate with others' are selected*. A normalized text is the string form of a code fragment with all the line separators removed and spaces normalized as one space. Consider our motivating example, the normalized text of `arr` in `arr.hashCode()` duplicates with that of `arr` in `arr.toString()`, so both `arr` will not be selected. Otherwise, their contexts need to be specified and consume extra prompt space. On the other hand, this workaround might filter out useful code fragments (features) like the above `arr`. But usually, the remaining features after

filtering are still sufficient for ProbVS to give a probability distribution that is distinguishable.

Computing Probabilities in ProbVS. Computation of the branches' probabilities (Equation (5)) can be accelerated by stacking the feature bitvectors into a matrix and using the matrix acceleration libraries. Our current implementation uses Breeze [31] and multithreading on CPU. Details can be found in the open-source repository [23]. We anticipate the performance could be further enhanced by migrating the computation to GPU with certain engineering efforts.

Prompt Design. To elicit an LLM's capability of estimating whether a code fragment is in the target program's evaluation trace, we refer to the chain-of-thought [27] and few-shot learning [24] tactics to guide the LLM to decompose the task step by step. Specifically, the first step is to identify and print the important constructs (called axes in the prompt) in the NL description. Then, the LLM classifies each given code fragment as an identified axis, or an inner/outer scope of an axis, or irrelevant. Such finer-grained classification aims to reduce hallucination. Only code fragments classified as axes are considered positive. In addition, to reduce the monetary cost and latency of estimating n code fragments in the same task, EXCALIBUR concatenate n retrievals in a single prompt. The final prompt template is as follows, and more details (e.g., the one-shot example) are in the supplementation [23].

> You are a Java expert and user assistant. The user would describe what kind of codes they want to extract. Your task is to analyze the user task and classify some code snippets according to their relations with the user task. Below are some examples.
>
> `<one-shot example>`
>
> ### User Task
> `<user-given description>`
>
> ### Example Code Snippet
> ``` `<example input>` ```
>
> The matched codes are `<example outputs>`.
>
> ### Assistant's Task
> The user-desired codes would be extracted by an XPath-like program, which navigates through AXIS nodes to reach the desired nodes. An AXIS node usually matches some part of the user description but NEEDS NOT match ALL. It can sometimes be the opposite. Your task is to first identify the AXIS constructs from the user description and list them one by one. Then, you need to read each given code, identify its syntactic type (e.g., expression, statement), and compare the code with the AXIS structs you identified above. Finally, classify the nodes as one of the following:
>
> `Axis (x)` denotes the code is exactly the (x) Axis construct you identify.
>
> `Outer` denotes the code is an outer scope of target codes or axes.
>
> `Inner` denotes the code is an inner part of target codes or axes.
>
> `Irrelevant` denotes the code has no relevance to the target codes or axes.
>
> Nodes to classify:
> `<code fragment 1>`
> `<code fragment 2>`
> ...
> ### Answer

VII. EVALUATION

In this section, we evaluate the overall performance of EXCALIBUR and the effectiveness of ProbVS with the following research questions:

- **RQ1.** How effective is EXCALIBUR in synthesizing code search programs?
- **RQ2.** How effective is ProbVS in improving question selection performance?
- **RQ3.** How does ProbVS affect the probability distribution of consistent programs?
- **RQ4.** How do configurable parameters affect the performance of ProbVS?

To answer these questions, we prepared a benchmark comprising 44 code search tasks (Section VII-A). RQ1 investigates the performance of EXCALIBUR and compares it with a replication of a recent code search PBE technique SPORQ [4]. RQ2 investigates the effectiveness of ProbVS in improving question selection, i.e., reducing the number of interactive rounds. We compare the results obtained using the baseline probability distributions (uniform and a priori) with those using ProbVS. RQ3 measures how ProbVS affects candidate programs' probabilities using metrics like entropy (Section VII-A). RQ4 investigates the impacts of configurable parameters (e.g., approximate search strategy) on the performance of EXCALIBUR by experimenting with different parameter settings.

A. Experimental Setting

Benchmark. We collected code search tasks from two sources: (a) Spoon codes from GitHub and (b) the evaluated tasks' descriptions in SPORQ. We consider Spoon because it is a popular code analysis library based on Java ASTs. We identified tasks in the scope of EXCALIBUR from code snippets using Spoon to perform code pattern matching. To find such code snippets, we searched GitHub with the keyword `import spoon.processor` (the package in Spoon for matching code patterns) and selected the top 10 projects ordered by their update time (ordered by stars is not available). We analyzed the documents/codes of these projects and selected those tasks with the complexity levels targeted by existing code search tools [4], [3], [32]. Specifically, the focal tasks could be described in one sentence involving several syntactic/semantic properties, e.g., expressions adding primitive values and strings.

To collect tasks from the SPORQ paper [4], we checked the reported tasks' descriptions, which focus on C/C++ patterns. Since EXCALIBUR is implemented for Java code search tasks, we excluded those tasks that involve features not supported by Java, e.g., casting unsigned to signed and doing pointer arithmetic. For the remaining tasks, we manually adapted them to Java based on their descriptions and the involved features, e.g., implicit type conversion follows similar but different rules in C++ and Java. We excluded 4 tasks that cannot be

expressed or accomplished by all the synthesizers evaluated in this evaluation, including EXCALIBUR and the baseline synthesizers. Finally, the benchmark consists of 36 tasks from GitHub and 8 tasks from the SPORQ paper.

For each task's target code pattern, we manually crafted a natural language description, a target program in ExPath, an input example code, and the associated output code fragments. These materials were prepared by a student helper and verified by an author of this paper to ensure the consistency between the NL description, example code, and target code pattern. We also made similar preparations for the baseline synthesizer.

Table II shows the statistics of the 44 benchmark tasks. Most tasks have **PathLen** ≤ 2 and **#SetOp** ≤ 1, while others are more complex, having **PathLen** $= 3$ or **#SetOp** ≥ 2. Both tasks with or without semantic operations (**#Semantics**) or wildcard patterns (**#Wildcard**) account for fair portions. Hence, this benchmark has a reasonable distribution in different settings and is suitable for evaluating the performance of EXCALIBUR. The full benchmark set is available on our project repository [23].

TABLE II: Benchmark tasks partitioned into groups (table rows) by **PathLen** (the maximum path length in the target program) and **#SetOp** (the number of set operators in the target program). **#Semantics** and **#Wildcard** denote the number of tasks relying on semantic relations and wildcard patterns in each group, respectively.

PathLen	#SetOp	#Tasks	#Semantics	#Wildcard
1	0	10	0	9
2	0	12	7	10
2	1	14	6	11
2	2	2	1	1
3	1	6	1	3
Total		44	15	34

Large Language Models. The LLMs used in this evaluation include GPT4o from OpenAI and two open models for coding, i.e., Qwen2.5-Coder-32B [33] and OpenCoder-8B-Instruct [34]. They are state-of-the-art open models, as indicated by the EvalPlus leaderboard [35], belong to distinct families, and are sufficiently small to be deployed on our machine. To mitigate the randomness of the LLMs, we repeated our experiments three times for each model and reported all the results.

Probability Distributions. As explained in Section IV, the probability distribution of programs is critical to the performance in the question selection problem. To evaluate whether ProbVS can provide a distribution that leads to a better performance, we experiment with the following different distributions and compare their effects on the question selection performance in RQ2 and RQ3.

1) The distributions given by ProbVS with different LLMs, which represent the practical cases, including GPT4o, QwenCoder, and OpenCoder.
2) APRIORI, the distribution computed with only the a priori score function (Section IV), which serves as the baseline with simple heuristic adjustment. It can also be regarded as a baseline of the traditional version spaces with simple ranking/scoring functions [7] normalized as probabilities.
3) RANDOM, the probability distribution assigning uniform probabilities to all programs, which serves as the baseline without any adjustment.
4) OPTIMAL, the distribution computed using ProbVS with perfect estimations extracted from the ground truth, which serves as an upper bound of the ProbVS approach.

Experimental Environment. All the experiments were conducted on a server with a 128-core processor, 1TB of memory, and two RTX 6000 Ada GPUs. It took around 80GB and 20GB of GPU memory to run QwenCoder and OpenCoder, respectively. When running a synthesizer, at most 8 cores and 10 GBs of memory were allowed to simulate the computing power of a desktop machine. A 10-minute time limit was allowed for the evaluating synthesizers to respond [8].

B. RQ1: Effectiveness of EXCALIBUR

The program synthesizers most relevant to EXCALIBUR are ALICE [3], SPORQ [4], and SQUID [32].

While SPORQ and SQUID are not publicly available, ALICE is available as an Eclipse plugin, and we run it on our benchmark tasks. Due to the mismatch of scope (i.e., ALICE only supports limited syntax), ALICE only succeeds in 2 out of 44 tasks. It also misses instances in five tasks, which is caused by the imperfect recall problem discussed in Section I. More details of the results can be found in the supplement [23].

Baseline. For a more in-depth comparison, we construct a baseline based on SPORQ's predecessor GENSYNTH [36], the underlying Datalog [37] synthesis engine based on a genetic algorithm. SPORQ is an application of GENSYNTH in the code search domain with several engineering optimizations. To simulate SPORQ, we implement all the optimizations introduced in its paper [4], including batching Souffle [38] executions, sequentialization of threads for machines with fewer physical cores, and pruning candidate Datalog relations (i.e., the search space). SPORQ is designed for C/C++ with the input Datalog relations based on the CODEQL schemes for C/C++. Therefore, to conduct the concerned experiments in Java, we construct the baseline with the input Datalog relations based on the CODEQL schemes for Java [39]. We use this replication of SPORQ as RQ1's baseline and still refer to it as SPORQ for simplicity.

SPORQ is based on Datalog. A SPORQ program inputs a set of relational tables, performs a series of "join" and "select" operations, and outputs a table with the desired relational tuples. Specifically, the input tables are extracted from a code repository according to a table scheme (e.g., CODEQL Java [39]), and the relational tuples in the tables encode various syntactic/semantic information. For example, a $\langle caller, callee \rangle$ tuple describes a method call in the code repository.

[7]The ranking functions in previous works are not feasible for direct comparison because they are typically domain-specific and heuristics-based. Ours is based on syntactic complexity, like FlashFill [5].

[8]The response time is the total time a synthesizer takes to process the initial specification or user feedback and generate the next question (EXCALIBUR) or example (SPORQ).

The inputs to SPORQ include (a) a code repository to be parsed as relational tables and (b) some of the relational tuples labeled as positive/negative examples. To waive the need to prepare a different repository for each task and make the example specification similar to that of EXCALIBUR, we put all the tasks' example snippets together as a repository and use it for all the tasks. For each task, we chose the example snippet prepared for EXCALIBUR and labeled the example relational tuples for SPORQ to simulate a similar example specification setting. SPORQ should be able to handle this repository because it contains only simple codes (i.e., example snippets for EXCALIBUR) and is thus easier to handle than real-world repositories (e.g., from Github).

Datalog PBE synthesizers usually consider a small number of relational tables as input due to the challenge of synthesis efficiency. However, the tables extracted from a code repository are usually enormous, and SPORQ [4] does not explain whether/how it selects the relational tables. Therefore, we include an additional experiment setting for SPORQ where only the relational tables used by the ground truth program are included. This setting is artificial since the relational tables used in the ground truth program cannot be known in advance in real-world scenarios. We use this simplified setting to upper-bound the performance of SPORQ in our benchmark. We refer to this setting as SPORQ (min).

Metrics. We ran the synthesizers on the benchmark tasks automatically using the ground truth programs to provide "user" feedback. We use the following metrics to evaluate the performances of EXCALIBUR and the SPORQ baseline.

- Success rate, i.e., the number of tasks such that the synthesizers could synthesize a program matching the initial input-output examples and the feedback in the follow-up interactions.
- Number of interaction rounds to accomplish the tasks.
- Response time for the synthesizers to process the initial input or user feedback and generate the next question.
- Precision. We reported how many synthesized programs had the same execution results (on all the example code snippets) as those of the corresponding ground truth programs. Note that this is a necessary but not sufficient condition for being the target program. We used this mechanical metric to compare EXCALIBUR and SPORQ. Given that SPORQ only returns one synthesized program, to have a fair comparison, we took the top-1 program ranked by EXCALIBUR for comparison.

Because SPORQ does not support natural language input, for a fair comparison, EXCALIBUR is based on the APRIORI probability distribution without using ProbVS. We refer to this setting as EXCALIBUR (APRIORI).

Results. The results are tabulated in Table III. In terms of success rate, EXCALIBUR solves all tasks within the time limit. The response time in most tasks and interaction rounds is within 10 seconds, and the majority is within 1 second. Only three tasks with more candidate programs took more than 10 seconds in the first two rounds. In contrast, SPORQ solved only a few (8 – 10) tasks within the time budget, possibly because a large number of input relation tuples overloaded the synthesis algorithm. Even with SPORQ (min), the simplified setting, nearly half of the tasks still timed out. This result matched our expectation because EXCALIBUR synthesizes loop-free DSL programs and can leverage domain-specific optimizations, while SPORQ synthesizes Datalog programs using a genetic algorithm and relies on an external Souffle process to compile, run, and verify the results, which is computationally costly.

In terms of precision, EXCALIBUR has 30 top-1 programs with the same execution results on these 44 code snippets. The above result is fine but also indicates that the simple ranking strategy of EXCALIBUR still has room for improvement, which we leave as future work. In comparison, SPORQ and SPORQ (min) have a higher precision ratio than EXCALIBUR, but the solved tasks are fewer than that of EXCALIBUR. The table also shows that SPORQ might synthesize overfitting programs, as discussed in Section I. We also note that when overfitting programs are presented, previous code search PBE tools need to restart the synthesis and cannot guarantee the next success while EXCALIBUR provides more programs in the ranked list of candidate programs for users to examine.

Summary. EXCALIBUR has faster execution speed than SPORQ and solves more tasks within the time limit. When comparing precision using the top-1 programs, EXCALIBUR's precision ratio is lower than SPORQ's, but EXCALIBUR can provide more programs in a ranked list for more fault tolerance. We conclude that EXCALIBUR is effective in synthesizing code search programs.

C. RQ2: Effectiveness of ProbVS

For RQ2, we investigate the effectiveness of ProbVS in reducing user interactions by comparing the required number of rounds across different probability distributions, including the baselines (APRIORI and RANDOM), the upper bound (OPTIMAL), and ProbVS with different LLMs (GPT4o, QwenCoder, and OpenCoder). We also evaluate the performance of LLMs in estimating the visited code fragments in a task (a binary classification problem) in terms of accuracy, precision, and recall.

Results. Table IV tabulates the main results, and Figure 11 visualizes how the number of solved tasks climbs as the number of rounds increases. In terms of the interactive rounds, we first observe that the two baselines, APRIORI and RANDOM, have nearly identical performances, taking about 4.5 average rounds to solve the tasks in the benchmark. This also shows that an APRIORI probability without using ProbVS has a marginal impact on the question selection performance. In contrast, ProbVS with GPT4o brings notable improvements, reducing $0.64 - 0.8$ average rounds ($14.26\% - 17.86\%$), compared with the baseline (4.48 average rounds). QwenCoder performs slightly better than the baseline. By examining the ranges #T($\cdot$), i.e., how many tasks are solved in a range (e.g., 0-3), we also found GPT4o solves more tasks within the first five rounds, and fewer tasks take more than seven rounds. However, OpenCoder performs below the baseline, taking $0.16 - 0.54$ more average rounds ($3.57\% - 12.05\%$). From the table, we observe that the performance results in interaction rounds are positively correlated with the performance results in

TABLE III: Effectiveness of EXCALIBUR versus SPORQ. The **#Success** denotes the number of tasks solved by synthesizers within the time limit. **Avg #Rounds**, **Avg Response Time**, and **Max Response Time** only consider the successful cases. Each superscript i denotes the i-th trial of running the synthesizer.

Synthesizer	#Success	Avg #Rounds	Avg Response Time (s)	Max Response Time (s)	Precision (Top 1)
EXCALIBUR (APRIORI)	44 (100%)	4.50	1.09	68	30 (68.18%)
SPORQ [1]	8 (18.18%)	2.25	38.50	402	7 (87.50%)
SPORQ [2]	9 (20.45%)	3.00	115.56	562	8 (88.89%)
SPORQ [3]	10 (22.73%)	3.10	134.52	554	9 (90.00%)
SPORQ (min)[1]	25 (56.82%)	3.24	12.20	128	21 (84.00%)
SPORQ (min)[2]	25 (56.82%)	3.04	11.95	159	22 (88.00%)
SPORQ (min)[3]	26 (59.09%)	3.77	29.68	395	22 (84.62%)

TABLE IV: Overall RQ2 results. Accuracy, Precision, Recall, and Rounds are averaged on tasks. #Unparsed denotes the number of tasks whose responses cannot be correctly parsed. #T(·) denotes the number of tasks whose number of rounds fall in the range. Model^i denotes the i-th run with the model. Shaded rows denote the best performance for each model.

Setting	Avg Accuracy	Avg Precision	Avg Recall	#Unparsed	Avg Rounds	#T(0–3)	#T(4–5)	#T(6–7)	#T(> 7)
APRIORI	N/A	N/A	N/A	N/A	4.5	19	11	7	7
RANDOM	N/A	N/A	N/A	N/A	4.48	19	10	9	6
OPTIMAL	100%	100%	100%	N/A	2.64	36	6	0	2
OpenCoder [1]	62.18%	32.86%	51.15%	4	4.68	18	13	4	9
OpenCoder [2]	62.75%	23.31%	55.76%	1	5.02	19	9	3	13
OpenCoder [3]	**66.21%**	**25.34%**	**46.13%**	**4**	**4.64**	**18**	**14**	**4**	**8**
QwenCoder [1]	**79.33%**	**46.52%**	**80.78%**	**2**	**4.25**	**20**	**9**	**8**	**7**
QwenCoder [2]	78.50%	47.22%	79.32%	2	4.39	20	9	7	8
QwenCoder [3]	81.79%	46.41%	83.01%	1	4.25	21	8	10	5
GPT4o [1]	89.76%	52.70%	76.38%	0	3.84	24	10	6	4
GPT4o [2]	89.56%	51.98%	76.14%	0	3.8	22	11	8	3
GPT4o [3]	**90.19%**	**54.63%**	**77.20%**	**0**	**3.68**	**21**	**14**	**7**	**2**

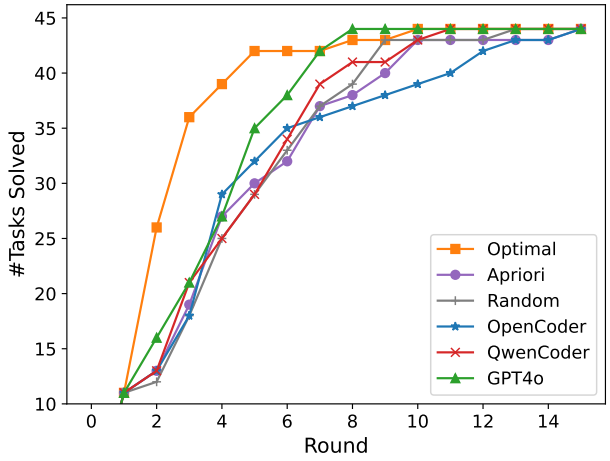


Fig. 11: The accumulated number of solved tasks per round. OpenCoder, QwenCoder, and GPT4o are plotted using the best-performing trial (shaded rows in Table IV).

estimating/predicting code fragments. For example, regarding accuracy, precision, recall, and the number of unparsed tasks, GPT4o generally outperforms other models. This order also aligns with the order of these models' performance in the EvalPlus leaderboard [35].

The OPTIMAL setting, which stands for ProbVS making perfect estimations and assigning probabilities 0.8 to positive features and 0.2 to negative features, significantly outperforms other settings. Nearly all tasks are solved within 5 rounds of interactions. This indicates that ProbVS has considerable potential for future improvement to be delivered by more advanced LLMs with better estimations. Although LLMs nowadays could not reach OPTIMAL's performance, GPT4o has been able to bring a substantial enhancement. Our experimental results also suggest that as an LLM gets more powerful, its estimation is more accurate, and the final performance in reducing interaction rounds is better.

Figure 12 plots a detailed task-wise comparison with the APRIORI baseline. Specifically, the comparison is based on each model's best result, i.e., the shaded rows in Table IV. By comparing with the APRIORI baseline without using ProbVS, we can examine the effectiveness of ProbVS. The RANDOM setting only has a subtle difference from the APRIORI setting, which matches the nearly identical results in Table IV. ProbVS with OpenCoder significantly worsens the results with 15 tasks having up to 4 more rounds. As for QwenCoder, 8 tasks take $1-4$ more rounds, and 11 tasks take $1-5$ fewer rounds, so the overall performance is slightly better. As for GPT4o, 20 tasks have improvement to different extant, and only 7 have minor (≤ 2) degradation. In particular, up to 9 rounds are reduced in a task that needs to take 15 rounds without ProbVS.

Summary. ProbVS can significantly impact the efficiency of question-answer interactions. While a less powerful model like OpenCoder might deteriorate the performance, both the SOTA open-sourced model (QwenCoder) and proprietary model (GPT4o) can bring notable improvement. Encouragingly, empirical results demonstrate the degree of improvement is positively correlated with the capability of the underlying LLM

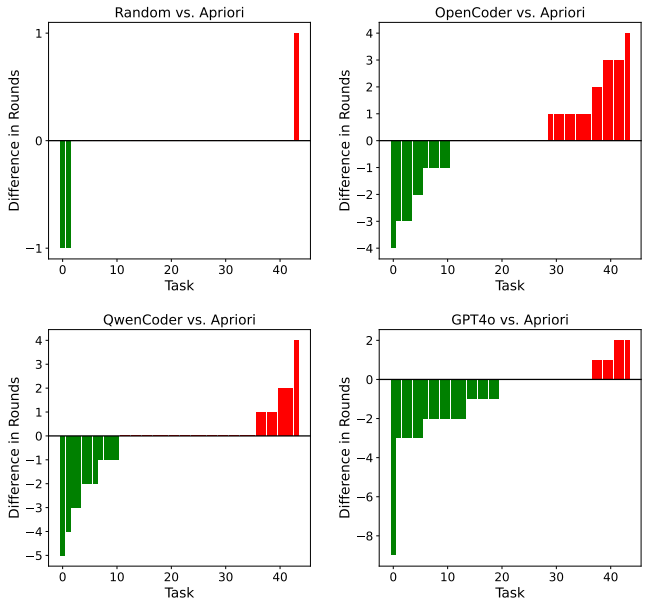


Fig. 12: Task-wise comparison of interactive rounds. Negative bars denote fewer rounds required, and positive bars denote more rounds required, compared with APRIORI.

model. In the future, ProbVS can potentially leverage more advanced LLMs or prompt engineering (Section VII-E) to better approximate the OPTIMAL performance.

D. RQ3: How ProbVS Affects Probability Distribution

To understand ProbVS's effectiveness, this RQ inspects how ProbVS affects a probability distribution from two perspectives, entropy and target probability. Specifically, we inspect the cumulative probability per VCF $\mathbb{P}(\tau)$ (Equation (6)) instead of the probabilities over programs to reduce the runtime overheads.

Entropy. Entropy is a common metric in information theory to describe the degree of uncertainty/randomness of a probability distribution, which is defined as

$$H(D) = -\sum_{\tau \in D} \mathbb{P}(\tau) \log \mathbb{P}(\tau)$$

where D denotes a probability distribution of VCFs (τ). The larger the entropy is, the more "uncertain" the distribution is about the VCF of the target program. The entropy changes as the interaction proceeds and becomes 0 if only one VCF having a probability of 1 remains in the distribution.

Figure 13 plots the entropies of different distributions in the first four rounds of interaction. The entropies of all LLMs' distributions are generally smaller than those of RANDOM or APRIORI distributions, including OpenCoder in the first three rounds, even though it makes wrong predictions and degrades the performance (Table IV). This indicates that ProbVS can change the probability distributions significantly, although it is not necessarily aligned with the user intent. If the change is aligned (e.g., GPT4o and QwenCoder), the performance can be improved, or degraded otherwise (OpenCoder).

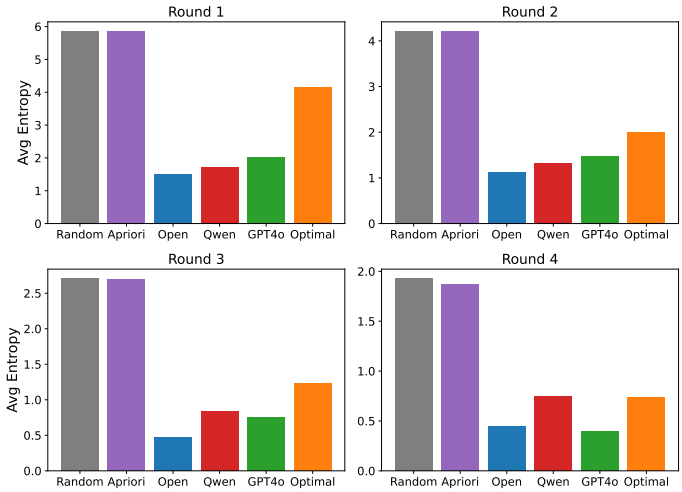


Fig. 13: Average entropies in the first four rounds

Target Probability. Given a probability distribution of VCFs of the consistent programs, the *target probability* we examine is the probability of the target program's VCF. Generally, with a higher target probability, the minimax-branch strategy is more likely to place the target program in a branch with fewer programs, thus simplifying the subsequent disambiguation problems and reducing the required interactive rounds, as illustrated in Section II-C.

To examine the impact of target probabilities on the number of interaction rounds, we perform a chi-square test of independence. Specifically, we test *prop* and *diff*, where *prop* is the proportion of a model's target probability to the baseline APRIORI's target probability in a task, and *diff* is the model's rounds minus the baseline's rounds and can be negative. We exclude the 11 simple tasks that can be finished in one round, and the remaining 33 tasks were run on 9 models (Table IV) to yield 297 samples.

We categorize the contingency table as follows. The chi-square test result reveals a significant association between *prop* and *diff* with a p value 0.006. From the table, we found that if the model has a better estimation ($prop > 1$), the number of rounds is likely to be reduced. Interestingly, the number of rounds may also decrease even if the estimation is not good ($prop < 1$). The analysis and theory of this phenomenon could be interesting future work.

	$diff > 0$	$diff = 0$	$diff < 0$
$prop < 1$	48	45	41
$prop > 1$	47	37	79

Summary. ProbVS can change the probability distributions and reduce entropies. The change has generally positive impacts and increases the target probabilities, which is positively correlated to reducing the interaction rounds.

E. RQ4: Impact of Configurable Parameters

This RQ investigates the impacts of two configurable parameters, namely, the approximate search strategy and the strategy to assign probabilities to features.

Approximate Search Strategy. As discussed in Section VI, searching the optimal question by enumeration is unrealistic, so EXCALIBUR uses an approximate algorithm, and the default strategy is greedy search. This RQ compares greedy search with beam search of different beam sizes (greedy search can also be viewed as beam search of beam size 1) in terms of effectiveness (# of interaction rounds) and efficiency (response time in seconds). This experiment is based on the GPT4o estimations, and the results are tabulated below.

Increasing the beam size significantly increases the response time but does not provide significant reductions in the number of interaction rounds. Whether a larger beam size would reduce the interaction rounds is unknown, but the response time must exceed the acceptable latency in user interaction. Therefore, among the smaller beam sizes, greedy search (size 1) is cost-effective and is set as EXCALIBUR's default strategy.

Strategy	Avg Rounds			Avg Response Time (s)		
	$4o^1$	$4o^2$	$4o^3$	$4o^1$	$4o^2$	$4o^3$
Greedy	3.84	3.80	3.68	2.93	2.36	2.95
Beam-2	3.77	3.73	3.68	3.00	2.38	3.13
Beam-5	3.80	3.70	3.65	5.13	4.10	4.55
Beam-10	3.84	3.73	3.68	7.56	5.38	6.45

Probability Assignment and Prompt. Recall that ProbVS depends on the probabilities $\mathbb{P}(\omega = l|\Phi)$ assigned to each feature ω, where l is the binary label classified by LLMs. Regarding how to assign probabilities, the default strategy of EXCALIBUR uses the log probabilities of the first label token. An alternative strategy is to assign a fixed value like 0.8.

This subsection compares the two assignment strategies. Since the fixed-value strategy depends on the classification accuracy, we also try another prompt that instructs LLMs to give short reasoning before classifying each label. We denote this prompt as COT and the default prompt CLASS. Note that COT does not fit the log-probability assignment because the reasoning process makes the log probabilities of labels deterministic (very close to 1 or 0).

Table V tabulates the average number of rounds under different settings and LLMs. Each LLM is run three times. GPT4o performs better using the CLASS prompt and the log-probability strategy, while QwenCoder performs better when using the COT prompt and the fixed 0.8 strategy. By inspecting the binary classification results (in the supplement [23]) of the two prompts, QwenCoder has higher accuracy under the COT prompt while the precision/recall remains roughly the same, which may explain why it performs better in question selection. For GPT4o, the accuracy and precision are higher in the CLASS prompt, while the recall is lower.

The overall difference in the final performance caused by probability assignment and prompts is notable. While this paper shows the effectiveness of ProbVS, how to optimize the prompt and probability assignment to approximate the OPTIMAL performance is left as future work.

F. Summary of Evaluation

EXCALIBUR is effective and responsive in synthesizing code search programs and addressing the overfitting limitation in existing code search synthesizers. ProbVS can significantly reduce the rounds of interactions by changing the candidate programs' probability distribution (reducing the entropy and increasing the probability of the target program). To make ProbVS more effective in practice, more efforts can be put into acquiring more accurate estimations, e.g., using more powerful LLMs or improving the prompt.

TABLE V: Average rounds under different prompts (CLASS or COT) and probability assignments (LogProb or 0.8/0.2)

Model	CLASS + LogProb (0.8/0.2)			COT + 0.8/0.2		
GPT4o	3.84 (3.98)	3.79 (4.00)	3.68 (3.86)	4.11	4.06	4.00
Qwen	4.25 (4.27)	4.38 (4.32)	4.25 (4.30)	4.11	4.18	3.90
Open	4.68 (4.61)	5.02 (4.93)	4.63 (4.57)	4.84	5.06	5.00

VIII. DISCUSSION

Threats to Validity. Our baseline is a replication of SPORQ and may not fully reflect the performance of the state-of-the-art code search synthesizer. Nonetheless, we have made our best efforts to implement the techniques in the paper [4] faithfully. For the most uncertain part of how the input relations are selected, we have tried both the minimal setting and the origin setting to give a bound of the baseline performance. The replication will be open sourced.

Also, the comparison between EXCALIBUR and SPORQ is not strictly fair, because SPORQ takes a code project as input but does not target completeness, while EXCALIBUR targets completeness but takes a user-provided simple code snippet as input. To date, formal synthesizers for code search such as SPORQ [4], SQUID [32], and EXCALIBUR commonly face the challenge of scalability due to the flexibility of code and the expressiveness required to describe the pattern. Although these synthesizers take different specifications as input, the difficulties of tasks (complexity of code patterns) that can be handled are comparable, and so are the tasks used for evaluation. Therefore, our experiments in RQ1 can still qualitatively demonstrate the effectiveness of EXCALIBUR.

For randomness (SPORQ-based baselines and LLMs) in the experiment, we have repeated the related experiments several times and reported the results. We found the results are basically consistent and replicable.

LLMs are usually concerned with data leakage [40], [41], i.e., overrating LLMs' performance if the evaluating subjects are leaked to the training data. This is a critical concern in a creative scenario like code generation. Conversely, our use of LLMs is classification based on common knowledge. Moreover, we manually prepared the example codes and user descriptions, which should not be seen during training.

Limitations and Future Work. The ranking performance of EXCALIBUR still has room for improvement. On one hand, the ranking algorithm can be improved with a better use of users' NL descriptions. On the other hand, the current two question templates (Equations (1) and (2)) cannot distinguish programs matching the same set of code fragments. Using one or more fine-grained question templates (e.g., regarding

ExStates) could distinguish and prune more unintended programs, making the target programs easier to spot.

As for ProbVS, to further improve the question selection performance, a cost-effective way is to improve the prompt to allow LLMs predict more accurately. Because code search tasks share similar logic but differ in the specific constructs/semantics, preparing a pool of examples and using retrieval-augmented generation (RAG) [42] seems a promising direction. Moreover, ProbVS is not necessarily coupled with Excalibur. The prerequisites of ProbVS include version spaces and that LLMs can understand the intermediate values in version spaces (e.g., the visited code fragments in Excalibur). They should also be available in other PBE domains, such as string manipulation [5] and table transformations [43], [44]. In the future, we will explore the possibility of extending ProbVS to such application domains.

IX. Related Work

Program Synthesis for Tasks on Code. Data-driven methods have been leveraged to synthesize sophisticated and specific analyses such as Javascript points-to analysis [45] and side channel analysis [46]. These works require specific oracles (e.g., whether the synthesized analysis is sound) and extensive data, which differs from the setting of code search synthesis. Another line of works learns patterns from program transformations (e.g., code refactoring) [19], [20]. Their algorithms are specialized for changed pairs (codes before and after) and may not work well in code search tasks without changed pairs. Alice [3], Sporq [4], and Squid [32] are all for discovering code pattern instances. Alice considers a limited set of syntax, while Sporq, Squid, and Excalibur consider all syntax and some semantic information, e.g., type.

Interactive Program Synthesis.In recent years, interactive learning [47] for program synthesis [9] is becoming increasingly popular, possibly due to the intrinsic overfitting problem in program synthesis [48]. Interactive program synthesis enables users to more effectively search the target program with either more examples or testcases [3], [4], [10], [49] or other kinds of constraints, such as program state [50], human demonstration [51], and the questions in Excalibur.

Alice [3] and Sporq [4] are both interactive synthesizers for discovering code pattern instances in a codebase. As discussed in Section VII, they rely on a codebase to refine programs but may still return overfitting programs. Our evaluation on Alice and the replication of Sporq also suggests that the overfitting happens in practice. Compared with Alice and Sporq, Excalibur is bounded-complete and does not suffer from overfitting.

Question selection is an important problem [10], [12], [11] that studies how to select questions to disambiguate programs while minimizing the user's effort. Dillig et al. [52] studied a similar problem that aims to minimize user effort to classify bug reports using abductive inference. Alps [53] adopts an active learning strategy but only considers a binary classification and does not consider the candidate programs' probabilities. Yewen et al. [54] proposes to use pragmatic communication to compute programs' probabilities and a fast implementation based on the version spaces structures. Excalibur proposes ProbVS that uses LLMs to estimate intermediate values in version spaces to obtain more user-aligned distributions for more effective question selection.

Version Spaces.Version spaces (VSs) [55] have been widely used in program synthesis [5], [22], [56], [57], [58], [59] to represent observationally equivalent programs succinctly and process them efficiently. The construction/learning of VSs can be backward [60], [5] or forward [57], [43], dependent on the domain-specific operators. Notably, DACE [57] constructs VSs based on finite tree automata (FTA), and VSs and FTAs are found equivalent later [18]. The use of a VS is usually accompanied by domain-specific operations. For example, Excalibur customizes the pruning of invalid consistent programs given the user answers (Section V-B). Moreover, Excalibur introduces probabilities to VSs and defines the operations to calculate probabilities.

Ranking Functions and Probabilities. A ranking function in traditional version spaces [5], [60] only defines a total order but does not necessarily defines a probability distribution. If a ranking function is based on scoring, the scores can be normalized as probabilities, such as the Apriori baseline in Section Section VII. Compared with probabilistic context-free grammar (PCFG) [26] that assigns a priori and uniform probabilities to grammar rules, the probabilities in ProbVS is conditioned on the concrete tasks and is expected to give more accurate estimations.

Large Language Models.LLMs are widely used for code generation recently [61], [62], [63], [64], but its performance in low-resourced languages (e.g., DSL) is suboptimal due to the lack of training data [14], [15], [16], [65]. Grammar prompting [14] is proposed to enhance code generation in DSL through in-context learning, but it is still not as reliable as well-trained languages like Python. Our pilot study on the GPT-4 series [66] to generate ExPath programs through in-context learning also encountered failures such as reversing the order of path expressions and choosing wrong path expressions due to misunderstanding their semantics, which suggests that a more sophisticated workflow (e.g., with RAG [42] and iterations) is required to generate code search scripts using purely LLMs. Moreover, purely LLMs cannot guarantee soundness or completeness like symbolic synthesizers, but their combination, i.e., neuro-symbolic [67], [68] methodslike ProbVS may preserve the bests of both worlds .

X. Conclusion

This paper presents Excalibur, a multi-modal and interactive program synthesizer for code search. Excalibur synthesizes ExPath (DSL) programs based on the example specification and guarantees the soundness and bounded completeness of synthesis. Moreover, Excalibur supports question-answer interaction to identify the user-intended program and advances the question selection performance with a novel approach dubbed probabilistic version space (ProbVS) based on the user-given NL description and LLMs. The evaluation on a benchmark of code search tasks shows the effectiveness of Excalibur and ProbVS.

REFERENCES

[1] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: object-oriented queries on relational data," in *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, ser. LIPIcs, S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:25. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[2] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of java source code," *Softw. Pract. Exp.*, vol. 46, no. 9, pp. 1155–1179, 2016. [Online]. Available: https://doi.org/10.1002/spe.2346

[3] A. Sivaraman, T. Zhang, G. V. den Broeck, and M. Kim, "Active inductive logic programming for code search," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 292–303. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00044

[4] A. Naik, J. Mendelson, N. Sands, Y. Wang, M. Naik, and M. Raghothaman, "Sporq: An interactive environment for exploring code using query-by-example," in *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, J. Nichols, R. Kumar, and M. Nebeling, Eds. ACM, 2021, pp. 84–99. [Online]. Available: https://doi.org/10.1145/3472749.3474737

[5] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 317–330, Jan. 2011. [Online]. Available: https://dl.acm.org/doi/10.1145/1925844.1926423

[6] D. C. Halbert, *Programming by example*. University of California, Berkeley, 1984.

[7] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu, "Abacus: Precise side-channel analysis," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 797–809. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00078

[8] M. Kellogg, M. Ran, M. Sridharan, M. Schäf, and M. D. Ernst, "Verifying object construction," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 1447–1458. [Online]. Available: https://doi.org/10.1145/3377811.3380341

[9] V. Le, D. Perelman, O. Polozov, M. Raza, A. Udupa, and S. Gulwani, "Interactive program synthesis," *CoRR*, vol. abs/1703.03539, 2017. [Online]. Available: http://arxiv.org/abs/1703.03539

[10] R. Ji, J. Liang, Y. Xiong, L. Zhang, and Z. Hu, "Question selection for interactive program synthesis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 1143–1158. [Online]. Available: https://doi.org/10.1145/3385412.3386025

[11] A. Tiwari, A. Radhakrishna, S. Gulwani, and D. Perelman, "Information-theoretic user interaction: Significant inputs for program synthesis," *CoRR*, vol. abs/2006.12638, 2020. [Online]. Available: https://arxiv.org/abs/2006.12638

[12] R. Ji, C. Kong, Y. Xiong, and Z. Hu, "Improving oracle-guided inductive synthesis by efficient question selection," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 819–847, 2023. [Online]. Available: https://doi.org/10.1145/3586055

[13] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[14] B. Wang, Z. Wang, X. Wang, Y. Cao, R. A. Saurous, and Y. Kim, "Grammar prompting for domain-specific language generation with large language models," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/cd40d0d65bfebb894ccc9ea822b47fa8-Abstract-Conference.html

[15] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[16] Z. Tang, J. Ge, S. Liu, T. Zhu, T. Xu, L. Huang, and B. Luo, "Domain adaptive code completion via language models and decoupled domain databases," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 421–433. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00076

[17] W. Lee, K. Heo, R. Alur, and M. Naik, "Accelerating search-based program synthesis using learned probabilistic models," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 436–449. [Online]. Available: https://doi.org/10.1145/3192366.3192410

[18] J. Koppel, "Version space algebras are acyclic tree automata," *CoRR*, vol. abs/2107.12568, 2021. [Online]. Available: https://arxiv.org/abs/2107.12568

[19] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 404–415.

[20] X. Gao, S. Barke, A. Radhakrishna, G. Soares, S. Gulwani, A. Leung, N. Nagappan, and A. Tiwari, "Feedback-driven semi-supervised synthesis of program transformations," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[21] "W3C XML path language," in *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*, R. Alhajj and J. G. Rokne, Eds. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4939-7131-2_101447

[22] T. A. Lau, P. M. Domingos, and D. S. Weld, "Version space algebra and its application to programming by demonstration," in *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, P. Langley, Ed. Morgan Kaufmann, 2000, pp. 527–534.

[23] Excalibur, "Excalibur," 2023. [Online]. Available: https://github.com/ExcaliburTool/Excalibur

[24] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[25] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[26] F. Jelinek, J. D. Lafferty, and R. L. Mercer, "Basic methods of probabilistic context free grammars," in *Speech Recognition and Understanding*, P. Laface and R. De Mori, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 345–360.

[27] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[28] G. Plokin, *Lattice theoretic properties of subsumption*. Edinburgh University, Department of Machine Intelligence and Perception, 1970.

[29] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and R. Reddy, "The hearsay-ii speech-understanding system: Integrating knowledge to

resolve uncertainty," *ACM Comput. Surv.*, vol. 12, no. 2, pp. 213–253, 1980. [Online]. Available: https://doi.org/10.1145/356810.356816

[30] T. Parr, *The definitive ANTLR 4 reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[31] scalanlp, "Breeze github page," 2024. [Online]. Available: https://github.com/scalanlp/breeze

[32] C. Wang, P. Yao, W. Tang, G. Fan, and C. Zhang, "Synthesizing conjunctive queries for code search," in *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, ser. LIPIcs, K. Ali and G. Salvaneschi, Eds., vol. 263. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 36:1–36:30. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2023.36

[33] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[34] S. Huang, T. Cheng, J. K. Liu, J. Hao, L. Song, Y. Xu, J. Yang, J. H. Liu, C. Zhang, L. Chai, R. Yuan, Z. Zhang, J. Fu, Q. Liu, G. Zhang, Z. Wang, Y. Qi, Y. Xu, and W. Chu, "Opencoder: The open cookbook for top-tier code large language models," 2024. [Online]. Available: https://arxiv.org/pdf/2411.04905

[35] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[36] J. Mendelson, A. Naik, M. Raghothaman, and M. Naik, "GENSYNTH: synthesizing datalog programs without language bias," in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 6444–6453. [Online]. Available: https://doi.org/10.1609/aaai.v35i7.16799

[37] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, 1989. [Online]. Available: https://doi.org/10.1109/69.43410

[38] H. Jordan, B. Scholz, and P. Subotic, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780. Springer, 2016, pp. 422–430. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_23

[39] CodeQL, "Codeql's db scheme for java," https://github.com/github/codeql/blob/main/java/ql/lib/config/semmlecode.dbscheme, accessed: 2024-03.

[40] N. Carlini, F. Tramèr, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. B. Brown, D. Song, Ú. Erlingsson, A. Oprea, and C. Raffel, "Extracting training data from large language models," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. D. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 2633–2650. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting

[41] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "What do code models memorize? an empirical study on large language models of code," *CoRR*, vol. abs/2308.09932, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.09932

[42] P. S. H. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[43] C. Wang, A. Cheung, and R. Bodik, "Synthesizing highly expressive SQL queries from input-output examples," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 452–466. [Online]. Available: https://doi.org/10.1145/3062341.3062365

[44] J. Wu, L. Wei, Y. Jiang, S. Cheung, L. Ren, and C. Xu, "Programming by example made easy," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, pp. 4:1–4:36, 2024. [Online]. Available: https://doi.org/10.1145/3607185

[45] P. Bielik, V. Raychev, and M. T. Vechev, "Learning a static analyzer from data," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10426. Springer, 2017, pp. 233–253. [Online]. Available: https://doi.org/10.1007/978-3-319-63387-9_12

[46] J. Wang, C. Sung, M. Raghothaman, and C. Wang, "Data-driven synthesis of provably sound side channel analyses," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 810–822. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00079

[47] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6

[48] S. Padhi, T. D. Millstein, A. V. Nori, and R. Sharma, "Overfitting in synthesis: Theory and practice," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds., vol. 11561. Springer, 2019, pp. 315–334. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_17

[49] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," *CoRR*, vol. abs/2208.05950, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2208.05950

[50] K. Ferdowsifard, S. Barke, H. Peleg, S. Lerner, and N. Polikarpova, "Loopy: interactive program synthesis with control structures," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–29, 2021. [Online]. Available: https://doi.org/10.1145/3485530

[51] R. Dong, Z. Huang, I. I. Lam, Y. Chen, and X. Wang, "WebRobot: web robotic process automation using interactive programming-by-demonstration," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 152–167. [Online]. Available: http://doi.org/10.1145/3519939.3523711

[52] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 181–192. [Online]. Available: https://doi.org/10.1145/2254064.2254087

[53] X. Si, W. Lee, R. Zhang, A. Albarghouthi, P. Koutris, and M. Naik, "Syntax-guided synthesis of Datalog programs," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 515–527. [Online]. Available: https://doi.org/10.1145/3236024.3236034

[54] Y. Pu, K. Ellis, M. Kryven, J. Tenenbaum, and A. Solar-Lezama, "Program synthesis with pragmatic communication," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/99c83c904d0d64fbef50d919a5c66a80-Abstract.html

[55] T. M. Mitchell, "Version spaces: A candidate elimination approach to rule learning," in *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*, R. Reddy, Ed. William Kaufmann, 1977, pp. 305–310. [Online]. Available: http://ijcai.org/Proceedings/77-1/Papers/048.pdf

[56] W. Lee and H. Cho, "Inductive synthesis of structurally recursive functional programs from non-recursive expressions," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 2048–2078, 2023. [Online]. Available: https://doi.org/10.1145/3571263

[57] X. Wang, I. Dillig, and R. Singh, "Synthesis of data completion scripts using finite tree automata," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 62:1–62:26, Oct. 2017. [Online]. Available: http://doi.org/10.1145/3133886

[58] C. Li, Y. Jiang, and C. Xu, "Push-button synthesis of watch companions for Android apps," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA:

Association for Computing Machinery, Jul. 2022, pp. 1793–1804. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510056

[59] R. Singh and S. Gulwani, "Predicting a correct program in programming by example," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 398–414. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_23

[60] O. Polozov and S. Gulwani, "FlashMeta: a framework for inductive program synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 107–126. [Online]. Available: https://dl.acm.org/doi/10.1145/2814270.2814310

[61] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[62] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. M. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with GPT-4," *CoRR*, vol. abs/2303.12712, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.12712

[63] Phind, "Phind," 2023. [Online]. Available: https://huggingface.co/Phind/Phind-CodeLlama-34B-v2

[64] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *CoRR*, vol. abs/2306.08568, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.08568

[65] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *CoRR*, vol. abs/2308.01861, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.01861

[66] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.08774

[67] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=rJ0JwFcex

[68] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong, "Solving olympiad geometry without human demonstrations," *Nat.*, vol. 625, no. 7995, pp. 476–482, 2024. [Online]. Available: https://doi.org/10.1038/s41586-023-06747-5