Question Selection for Interactive Program Synthesis

Ruyi Ji
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
Beijing, China
jiruyi910387714@pku.edu.cn

Jingjing Liang
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
Beijing, China
jingjingliang@pku.edu.cn

Yingfei Xiong∗
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
Beijing, China
xiongyf@pku.edu.cn

Lu Zhang
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
Beijing, China
zhanglucs@pku.edu.cn

Zhenjiang Hu
Key Lab of High Confidence Software Technologies, Ministry of Education
Department of Computer Science and Technology, EECS, Peking University
Beijing, China
huzj@pku.edu.cn

Abstract
Interactive program synthesis aims to solve the ambiguity in specifications, and selecting the proper question to minimize the rounds of interactions is critical to the performance of interactive program synthesis. In this paper we address this question selection problem and propose two algorithms. SampleSy approximates a state-of-the-art strategy proposed for optimal decision tree and has a short response time to enable interaction. EpsSy further reduces the rounds of interactions by approximating SampleSy with a bounded error rate. To implement the two algorithms, we further propose VSampler, an approach to sampling programs from a probabilistic context-free grammar based on version space algebra. The evaluation shows the effectiveness of both algorithms.

CCS Concepts: • Software and its engineering → Software notations and tools; General programming languages.

Keywords: Interaction, Program Synthesis

ACM Reference Format:

1 Introduction

Program synthesis [3, 20] finds a program meeting a given specification and has many applications [14, 15, 17, 35, 40, 46, 47, 50, 51]. A classic program synthesis problem assumes that the specification is complete, and the synthesizer only needs to find any program that meets the specification. However, in practice the specification is often incomplete, leading to ambiguity among many program candidates [36]. For example, in programming by example [18], it is common that the examples are few and many programs are consistent with the examples [38]. In program repair, it is also known that incorrect patches are often produced because of incomplete specification [34, 42].

A fundamental way to address the ambiguity is interactive program synthesis [30]. In interactive program synthesis, the system asks easy-to-answer questions to the user, and narrows down the program domain based on the answer from the user. The interaction ends when there is no ambiguity remaining. Compared to non-interactive program synthesis, interactive program synthesis adds the extra burden of answering questions to the user. Therefore, it is critical to minimize the number of questions needed to be asked.

As a matter of fact, the selection of questions has a great impact on the number of questions needed. Let us consider a program domain $P_e$ defined by the following grammar:

$$S ::= E \mid \text{if } E \leq E \text{ then } x \text{ else } y \quad E ::= 0 \mid x \mid y$$
There are nine semantically different programs in the domain, as follows.

\[(p_1) 0 \quad (p_2) \text{if } 0 \leq x \text{ then } x \text{ else } y \quad (p_3) \text{if } 0 \leq y \text{ then } x \text{ else } y \quad (p_4) x \quad (p_5) \text{if } x \leq 0 \text{ then } x \text{ else } y \quad (p_6) \text{if } x \leq y \text{ then } x \text{ else } y \quad (p_7) y \quad (p_8) \text{if } y \leq 0 \text{ then } x \text{ else } y \quad (p_9) \text{if } y \leq x \text{ then } x \text{ else } y\]

To find out which program is the desired one, an interactive synthesis system may use input-output questions, i.e., in each round, the system selects an input, and asks the output of the input from the user. The selection of questions can significantly affect the number of questions. For example, when the target program is \(p_6\), it is possible to finish synthesis with two questions: (1, 2) and (2, 1), where \((a, b)\) represents input \(x = a, y = b\). But if the synthesizer always chooses inputs from \([0, i]\) \(|i \geq 0\), the interaction will never finish since these inputs cannot distinguish \(p_6\) from \(p_1\).

As a result, it is important to study the question selection problem. Given a program domain \(P\) with a probability distribution \(\varphi\) on it, a domain \(Q\) of questions, and a sequence \(C\) of previous questions and answers, the question selection problem is to select the next question from \(Q\) such that the expected number of questions needed to remove ambiguity is minimized. As far as we are aware, the question selection problem has not been studied in the program synthesis community, and recent interactive program synthesis approaches all randomly select questions, leading to unnecessarily extra burden to the user.

In this paper we address the question selection problem for interactive program synthesis. Our basic idea is to adapt a state-of-the-art polynomial-time strategy for optimal decision tree problem, \textit{minimax branch} [2, 11, 21], which selects the question where the worst user answer gives the best reduction of the program domain. In this example, \((-1, 1)\) is one best choice for the first question because it can exclude at least 5 programs whatever the answer is.

However, there are two main challenges in adapting \textit{minimax branch}. First, applying this strategy requires to calculate for each program, each question, and each answer, whether the program is excluded or not. As a result, the time complexity is \(O(|P| \times |Q| \times |A|)\), where \(|P|, |Q|, \) and \(|A|\) are the sizes of the program domain, question domain, and answer domain, respectively. Even for this small example, \(|P| \times |Q| \times |A|\) is already \(9 \times 2^{96}\) (assuming \(x\) and \(y\) are 32-bit integers). Such a computation is almost impossible to be finished in a fast interactive session. Second, due to the complexity of program domains, \textit{minimax branch} still results in too many questions for some practical problems. In our evaluation, an algorithm approximating \textit{minimax branch} uses up to 18 questions, which involves too much burden to the user.

We propose two algorithms to address the two challenges. The first algorithm, \textit{SampleSy}, approximates \textit{minimax branch} using a Monte Carlo method: in each turn, \textit{SampleSy} firstly samples a small set \(P\) of programs from the remaining program domain, and then applies \textit{minimax branch} only to \(P\). For example, if the samples are \(p_1, p_3, p_7\), \textit{SampleSy} still chooses \((-1, 1)\), which is the best input on distinguishing these three samples. In this way \textit{SampleSy} removes the enumeration on the program domain. To further remove the enumerations on the question and answer domains, \textit{SampleSy} generates constraints whose sizes only depend on \(|P|\) and utilizes an SMT solver to solve them: whether there exists question \(q\), such that at most \(t\) programs whose answers are the same. In this way, \(t\) is an upper bound of the worst-case program domain reduction for question \(q\). By searching on \(t\), \textit{SampleSy} finds the best question \(q\). Finally, since sampling \(P\) also requires a significant amount of time, \textit{SampleSy} utilizes the intervals when the user answers the question to perform the sampling, and thus ensures a short response time. We show that \textit{SampleSy} approximates \textit{minimax branch} with a bounded probability.

The second algorithm, \textit{EpsSy}, reduces the rounds of interactions by allowing bounded errors, i.e., potentially returning an incorrect program with a bounded probability. \textit{EpsSy} reduces the rounds of interactions from two aspects. First, when a random sample from the program domain already ensures the bounded error rate, we stop the interaction process. For example, when the bounded error rate is 5%, if in the remaining program domain there is a program whose probability is larger than 95%, the interaction process will be stopped. Second, \textit{EpsSy} further utilizes the fact that many modern synthesizers are able to accurately predict the desired program in many cases, and if a predicted program survives from a number of questions that are selected to challenge it, the predicted program is probably the correct program. For example, suppose the samples are \(p_1, p_3, p_7\), a synthesizer predicts \(p_1\) to be the target program, and \textit{EpsSy} asks for \((-1, 1)\). If \(p_1\) is incorrect, the probability for it to survive from this question will be about \(1/3\), since it performs differently from 2/3 samples. We show that the error rate of \textit{EpsSy} is bounded, and is controlled by adjusting the number of samples and the number of challengeable questions.

To implement the two algorithms, we need a sampler to sample from the remaining program domain according to a distribution. In other words, the sampler not only finds programs that are consistent with the existing questions and answers, as a standard synthesizer does, but also needs to ensure the programs are drawn from a distribution. We introduce \textit{VSampler} to perform this task. \textit{VSampler} samples programs from a version space algebra (VSA) [49] according to a probabilistic context-free grammar (PCFG) [25]. PCFG is a standard way to model distributions over a set of programs represented by a CFG. VSA is a commonly-used data structure to represent a program domain reduced from input-output questions, which are the most common type of questions. Since the size of a program is also an important indicator of probability and PCFG does not directly support size-related distribution, \textit{VSampler} also introduces a method
The results suggest that both algorithms significantly reduce the number of questions: a random strategy uses 38.5% and 13.9% more questions than SampleSy, and 54.4% and 35.0% more questions than EpsSy, respectively for program repair and string manipulation. Also, the error rate of EpsSy is small, where the overall error rate is 0.60%.

To sum up, this paper makes the following main contributions.

- **SampleSy** that selects questions with a short response time by approximating minimax branch with a bounded probability.
- **EpsSy** that further reduces the number of questions by allowing a bounded error.
- **VSampler** that samples programs from a VSA according to a PCFG, and can be used to implement SampleSy and EpsSy.
- An evaluation on a set of SyGuS problems showing the effectiveness of both SampleSy and EpsSy.

## 2 Question Selection Problem in Interactive Program Synthesis

In this section we define the question selection problem and introduce the minimax branch strategy.

### 2.1 Question Selection Problem

**Definition 2.1 (Oracle Function).** Given a program domain \( P \), a question domain \( Q \), and an answer domain \( A \), an oracle function \( D : P \rightarrow Q \rightarrow A \) associates a program \( p \) and a question \( q \) with an answer, denoted as \( D[p](q) \).

Obviously, interactive synthesis can only distinguish programs if the oracle function could distinguish them.

**Definition 2.2 (Distinguishable Programs).** Two programs \( p_1 \) and \( p_2 \) are distinguishable w.r.t an oracle function \( D \) if and only if \( \forall q \in Q, D[p_1](q) \neq D[p_2](q) \), otherwise we say the two programs indistinguishable.

The goal of interactive program synthesis is to find a program \( p \) that is indistinguishable from a target program \( r \). For the convenience of further description, we define the concept of valid programs, which are programs consistent with a set of question-answer pairs. In the following definitions, we implicitly assume the existence of a universal oracle function \( D \) for any program, question, and answer domains.

**Definition 2.3 (Valid Programs).** Let \( C \in (Q \times A)^* \) be a sequence of question-answer pairs, and \( P \) be a set of programs. The valid programs with respect to \( C \), denoted as \( P_C \), are programs which are consistent with all these question-answer pairs, i.e., \( \{p|p \in P, \forall (q,a) \in C, D[p](q) = a\} \).

Then we come to the interaction part. We first define what is considered as a question to ask.

**Definition 2.4 (Question Selection Function).** Given a program domain \( P \), a question domain \( Q \), and an answer domain \( A \), a question selection function \( QS : (Q \times A)^* \rightarrow \{\top\} \cup Q \) is a function satisfying the following two conditions for any \( C \in (Q \times A)^* \)

\[
q^* = \top \iff \forall p_1, p_2 \in P_C, q \in Q, D[p_1](q) = D[p_2](q)
\]

\[
q^* \neq \top \iff \exists p_1, p_2 \in P_C, D[p_1](q^*) \neq D[p_2](q^*)
\]

Here, \( \top \) represents the completed signal of the interaction.

With a given question selection function \( QS \), the interactive synthesis process will be carried out as follows (starting with \( C = \emptyset \)):

1. If \( QS(C) = \top \), return any program in \( P_C \).
2. Show \( QS(C) \) to the developer and get the answer \( a \).
3. Add \( (QS(C), a) \) to \( C \) and return to step 1.

We use \( len(QS, r) \) to represent the number of questions during the interaction.

Let \( \varphi \) be a priori probabilistic distribution on the program domain \( P \), which measures the likelihood for each program to be the target. The goal of the question selection problem is to find the optimal question selection function with respect to \( \varphi \).

**Definition 2.5 (Optimal Question Selection Function).** Let \( P \) be a program domain and \( \varphi \) be a distribution on \( P \). An optimal question selection function \( OQS \) is the question selection function that minimizes \( \sum_{r \in P} \varphi(r) len(OQS, r) \), i.e., the expected number of questions. For convenience, we call this expectation as the cost of a question selection function.

The optimal question selection problem, denoted as \( OQS \), is to implement an optimal question selection function. As can be found in the supplementary material [1], we show that \( OQS \) is polynomial-time transformable to the problem of constructing optimal decision tree [28], and vice versa. Since the optimal decision tree problem has been proved to be NP-hard, \( OQS \) is also NP-hard.

**Theorem 2.6.** \( OQS \) is a NP-hard problem with respect to the size of \( P, Q \) and \( A \).

Due to space limit, we omit all proofs, and the details of the proofs can be found in the supplementary material [1].

### 2.2 A Greedy Strategy: Minimax Branch

As mentioned before, minimax branch is a polynomial-time strategy to approximate the optimal decision tree, and here we adapt it for optimal question selection. **minimax branch** is based on the following intuition: in order to minimize the expected number of questions, in each turn, the chosen question should disqualify as many programs as possible. For a program set \( P \), define its weighted size \( w(P) = \sum_{p \in P} \varphi(p) \) i.e., the probability for a random program to be in \( P \).
branch aims to minimize the worst-case weighted size in each turn.

**Definition 2.7 (minimax branch).** minimax branch is a question selection function minimax defined as follows:

- If the interaction is finished, i.e., if \( \forall p_1, p_2 \in P|C, \forall q \in Q, \mathbb{D}[p_1](q) = \mathbb{D}[p_2](q) \), minimax(C) will be \( T \).
- Otherwise, minimax(C) is defined to be:

\[
\text{minimax}(C) = \arg \min_{q \in Q} \left( \max_{a \in A} \left( \mathbb{P}_{C \cup \{(q, a)\}}(x) \right) \right)
\]

Based on the results from the optimal decision tree problem [2, 10, 11], minimax branch achieves the best approximation ratio in multiple variants of optimal decision tree problems and is a state-of-the-art strategy.

**Theorem 2.8.** The minimax branch strategy approximates the optimal question selection function with an approximation ratio of \( O(\log^2 m) \), where \( m \) is the size of the program domain.

### 3 Optimized Algorithm I: SampleSy

In an interactive session, the response time is expected to be short, preferably in a couple of seconds. As mentioned before, implementing minimax branch requires to enumerate three large domains, which is almost impossible for an interactive algorithm.

In this section we introduce SampleSy that approximates minimax branch but has a short response time.

#### 3.1 Overview

The basic idea of SampleSy is to sample a small number of valid programs, and then apply minimax branch only to these samples. In this way, the number of programs considered by minimax branch is sharply reduced. After that, instead of enumerating \( Q, A \), SampleSy uses an SMT solver to find out the best question directly.

SampleSy is formed by three parallel processes:

- **Sampler.** A background process that continually samples programs from \( P|C \).
- **Decider.** A background process that decides whether the termination condition is met, i.e., whether \( \forall p_1, p_2 \in P|C, \forall q \in Q, \mathbb{D}[p_1](q) = \mathbb{D}[p_2](q) \).
- **Controller.** The main process of SampleSy, a foreground process that deals with the interaction.

Algorithm 1 shows the pseudo code of the controller. In each turn (Lines 2-7), the controller firstly requires a set of samples \( P \) from the sampler \( S \) (Line 2), and then it applies minimax branch to find out the best question \( q^* \) for \( P \) (Line 3). After that, the controller shows \( q^* \) to the developer and waits for the corresponding answer \( a^* \) (Line 4). Finally it updates \( S \) and \( D \) with the new example \( (q^*, a^*) \) (Line 5): \( S \) will filter out the samples of which the answer on \( q^* \) is not \( a^* \), and both \( S \) and \( D \) will update their internal data structure. The interaction repeats until the decider determines that the termination condition is met (Line 6).

---

**Algorithm 1** The controller of SampleSy

**Input:** Instance \( (P, Q, A, \phi) \) of \( \mathcal{OQ8} \), sampler \( S \) and decider \( D \).

**Output:** Synthesized program \( p \in P \)

1: \( P \leftarrow S.Samples \)
2: \( q^* \leftarrow \text{Minimax}(P, Q, A) \)
3: \( a^* \leftarrow \text{Interact}(q^*) \)
4: \( D.AddExample(q^*, a^*), S.AddExample(q^*, a^*) \)
5: while \( D.IsFinished \) do
6: \( S \leftarrow \text{AddExample}(p, q^*) \)
7: \( S \leftarrow \text{AddExample}(p, a^*) \)
8: return Some program in \( S.Samples \)

---

In the following subsections, we shall go into the details of SampleSy.

#### 3.2 Sampler \( S \)

The sampler \( S \) is a program synthesizer that can repeatedly and independently sample programs from the remaining program domain \( P|C \) according to distribution \( \phi \). More precisely, it samples programs from the conditional distribution \( \phi|C \) on \( P|C \) since other programs have been excluded:

\[
\phi|C(p) = P_{x \sim \phi} [x = p|x \in P|C] = \phi(p) w(P|C)^{-1}
\]

In Section 5, we will show how to build an efficient sampler for distributions defined by probabilistic context-free grammars.

The sampler also needs to implement AddExample. When this method is called, the sampler checks whether the current samples are consistent with a new question-answer pair, and discard those that are inconsistent.

#### 3.3 Decider \( D \)

The decider \( D \) is a program synthesizer that can determine whether there are two distinguishable programs in \( P|C \), i.e., the negation of the termination condition. Formally, \( D \) needs to determine whether there exists \( p_1, p_2 \) and \( q \) such that the following formula is satisfied.

\[
\psi_{\text{unfin}}(p_1, p_2, q) \overset{\text{def}}{=} (p_1, p_2 \in P|C) \land (\mathbb{D}[p_1](q) \neq \mathbb{D}[p_2](q))
\]

where \( p \in P|C \iff (\land_{(q^*, a^*) \in C} D[p](q^*) = a^*) \land (p \in P) \).

In order to utilize SMT solvers, throughout this paper, we assume domain constraints \( \exists p \in P, \exists q \in Q \) and the semantics function \( \mathbb{D} \) can be translated into SMT queries and formulas respectively. Clearly, when the program domain \( P \) and the question domain \( Q \) are both finite, these assumptions always hold. Moreover, there have been lots of studies on effective encoding schemes \([23, 37]\); any of them can be used here.

Under these assumptions, \( \psi_{\text{unfin}} \) can be encoded into an SMT query, and \( D \) can be implemented by solving it with an SMT solver.
3.4 Question Selection

This subsection explains how to implement $\text{MINIMAX}(P, Q, A)$ (Line 3) in Algorithm 1. According to minimax branch described in Section 2, the output of $\text{MINIMAX}$ should be:

$$\text{MINIMAX}0(P, Q, A) = \arg \min_{q \in Q} \left( \max_{a \in A} |P(q, a)| \right)$$

where $P(q, a)$ denotes the samples in $P$ which are consistent with question-answer pair $(q, a)$. Note that since $P$ is obtained by sampling, $\text{MINIMAX}$ needs only to minimize the size of $P(q, a)$ rather than the weighted size.

There is a double loop on $Q$ and $A$ in $\text{MINIMAX}0$. To speed it up, we use an SMT solver to substitute the enumeration on $Q$. We define formula $\psi_{\text{cost}}(q, t)$ to represent whether there are more than $t$ programs perform the same on question $q$.

$$\psi_{\text{cost}}(q, t) \overset{\text{def}}{=} \bigwedge_{a \in A} \left( |P(q, a)| \leq t \right)$$

We assume $\mathcal{D}$ can be translated to an SMT formula and thus the whole formula are translatable. This formula tests whether there is a question with cost no more than a given threshold. Therefore, $\text{SampleSy}$ can quickly find the best question by firstly calculating the smallest $t$ that makes $\psi_{\text{cost}}(q, t)$ satisfiable and then finding a corresponding choice of $q$. There are various ways to find the smallest $t$. In our implementation, $\text{SampleSy}$ searches on all possible values of $t$ by binary search and checks the satisfiability by an SMT solver.

So far, the loop on $Q$ has been eliminated. However, the size of the resulting formula $\psi_{\text{cost}}(q, t)$ is $O(|A| \times |P|)$, which is too large for an SMT solver to cope with. To do further optimizations, we utilize the fact that the programs in $P$ can only produce at most $|P|$ different answers on a given question. We construct the following formula $\psi'_{\text{cost}}(q, t)$ and thus reduce the length from $O(|A| \times |P|)$ to $O(|P|^2)$.

$$\psi'_{\text{cost}}(q, t) = \bigwedge_{i=1}^{|P|} \left( \sum_{j=i+1}^{|P|} |\mathcal{D}[p_j](q) - \mathcal{D}[p_i](q)| \leq t \right)$$

To sum up, instead of enumerating $Q$ and $A$, $\text{SampleSy}$ constructs the following formula and uses an SMT solver to find the best question quickly.

$$\text{MINIMAX}(P, Q, A) = \arg \min_{q \in Q} t \ s. t. \ \psi'_{\text{cost}}(q, t)$$

3.5 Parallelization

To further reduce the response time, we utilize the fact that the developer may take a while to answer the question. Therefore in $\text{SampleSy}$, Sampler 8 and Decider $\mathcal{D}$ are implemented as individual background processes so that they can utilize the pending time (Line 4 in Algorithm 1) to sample programs and evaluate the termination condition respectively.

If the sampler is too inefficient, there will not be enough samples for $\text{MINIMAX}$ (Line 3) to find an efficient question. At this time, the developer can choose to wait more time for more samples, or use a potentially inefficient question. In Section 6, we shall demonstrate that our implementation is efficient enough for most real synthesis tasks.

So far, except $\text{MINIMAX}(P, Q, A)$, all other calculations are processed in the background and do not affect the response time. Meanwhile, the time cost of $\text{MINIMAX}(P, Q, A)$ can be controlled by limiting the size of $P$. In the implementation, we limit the response time to two seconds by setting an upper bound for $|P|$. We believe such a response time is small enough for an interactive algorithm.

3.6 Properties of $\text{SampleSy}$

In previous subsections, we discuss the optimizations in $\text{SampleSy}$ and demonstrate that it is efficient on speed. In this subsection, we shall go back to the number of questions and show $\text{SampleSy}$ performs well in this aspect. We first show that $\text{MINIMAX}$ are equivalent to $\text{MINIMAX}0$.

**Lemma 3.1.** For any $\text{OQS}$ instance $(P, Q, A, \varphi)$ and any $P \subseteq P$, let $q_0$ and $q_1$ be the questions selected by $\text{MINIMAX}0$ and $\text{MINIMAX}$, respectively, then $q_0$ and $q_1$ have the same efficiency on $P$, i.e.:

$$\max_{a \in A} |P(q_0, a)| = \max_{a \in A} |P(q_1, a)|$$

We then show that $\text{SampleSy}$ well approximates minimax branch. Theorem 3.2 shows that for any $\epsilon > 0$, when $|P|$ is large enough, the question of $\text{SampleSy}$ is almost surely a $1 + \epsilon$ approximation to that of minimax branch.

**Theorem 3.2.** For any $\text{OQS}$ instance $(P, Q, A, \varphi)$ and any sequence of examples $C \in (Q \times A)^*$, define the cost of an question $\text{cost}(q)$ to be $\max_{a \in A} w(q_i \cap \{a \})$. Let $P$ be a set of independent samples from $\varphi|_{C}$, $q_1$ and $q_0$ be the questions chosen by $\text{SampleSy}$ and minimax branch respectively, then $\forall \epsilon > 0$:

$$\Pr[\text{cost}(q_1) > (1 + \epsilon)\text{cost}(q_0)] \leq 2d|Q| \exp\left(-\frac{|P|\epsilon^2}{2d^2}\right)$$

where $d = \max_{q \in Q} |\mathcal{D}[p](q) | p \in P|$ which represents the maximum number of different answers on a single question.

Theorem 3.2 demonstrates that the probability of getting a bad question decreases exponentially when the number of samples increases, which can be summarized as the following corollary.

**Corollary 3.3.** For any constant $\epsilon > 0$, we have:

$$\Pr[\text{cost}(q_1) > (1 + \epsilon)\text{cost}(q_0)] = O(\exp(-|P|))$$

4 Optimized Algorithm II: $\text{EpsSy}$

As mentioned before, the number of questions can be large, and we reduce them by allowing bounded errors. In this section we first introduce the problem question selection with bounded error and then introduces $\text{EpsSy}$, an algorithm built upon $\text{SampleSy}$ but allowing bounded errors.
4.1 Question Selection with Bounded Error

Definition 4.1 (Unsafe Question Selection Function). Given a program domain $P$ defined on question domain $Q$ and answer domain $A$, an unsafe question selection function $US : (Q \times A)^* \rightarrow P \cup Q$ is a function satisfying the following two conditions for any $C \in (Q \times A)^*$ and $x = US(C)$:

\[ x \in P \implies \forall (q, a) \in C, D[x](q) = a \quad (3) \]
\[ x \in Q \implies \exists p_1, p_2 \in P[C, D[p_1](x) \neq D[p_2](x)] \quad (4) \]

Similar to Definition 2.4, the interactive synthesis process can be determined by an unsafe question selection function $US$ as follows (starting with $C = \emptyset$):

1. If $US(C) \in P$, return $US(C)$.
2. Show $US(C)$ to the developer and get the answer $a^*$.
3. Add $(US(C), a^*)$ to $C$ and return to Step 1.

Let $r$ be the target program, we use $oup(US, r)$ to represent the program generated from the above process and $len(US, r)$ to represent the number of queries used by $US$.

Comparing Definition 4.1 with Definition 2.4, Definition 4.1 loses the first condition and allows the unsafe question selection function to "guess" a program at any time. Besides, the second condition remains unchanged, which guarantees the termination of an unsafe question selection function.

Then we define the concept of error rate and bound the error rate for unsafe question selection functions.

Definition 4.2 (Error Rate). Let $P$ be a program domain, $\varphi$ be a distribution on the programs in $P$, $US$ be an unsafe question selection function. The error rate $e(US, \varphi)$ is the probability for $US$ to return an incorrect program, i.e., a program distinguishable from $r$:

\[ e(US, \varphi) = \Pr_{r \sim \varphi} [\exists q \in Q, D[oup(US, r)](q) \neq D[r](q)] \]

Definition 4.3 ($\varepsilon$-Unsafe Question Selection Function). Given program domain $P$ and distribution $\varphi$ on $P$, an unsafe question selection function $US$ is $\varepsilon$-unsafe iff $e(US, \varphi) \leq \varepsilon$.

4.2 Algorithm EpsSy

Now we come to EpsSy, an efficient algorithm for OUS which is built upon SampleSy. EpsSy is also formed by three parallel processes:

- **Sampler.** A background process that continually samples programs from $P[C]$.
- **Recommender.** A background process that recommends the most probable program in $P[C]$.
- **Controller.** The main process of EpsSy.

As we shall show later, the termination condition of EpsSy is simple enough to be calculated foreground. Therefore EpsSy does not require a background decider to check termination.

Algorithm 2 The controller of EpsSy

**Input:** Instance $(P, Q, A, \varphi, \varepsilon)$ of OUS, a threshold $f_e$, sampler $S$ and recommender $R$.

**Output:** Synthesized program $p \in P$

```latex
1: $r \leftarrow R$.Recommendation
2: $c \leftarrow 0$
3: **do**
4: $P \leftarrow S$.Samples
5: **if** $\exists p^* \in P$, $OccurNumber(P, p^*) \geq (1 - \varepsilon/2)|P|$ **then**
6: **return** $p^*$
7: **end if**
8: $q^*, v \leftarrow GetChallengeableQuery(r, P, Q, A)$
9: $a^* \leftarrow Interact(q^*)$
10: $S$.AddExample($q^*, a^*$), $R$.AddExample($q^*, a^*$)
11: **if** $D[r](q^*) = a^*$ **then**
12: $c \leftarrow c + v$
13: **else**
14: $c \leftarrow 0, r \leftarrow R$.Recommendation
15: **end if**
16: **while** $c < f_e$
17: **return** $r$
```

Algorithm 2 shows the pseudo code of the controller. EpsSy maintains a recommendation $r$ (Line 1) and a confidence value $c$ to it (Line 2). In each turn (Lines 3 to 16), EpsSy firstly requires a set $P$ of samples from sampler $S$ and uses an SMT solver to check whether $P$ contains at least $(1 - \varepsilon/2)|P|$ indistinguishable samples (Line 5). If $P$ does, EpsSy returns one of them as the answer directly (Line 6). Otherwise, EpsSy uses a question $q^*$ to challenge the recommendation (Line 8), and $v$ is the difficulty of this question. After the developer feeds back the corresponding answer (Line 9), EpsSy updates $S$, $R$ (Line 10) and check whether $r$ survives from this question (Line 11). If so, the confidence will be increased by the difficulty (Line 12). Otherwise EpsSy will recalculate $r$ and clear the confidence (Line 14). EpsSy returns $r$ as the answer once the confidence reaches a threshold $f_e$ (Line 16).

Now we go into the details of EpsSy.

4.2.1 Recommender $R$. EpsSy utilizes the fact that a modern synthesizer is able to accurately infer the program desired by the user. EpsSy relies $R$ to recommend likely programs and uses questions to verify them.

Note that the recommender can be any synthesizer that is able to synthesize a program consistent with the answers. Moreover, as we shall show later, the output of this synthesizer does not affect the bound on the error rate. Despite this, the more accurate the recommendation is, the earlier the target program will be found, and the fewer questions will
be asked. To reduce the number of questions, EpsSy expects \( R \) to be accurate in recommending the desired program.

4.2.2 Termination Condition. Compared with SampleSy, EpsSy uses two different termination conditions:
- A certain proportion of the samples is indistinguishable (Line 5). The samples reflect the distribution on \( \mathbb{P}|_C \): if most of the samples are indistinguishable, with high probability, most of the programs in \( \mathbb{P}|_C \) will be indistinguishable, too.
- The confidence reaches a threshold (Line 16). If the recommendation \( r \) is incorrect, it has non-zero probability to be excluded in each turn. Therefore, if a recommendation survives from many questions, with high probability, it will be the correct answer.

Both of these two conditions can be evaluated efficiently: The first condition can be verified by checking distinguishability for \( \mathcal{O}(|\mathbb{P}|) \) pairs of samples, which can be efficiently done by an SMT solver with the following formula:
\[
\psi_{\text{dist}}(p_1, p_2) = \exists q \in \mathbb{Q}, \mathbb{D}[p_1](q) \neq \mathbb{D}[p_2](q)
\]
and the second condition is just a comparison. Therefore EpsSy puts them in the foreground. In Subsection 4.4, we shall show the error rate of EpsSy is bounded by these two conditions.

4.3 Question Selection
This subsection is about GetChallengeableQuestion (Line 8) in Algorithm 2. This function aims to find a question which has a high probability of excluding an incorrect recommendation. To ensure it EpsSy tries to locate a question where the proportion of the sample programs returning an answer different from \( r \) is at least \( w \). In this way, if \( r \) is not the desired program, the developer’s answer will have around \( w \) probability to exclude \( r \). More formally, we introduce a formula \( \psi_{\text{good}}(q, w) \) as follows (\( \mathbb{P}\setminus r \) denotes the programs that are distinguishable from \( r \) in \( \mathbb{P} \), and can be obtained by checking the distinguishability between programs in \( \mathbb{P} \) and \( r \) by \( \psi_{\text{dist}} \) since both \( \mathbb{P} \) and \( r \) are known):
\[
\psi_{\text{good}}(q, w) \overset{\text{def}}{=} \left( \sum_{p \in \mathbb{P}\setminus r} \mathbb{D}[p](q) = \mathbb{D}[r](q) \right) \leq (1 - w)|\mathbb{P}|
\]

To faster exclude incorrect recommendations, the question should satisfy \( \psi_{\text{good}}[r](q, w) \) with a larger \( w \); to minimize the size of \( |\mathbb{P}|_C \), the question should instead satisfy \( \psi_{\text{cost}}(q, t) \) with a small \( t \). EpsSy makes a tradeoff between these two aspects as shown in Algorithm 3. If there exists a question which is good (Line 1), EpsSy will find the question with the smallest cost among all good questions, and return it with difficulty \( v = 1 \) (Lines 2, 3). Otherwise, EpsSy does the same as SampleSy and return with difficulty \( v = 0 \) (Lines 5, 6). Intuitively, for an incorrect recommendation to be returned, it has to survive on \( f_e \) good questions, of which the probability is only around \( (1 - w)^{f_e} \); thus the error rate is bounded.

Algorithm 3 The question selection in EpsSy

Input: Recommendation \( r \), samples \( P \), question domain \( \mathbb{Q} \) and answer domain \( \mathbb{A} \).
Output: A question \( q^* \) and the corresponding difficulty \( v \).

1. if \( \psi_{\text{good}}[r](q, w) \) is satisfiable then
2. \( q^* \leftarrow \arg \min_q t \; \text{s.t.} \; \psi_{\text{good}}[r](q, w) \land \psi_{\text{cost}}(q, t) \)
3. return \( q^*, 1 \)
4. else
5. \( q^* \leftarrow \arg \min_q t \; \text{s.t.} \; \psi_{\text{cost}}(q, t) \)
6. return \( q^*, 0 \)
7. end if

Example 4.4. Assume the program domain is \( \mathbb{P}_e \) (the program domain discussed in Section 1), the samples \( P \) are \( p_1, p_2, p_4, p_5, p_7, p_8 \) and the recommendation is \( p_7 \), then:
- When \( w = 0.5 \), one best question is input \((-1, 1) \) on which \( p_7 \) performs differently with \( p_1, p_4, p_5 \), accounting for 60\% in \( \mathbb{P}\setminus r \), and \((-1, 1) \) can exclude 3 samples in the worst case.
- When \( w = 0.9 \), one best question is input \((0, -1) \) on which \( p_7 \) performs differently with all samples in \( \mathbb{P}\setminus r \), and \((0, -1) \) can only exclude 1 sample in the worst case.

A delicate point is how to determine the value of \( w \). To achieve a certain error rate, the larger \( w \) is, the smaller \( f_e \) will be, and thus the fewer questions required by EpsSy will be. On the other hand, the larger \( w \) is, the fewer questions on which \( \psi_{\text{good}}[r](q, w) \) is satisfiable will be: it may affect the performance of EpsSy on excluding samples, as shown in Example 4.4. Lemma 4.5 shows that \( w = 1/2 \) is a threshold of the satisfiability of \( \psi_{\text{good}}[r](q, w) \).

Lemma 4.5. For any program set \( P \) and constant \( w \), define \( B(P) \subseteq P \) as the set of programs which makes \( \psi_{\text{good}}[P](q, w) \)
unsatisfiable, i.e., \( B(P) = \{ p \in P | \forall q \in \mathbb{Q}, \neg \psi_{\text{good}}[p](q, w) \} \):
- \( w \leq 1/2 \Rightarrow 4P, \) programs in \( B(P) \) are indistinguishable from each other.
- \( w > 1/2 \Rightarrow \forall t > 0, \exists P \) s.t. \( B(P) \) contains at least \( t \) programs which are distinguishable from each other.

Lemma 4.5 shows that if \( w \leq 1/2 \), whatever \( P \) is, the probability that \( \psi_{\text{good}}[r](q, w) \) is unsatisfiable is always small; but once \( w \) is larger than \( 1/2 \), this probability can be extremely large. Therefore, in our implementation, EpsSy sets \( w \) to \( 1/2 \).

4.4 Properties of EpsSy
Theorem 4.6 demonstrates the most important property of EpsSy: it can reach an arbitrarily small error rate with logarithmic level \( f_e \).

Theorem 4.6. For an \( \mathcal{O}(\mathbb{L}) \) instance \( (P, \mathbb{Q}, A, \varphi, e) \), let \( n \) be a lower bound of the number of samples in each turn, \( \beta \) be an
upper bound of questions required by EpsSy, then:
\[ \forall n \geq \max \left( 18 \ln \left( \frac{2|Q|}{\epsilon} \right), \frac{16 \ln 2}{\epsilon^2} + \frac{8}{\epsilon^2} \ln \left( \frac{1}{\epsilon} \right) \right), \]
\[ \forall \alpha \in V, \sum_{(\alpha, \beta) \in R} \gamma(\alpha, \beta) = 1. \]

5 Sampler

In this section, we shall introduce a sampler VSampler which uses version space algebra to represent the set of valid programs \( \mathbb{P}_G \) and can efficiently sample programs from distributions defined by probabilistic context-free grammars. VSampler supports input-output questions, one of the most commonly used types of questions.

5.1 Preliminaries

We start from Context-free Grammar (CFG).

Definition 5.1 (Context-Free Grammar). Context-free grammar (CFG) \( G \) is a 4-tuple \((N, \Sigma, R, S)\) where \( N \) is a set of non-terminal symbols, \( \Sigma \) is a set of terminal symbols, \( R \) is a subset of \( N \times (N \cup \Sigma)^* \) which represents a set of rules and \( S \) is the start symbol. A rule \((\alpha, \beta)\) (or denoted as \( \alpha := \beta \)) represents the nonterminal symbol \( \alpha \) can produce sequence \( \beta \). \( G \) contains all the sequences comprised of only terminal symbols that can be produced by the start symbol \( S \) directly or indirectly.

Throughout this section, we assume the grammar is always unambiguous, i.e., for any sequence contained in the grammar, there exists a unique leftmost derivation from the start symbol to the sequence.

VSampler assumes the program space can be represented by a version space algebra (VSA). VSA is a subset of CFG where only the following three kinds of rules are allowed and each non-terminal can only appear in the left once:

\[
\begin{align*}
s & := p_1|p_2|\ldots|p_k \quad s \in N, \; p_i \in \Sigma^* \\
s & := s_1|s_2|\ldots|s_k \quad s \in N, \; s_i \in N \\
s & := F(s_1, \ldots, s_k) \quad s \in N, \; F \in \Sigma, \; s_i \in N
\end{align*}
\]

Note that the first two kinds of rules can be split into \( k \) different rules \( s := p_i \) \((s := s_i)\) in CFG.

Example 5.2. The program domain \( \mathbb{P}_e \) can be represented by the following VSA:

\[
S := E|S_1 \quad S_1 := \text{if (E, E)} \quad E := 0|x|y
\]

where if \((E, E)\) is an abbreviation for if \((E \leq E)\) then \( x \) else \( y \) and the start symbol is \( S \).

Now we come to the PCFG model which is an extension of CFG by assigning probabilities to the rules.

Definition 5.3 (Probabilistic Context-Free Grammar). A probabilistic context-free grammar \( G \) is a context-free grammar (CFG) \((N, \Sigma, R, S)\) combined with a function \( \gamma : R \mapsto \mathbb{R}^+ \), which satisfies \( \forall \alpha \in V, \sum_{(\alpha, \beta) \in R} \gamma(\alpha, \beta) = 1. \)

The PCFG model uses \( \gamma(\alpha, \beta) \) to represent the probability of choosing rule \((\alpha, \beta)\) to expand a non-terminal symbol \( \alpha \). Therefore, for a program which is produced by rule \((\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n)\), its probability is equal to \( \prod_{i=1}^{n} \gamma(\alpha_i, \beta_i) \).

Since VSA is a subset of CFG, the PCFG model can also be defined on a VSA.

Example 5.4. The following PCFG defines a uniform distribution on the VSA in Example 5.2.

\[
\begin{array}{c|c|c|c}
S & := E & 1/4 & S := S_1 & 3/4 \\
E & := 0 & 1/3 & E := x & 1/3 \\
E & := y & 1/3 & E := y & 1/3
\end{array}
\]

Rule "\( S := E \)" and "\( E := 0 \)" are used to obtain "0", therefore the probability of "0" is equal to \( \frac{1}{4} \times \frac{1}{3} = \frac{1}{12} \). For program "\( x \leq y \)" or "\( S := \text{if (E, E)} \)" and "\( S_1 := \text{if (E, E)} \)" are used once while "\( E := x \)" is used twice, therefore its probability is \( \frac{3}{4} \times \frac{1}{3} \times \frac{1}{3} = \frac{1}{12} \).

In program synthesis frameworks such as FlashMeta [17, 29, 41], a VSA is transformed to represent a subset of the program domain that satisfies the specification. Such a transformation is performed by annotating the non-terminals in a VSA with constraints that the programs expanded from the non-terminal must satisfy. When a non-terminal represents an expression and the specification is given as input-output examples, a common annotation is the return value of the expression. More concretely, given a VSA \( G = (N, \Sigma, R, S) \) and a set of input-output examples \( \{I_i \rightarrow O_i\}_{i=1}^{n} \), a new VSA \( G' \) is constructed to represent programs in \( G \) which are consistent with all examples. In the new \( G' \), each symbol has the form \( \langle s, o_1, \ldots, o_n \rangle \) which represents a set of programs which \((1)\) can be produced by non-terminal symbol \( s \) in \( G \), and \((2)\) output \( o_i \) on the \( i \)th input \( I_i \). As a result, symbol \( \langle s, O_1, \ldots, O_n \rangle \) represents a set of programs consistent with all examples. The rules in \( G' \) are constructed according to the rules in \( G \) and the semantics of the operators: every rule in \( G' \) can be mapped back to some rule in \( G \). Example 5.5 demonstrates this process.

Example 5.5. The following grammar is the transformed version from the VSA in Example 5.2 based on input-output pair \((0, 1) \rightarrow 0\). We take \((S, 0)\) as an example to show how the rules are constructed. \((S, 0)\) represents programs which are expanded from \( S \) and output 0 on input \((0, 1)\), and there are two ways to get such programs: either through sub-rule \( S \rightarrow E \) or through sub-rule \( S \rightarrow S_1 \). Both cases require a program which is expanded from the right symbol and still outputs 0. Therefore the rule of \((S, 0)\) should be \((E, 0) \mid \langle S_1, 0 \rangle\).

\[
\langle S, 0 \rangle := \langle E, 0 \rangle \mid \langle S_1, 0 \rangle \quad (E, 0) := x \mid 0 \quad (E, 1) := y
\]

\[
\langle S_1, 0 \rangle := \text{if (} E, 0 \text{), (} E, 0 \text{)} \mid \langle E, 0 \rangle \mid \langle S_1, 0 \rangle \mid \langle E, 1 \rangle \mid \langle E, 1 \rangle
\]

Strictly speaking, this grammar is not a VSA, but it can be converted to a VSA by adding auxiliary non-terminal symbols to split the last rule.
5.2 Sampling from VSA

Given program domain $P$ defined on VSA $G$, examples $C$ and a distribution $\varphi$ over $P$ defined by a PCFG, $VSampler$ firstly transforms $G$ to the VSA $G'$ representing $P|_{C}$ as in existing approaches [17, 29, 41], and then samples programs from $G'$ according to the distribution $\varphi|_{C}$.

$VSampler$ uses two functions GetPr($N$) and Sample($N$) to perform sampling. Given a PCFG ($V, \Sigma, R, S, y$), the definitions of these two functions are shown in Figure 1. For each non-terminal symbol $N$, $VSampler$ firstly uses GetPr to preprocess the program distributions of the programs in $N$. Utilizing those probabilities, function Sample samples programs according to $\varphi|_{C}$ by recursively sampling the subprograms.

Example 5.6. Assume $VSampler$ is going to sample programs from VSA in Example 5.5 and the PCFG in Example 5.4. Firstly, it uses GetPr to calculate the probability for each non-terminal symbol, and the results are:

- $\text{GetPr}((E, 0)) = 2/3$
- $\text{GetPr}((E, 1)) = 1/3$
- $\text{GetPr}((S, 0)) = 7/9$
- $\text{GetPr}((S, 0)) = 3/4$

Now consider the probability for “if $x \leq y$ then $x$ else $y$” to be sampled. Starting from $(S, 0)$, with probability $7/9$, Sample selects programs from $(S, 0)$. Then, with probability $2/7$, it recurses into “if $(E, 0)$, $(E, 1)$” 

The first parameter has $1/2$ chance to be sampled, and the second parameter must be sampled. In conclusion, the probability of “if $x \leq y$ then $x$ else $y$” is $7/9 \times 2/7 \times 1/2 = 1/9$, which is consistent with $\varphi|_{C}$.

5.3 Properties of $VSampler$

Theorem 5.7 demonstrates the correctness of $VSampler$ on finite program domains.

Theorem 5.7. For a program domain $P$, an example sequence $C$ and a distribution $\varphi$ defined by a PCFG, let $G$ be an acyclic VSA that represents $P|_{C}$, then Sample($S$) is subject to the conditional distribution $\varphi|_{C}$, where $S$ is the start symbol of $V$.

Besides the correctness, $VSampler$ is also efficient on the time cost. Let $m$ be the number of rules in the VSA, $k_{0}$ be the maximum $k$ among all rules and $s_{0}$ be an upper bound of the programs’ size, then the time complexities of GetPr and Sample are $O(m k_{0})$ and $O(s_{0} k_{0})$, respectively. So the time complexity for $VSampler$ to generate $t$ samples is $O(m k_{0} + t s_{0} k_{0})$. Note that constructing such a VSA requires at least $O(m k_{0})$ time, which means performing sampling is not the bottleneck of $VSampler$, especially when the number of samples is not large.

5.4 Representing Size-Related Distribution

The success of enumeration based synthesizers [3, 4] has shown that the size of a program is an important indicator of finding the most probable program. However, the PCFG model cannot encode any size-related distribution, since it is based on a context-free grammar. $VSampler$ resolves this problem by constructing an auxiliary CFG which encodes size information into non-terminals.

Definition 5.8 (Auxiliary CFG). Given a CFG ($V, \Sigma, R, S$) and a size limit $n$, its auxiliary CFG is a context-free grammar ($(V \times [n]) \cup (S', \Sigma, R', S')$, where $R'$ is defined as:

- For any $(\alpha, \beta) \in R$ with $k$ non-terminals in $\beta$, for any positive integers $s_{1}, \ldots, s_{k} \in [n]$, $(\alpha, 1 + \sum_{i} s_{i}, \beta')$ is added to $R'$, where $\beta'$ is constructed from $\beta$ by replacing the $i$th non-terminal $v_{i}$ with $(v_{i}, s_{i})$.
- For any $s \in [n]$, $(S', (S, s))$ is added to $R'$.

For any CFG $G$ and size limit $n$, its auxiliary CFG $G'$ contains exactly all programs in $G$ with size at most $n$. Thus PCFGs on $G'$ can represent size-related distributions on $G$.

Example 5.9. The following PCFG represents a distribution on the VSA in Example 5.2, where the probability of a program is inversely proportional to its size:

$$
\begin{align*}
S' & \xrightarrow{1/2} (S, 1) \\
S' & \xrightarrow{1/2} (S, 3) \\
(S, 1) & \xrightarrow{1} (E, 1) \\
(S, 3) & \xrightarrow{1} (S, 3) \\
(S, 3) & \xrightarrow{1} (S, 3)
\end{align*}
$$

In this model, the probability of “0” is $1/2 \times 1 \times 1/3 = 1/6$ and the probability of “if $x \leq y$ then $x$ else $y$” is $1/2 \times 1 \times 1/3 = 1/6$.

6 Evaluation

To evaluate $SampleSy$ and $EpsSy$, we report several experiments designed to answer the following research questions:

- **RQ1:** How do $SampleSy$ and $EpsSy$ compare against existing strategies?
- **RQ2:** How does the prior distribution affect the performance of $SampleSy$?
- **RQ3:** How does the number of the samples affect the performance of $SampleSy$ and $EpsSy$?
- **RQ4:** How does the value of $f_{c}$ affect the performance of $EpsSy$?

6.1 Implementation

We implemented $SampleSy$ and $EpsSy$ in C++ and used Z3 [13] as the underlying SMT solver for them. Our implementation uses the SyGuS format to specify the synthesis task, and all the questions are in the form of input-output examples. Our implementation is open source and can be found in the supplementary material [1].

For $VSampler$, we implemented the algorithm in FlashMeta [41] to construct VSAs. For $SampleSy$, we used Second Order Solver [37], a state-of-the-art solver-based synthesizer, to implement the decider. For $EpsSy$, we chose $Euphony$ [31], a synthesizer utilizing learned probabilistic models to find the best program, to be the recommender; for the benchmarks which are not supported by $Euphony$, we took $Eusolver$ [4], an efficient enumerative based synthesizer, instead.
GetPr(s) = \begin{cases} 
\sum_{i=1}^{k} y(\sigma(s, p_i)) & \text{SAMPLE}(s) = \{ p_1, i \propto y(\sigma(s, p_i)) \} \\
\sum_{i=1}^{k} y(\sigma(s, s_i)) \text{GetPr}(s_i) & \text{SAMPLE}(s_i), i \propto y(\sigma(s, s_i)) \text{GetPr}(s_i) \\
y(\sigma(s, F)) \prod_{i=1}^{k} \text{GetPr}(s_i) & \text{F(SAMPLE}(s_1), \ldots, \text{SAMPLE}(s_k)) 
\end{cases}

Figure 1. The definitions of function GetPr and Sample used by VSampler. \( i \propto f(i) \) represents \( i \) is sampled from a distribution where \( \text{Pr}[i = a] \) is proportional to \( f(a) \), and \( \sigma \) is a function maps annotated rules in \( G' \) back to the origin rules in \( G \).

6.2 Compared Approach and Configuration

We implemented the random strategy used by Mayer et al. [36], denoted as RandomSy, as the baseline for RQ1. In each turn, RandomSy repeatedly selects a question \( q \) from \( Q \) randomly, until there are two remaining programs performing differently on \( q \), i.e., \( \exists p_1, p_2 \in \mathbb{P}|C, D[p_1](q) \neq D[p_2](q) \). Such a formula can be evaluated by the decider described in Subsection 3.3, and in our implementation, RandomSy and SampleSy share the same decider.

For all strategies, we used a simulator to simulate the human-computer interaction: For each question (Line 4 in Algorithm 1 and Line 9 in Algorithm 2), the simulator firstly delays 1 minute to simulate the time cost for the developer to calculate the answer, and then sends the correct answer to the controller. Since this paper focuses on selecting the best question, we ignored other potential problems during the human-computer interaction, e.g., the user may make mistakes, for simplicity. Designing solutions for these problems is orthogonal to question selection and is future work. Besides, the response time, i.e., the time cost of Minimax in Algorithm 1 and GetChallengableQuery in Algorithm 2, are limited to 2 seconds: when the number of samples is too large, starting from a small subset, we gradually extend the set until the time is used up.

For EpsSy, we set the threshold \( f_e \) to 5 by default. For SampleSy and EpsSy, we constructed a size-related distribution \( \varphi_s \) as the default prior distribution: \( \varphi_s(p) = (5 \times n_{\text{size}}(p))^{-1} \), where \( S \) is the maximum size of programs in \( \mathbb{P} \) and \( n_i \) is the number of programs with the size \( i \) in \( \mathbb{P} \). \( \varphi_s \) has two advantages: (1) The size of \( p \) is subject to a uniform distribution, i.e., \( \varphi_s \) does not have any preference on size, which lets the samples better represent the whole space. (2) For a program space defined by a grammar, \( n_i \) increases exponentially when \( i \) increases. Therefore the smaller the size is, the more probable a program will be chosen.

All of the following experiments were conducted on Intel Core i7-8700 3.2 GHz 6-Core Processor with 32GB of RAM. Our experiment data are available in the supplementary material [1].

6.3 Benchmarks and Experiments

We constructed two datasets Repair and String corresponding to two promising applications of interactive synthesis: program repair and string manipulation (e.g., data wrangling tasks in spreadsheets):

\textbf{Repair}. This dataset is based on the Program Repair track in SyGuS competition which contains 18 benchmarks extracted from the program repair process of real-world Java bugs. In each benchmark, a program needs to be synthesized from a given grammar \( G \) to satisfy a set of input-output examples. We used the following way to construct interactive synthesis benchmarks: (i) The program domain \( \mathbb{P} \) is defined by \( G \) plus a depth limitation. (ii) The question domain \( Q \) is defined by the parameters of the target program, e.g. if the program takes two integers as the input, then \( Q = \mathbb{Z} \times \mathbb{Z} \) where \( \mathbb{Z} \) represents the set of integers. (iii) The target program \( r \) is a program satisfying the input-output examples. Please note that the input-output examples are not available to the interactive synthesis systems.

\textbf{String}. This dataset is based on the string benchmarks collected by Lee et al. [31], which contains 108 string-related benchmarks in SyGuS, 37 questions by spreadsheet users in StackOverflow, and 60 articles about Excel programming in ExcelJet. All the benchmarks correspond to common data manipulation tasks in spreadsheet and a set of input-output examples are given for each benchmark. We converted those benchmarks into interactive synthesis tasks in the following way: (i) The program domain \( \mathbb{P} \) is defined by the grammar used in FlashFill [17] plus a depth limitation. We did not use the original grammars in the benchmarks because they contain int/string conversions that are not supported by FlashMeta framework, which is used to implement VSampler. (ii) The question domain \( Q \) contains all inputs in given examples. The maximum number of examples per benchmark is 400 and the average number is 45.8. We did not include inputs beyond the examples, because the tasks in string benchmarks are all data wrangling tasks, where it is not necessary to process data outside the benchmark. (iii) The target program \( r \) is a program satisfying all input-output examples.

While constructing benchmarks, a depth limit \( d \) was used to make the program space finite, and for all benchmarks, we set \( d \) to the maximum value that can be dealt by the client synthesizers within 1 minute: FlashFill in the sampler, Second-order Solver in the decider, EuSolver and Euphony in the recommender. Besides, we filter out all benchmarks which cannot be solved by those solvers even with smallest possible \( d \). Table 1 lists the information of the final datasets.

Based on the two datasets, we conducted four sets of experiments to answer the four research questions. In all the experiments, we repeated each single execution 5 times and
measured the average value, since all the approaches have randomness.

6.4 Exp 1: Comparison of Approaches (RQ1)

**Procedure.** In this experiment, we ran *SampleSy, RandomSy* and *EpsSy* on all benchmarks in two datasets until a program is synthesized and measured the number of questions required by them.

**Results.** Figure 2 shows the results of this experiment. From this figure, we make the following observations.

![Repair Track](Image)

![String Track](Image)

For each approach, we sort the benchmarks in the increasing order of the number of questions and plot the *i*th benchmark as a point (*i, y_i*) where *y_i* is the average number of questions on it.

**Figure 2.** The results of experiment 1

*SampleSy.* On average, *RandomSy* requires 38.5% and 13.9% more questions than *SampleSy* in *Repair* and *String* dataset respectively. Besides, Figure 2 shows that the performance gap between *SampleSy* and *RandomSy* increases as the difficulty of the benchmark rises: On the hardest 30% benchmarks, *RandomSy* requires 117% more questions than *SampleSy* in *Repair* and 24.8% more questions in *String.*

*EpsSy.* The overall error rate of *EpsSy* is 0.60%. On average, *RandomSy* requires 54.4% and 35.0% more queries than *EpsSy* in the two datasets. When only the hardest 30% benchmarks are considered, those ratios rise to 269% and 84.6%, respectively. Comparing to *SampleSy,* *EpsSy* can further reduce the number of questions even with a small error rate.

The tendency that *SampleSy* and *EpsSy* perform better on hard tasks is caused by the internal greedy strategy: *minimax branch* aims to improve the quality of each question, and its benefits accumulate during the interaction. Therefore, the more the questions are, the longer the interaction will be, and thus the more significant the accumulated advantages of *SampleSy* and *EpsSy* will be.

6.5 Exp 2: Comparison of Prior Distributions (RQ2)

**Procedure.** In this experiment, we tested the sensitivity of our approaches on the choice of prior distribution. As discussed in Section 6.2, *SampleSy* and *EpsSy* take \( \phi_s(p) = (S \times \text{size}(p))^{-1} \) as the prior distribution by default, where *S* is the maximum size of programs in \( \mathbb{P} \) and *n_i* is the number of programs with size *i* in \( \mathbb{P} \). Besides, we further considered the following distributions:

- **Enhanced** \( \phi_s \), in which we manually enhanced the accuracy of \( \phi_s \). At each sampling, with a probability of 0.1, the sampler directly returns the target program, and samples a program according to \( \phi_s \) otherwise.
- **Weakened** \( \phi_s \), in which we manually weakened the accuracy of \( \phi_s \). At each sampling, the sampler firstly samples a program \( p \) according to \( \phi_s \). If \( p \) is indistinguishable from the target program, with a probability of 0.5, the sampler resamples a program, and returns \( p \) otherwise.
- **Uniform** Distribution \( \phi_u \), in which the probability of all programs are equal.

Instead of sampling programs from a given distribution, many state-of-the-art synthesizers find the top-*k* programs according to a given ranking function \([31, 41]\). Therefore, we also explored the possibility of directly using these synthesizers as samplers: we evaluated a strategy, denoted as *Minimal*, in which the sampler enumerates programs in the increasing order of the size instead of truly sampling.

For each distribution mentioned above, we measured the average number of questions required by *SampleSy* and *EpsSy* on both datasets.

**Results.** The results are summarized in Table 2. In general, the effectiveness ranking of these distributions is: Enhanced \( \phi_s > \text{Default} \phi_s > \text{Weakened} \phi_s > \text{Uniform} \phi_u \approx \text{Minimal}, * which demonstrates that the accuracy of the prior distribution does have a positive correlation to the performance of *SampleSy* and *EpsSy.* On the other hand, the impact is not significant: even when the distribution is Uniform \( \phi_u \) or Minimal, *SampleSy* and *EpsSy* still perform much better than *RandomSy.* This result suggests that even if the distribution cannot be precisely determined, our algorithms can still perform well by either using a coarse distribution like the uniform distribution or directly using off-the-shelf synthesizers as samplers.

One surprising result is that in *String,* *SampleSy* with Uniform \( \phi_u \) performs better than \( \phi_s \). A possible reason is that in the last few rounds, most incorrect programs (programs distinguishable from the target) have been excluded, and thus the probability of sampling remaining incorrect
programs becomes extremely small. Note that the target of SampleSy is to exclude all incorrect programs: the lack of incorrect samples to the controller in each turn. We ran SampleSy on all benchmarks with a limitation \( w \) on the number of samples, i.e., the sampler only returns at most \( w \) samples to the controller in each turn. We ran SampleSy three times with \( w = 2, 20, 5000 \), respectively and measured the number of questions required.

### 6.6 Exp 3: Comparison of the Sample Size (RQ3)

**Procedure.** In this experiment, we ran SampleSy on all benchmarks with a limitation \( w \) on the number of samples, i.e., the sampler only returns at most \( w \) samples to the controller in each turn. We ran SampleSy three times with \( w = 2, 20, 5000 \), respectively and measured the number of questions required.

### Results

The results are summarized in Figure 3. For convenience, we use \( S(k) \) to represent SampleSy with \( w \) equal to \( k \). Figure 3 shows that \( S(2) \) performs significantly worse than \( S(5000) \): On the hardest 30% benchmarks, \( S(2) \) takes 50.0% more questions than \( S(5000) \) in Repair and 12.7% more questions in String. However, such a performance gap disappears when \( w \) is only increased by 18: the curves representing for \( S(20) \) and \( S(5000) \) almost coincide in Figure 3. More precisely, \( S(20) \) only uses 3.6% more questions than \( S(5000) \) in Repair and 0.5% more questions in String.

This result shows that the performance of SampleSy converges rapidly when \( w \) increases, which is consistent with Theorem 3.2.

### 6.7 Exp 4: Comparison of Values of \( f_\epsilon \) (RQ4)

**Procedure.** In this experiment, for each \( f_\epsilon \in [0, 0.5] \), we ran EpsSy on all benchmarks, and recorded whether the result is correct and the number of questions required.

**Results.** The results of the experiment are summarized in Figure 4. We make the following observations from it:

- For both datasets, the error rate drops exponentially when \( f_\epsilon \) increases, we are able to obtain a high accuracy with a small \( f_\epsilon \). This result is consistent with Theorem 4.6. Moreover, the average number of questions only increases linearly (or even sub-linearly) when \( f_\epsilon \) increases, which means a much better accuracy can be obtained with only a small loss on the number questions.

**Table 2.** The average number of required questions for SampleSy and EpsSy with each considered distribution on each dataset.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>SampleSy Repair</th>
<th>SampleSy String</th>
<th>SampleSy Combined</th>
<th>EpsSy Repair</th>
<th>EpsSy String</th>
<th>EpsSy Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weakened ( \phi_w )</td>
<td>8.100</td>
<td>3.511</td>
<td>3.947</td>
<td>6.538</td>
<td>2.837</td>
<td>3.278</td>
</tr>
<tr>
<td>Default ( \phi_d )</td>
<td>7.375</td>
<td>3.463</td>
<td>3.835</td>
<td>6.538</td>
<td>2.910</td>
<td>3.254</td>
</tr>
<tr>
<td>Enhanced ( \phi_e )</td>
<td>7.113</td>
<td>3.477</td>
<td>3.822</td>
<td>6.025</td>
<td>2.846</td>
<td>3.147</td>
</tr>
<tr>
<td>Uniform ( \phi_u )</td>
<td>9.938</td>
<td>3.327</td>
<td>3.958</td>
<td>6.337</td>
<td>3.227</td>
<td>3.520</td>
</tr>
</tbody>
</table>

![Figure 3](image-url) The results of experiment 3. The figure are drawn in the same way as Figure 2.

![Figure 4](image-url) The results of experiment 4. The x-axis represents the value of \( f_\epsilon \), the left y-axis represents the error rate while the right y-axis represents the average number of questions required by EpsSy.
• The error rate in \textit{String} drops much faster than that in \textit{Repair} while the number of questions increases much slower in \textit{String}. This is because different termination conditions dominate \textit{EpsSy} in the two datasets: For \textit{Repair}, since the numbers of questions required are generally large, \textit{EpsSy} terminates mainly when the confidence reaches \(f_e\). Therefore the number of questions grows around linearly with \(f_e\) while the error rate drops relatively more slowly. For \textit{String}, \textit{EpsSy} terminates mainly when the conditional probability of some semantics is too large. Therefore the number of questions and the error rate is more independent with \(f_e\) than \textit{Repair}.

7 Related Work

\textbf{Question selection in interactive synthesis.} As mentioned before, multiple existing interactive synthesis systems [36, 49] already contain a question selection component. In those systems, the question selection components try to select a \textit{distinguishing} question, i.e., at least two programs have different answer. Among all \textit{distinguishing} questions, the selection is random. For example, Mayer et al. [36] use input-output questions and randomly select input within a test suite until a distinguishing input is found. Wang et al. [49] also use input-output questions and randomly mutate existing inputs until a distinguishing input is found. Such a selection component is equivalent to \textit{RandomSy} in our evaluation. Compared to the existing approaches, our approach selects a question within the whole question space that approximates an optimal question, which significantly outperforms \textit{RandomSy} as our evaluation indicates.

Padhi et al. [39] explore another form of question selection problem for input-output questions. Their approach, called \textit{Significant Inputs}, returns a set of questions in each turn and lets the user select which questions to answer among all selected questions. \textit{Significant Inputs} uses a syntactic profile over structured inputs to perform the selection. Compared with \textit{Significant Inputs}, our approach returns one question each turn and does not require a syntactic profile, applicable to unstructured inputs such as integers. Furthermore, our approach is the first to give the theoretical guarantees on disambiguation and the number of required queries. The two approaches can potentially be combined: \textit{Significant Inputs} selects a set of questions while our approaches choose among these questions.

\textbf{User-driven interactive synthesis.} Besides asking easy-to-answer questions, another form of interaction in program synthesis is to demonstrate a set of solutions to the user, and the user decides what actions to take to reduce the ambiguity. The actions include (1) selects a program from the demonstrated programs [36, 52], (2) provides input-output examples to refine the specification [16, 26, 52], and (3) changes some configuration options of the synthesis algorithm [5, 22, 26, 32]. In such interactions there is no question selection problem, as it is the user who decides what action to take. In contrast to these approaches that give an open task to the user, our algorithms shift the burden of question selection from the user to the system, and the user only has to answer deterministic questions.

\textbf{General framework of interactive program synthesis.} Le et al. [30] build an abstract model of interactive program synthesis along three dimensions: \textit{incrementality, step-based formulation, and feedback-based interaction}. Our algorithms can be viewed as instantiations of the framework, providing feed-back interaction by selecting questions.

\textbf{Counterexample guided inductive synthesis.} Counterexample guided inductive synthesis (CEGIS) [3, 19, 47, 48] assumes the existence of a verification oracle, which could verify whether a synthesized program is correct, and returns a counterexample if it is not. Oracle-guided inductive program synthesis (OGIS) [24] is a generalization of CEGIS, which allows more types of questions to be asked to the oracle. CEGIS and OGIS are similar to interactive program synthesis as humans can be considered as an oracle. However, since OGIS and CEGIS focus on automated oracles, so far there is no practical algorithm for CEGIS/OGIS to minimize the number questions as far as we are aware. Furthermore, many questions considered in CEGIS/OGIS are not easy for human to answer, such as determining whether a program is correct.

\textbf{Optimization Modulo Theories Solvers.} Optimization Modulo Theories (OMT) is an extension of SMT which allows for finding models that optimize given objectives. Since finding the best question according to \textit{minimax branch} is an optimization problem, OMT solvers [6, 7, 27, 33, 44, 45] are related to our work. To deal with an optimization problem, existing OMT solvers require to encode the problem into some pre-designed domain-specific languages. However, encoding \textit{minimax branch}, i.e., Definition 2.7, is a non-trivial challenge: \(w\{P\mid_{\mathcal{C}\cup\{(q,a)\}}\}\) is an instance of \textit{weighted model counting}, which is not supported by any existing OMT solver. Therefore, instead of using OMT solvers to find the best question, we designed \textit{SampleSy} and \textit{EpsSy} to approximate \textit{minimax branch} via sampling.

\textbf{Optimal decision trees.} Previously sections have discussed optimal decision trees [28] and optimal question selection are mutually polynomial-time transformable. However, compared with interactive synthesis, the program/question/answer domains in optimal decision tree are usually much smaller, and thus strategies like \textit{minimax branch} can be directly applied. Our work starts from \textit{minimax branch}, and proposes a set of approximation algorithms to make it practical in interactive synthesis.

\textbf{Active learning.} Active learning is a research domain about finding objects with some target properties from a candidate domain by interactively asking questions to some oracle. In
this domain, the question selection problems are also studied to reduce the number of questions or to enhance the efficiency of a fixed number of questions. However, none of them can be directly applied to interactive program synthesis: (i) Some of them focus on specific models and utilizes concrete properties which do not hold in program synthesis, e.g., Bshouty [8] studies exact learning, a sub problem of active learning that focuses on learning an exact model, but for only boolean functions; Schohn and Cohn [43] study specifically for support vector machines (SVM). (ii) Some of the previous works focus on general question selection problems and their results cannot be applied to interactive program synthesis due to the scalability, e.g., Dasgupta [12] analyzes a greedy strategy of which the time complexity is polynomial to the number of candidates, i.e., the size of the program space. (iii) Others only focus on theoretic results, e.g., Bshouty et al. [9] proves multiple theoretic properties of exact learning, but these properties cannot be directly used to design algorithms. As we are aware, this paper is the first work studying the question selection problem for interactive program synthesis.

8 Conclusion

In this paper, we address the question selection in interactive program synthesis with the goal of minimizing the number of questions to the user. Staring from the minimax branch strategy, we design two efficient algorithms SampleSy and EpsSy for two different scenarios: SampleSy ensures the result to be correct while EpsSy has a bounded error rate. On the theoretical side, we show that both the loss of SampleSy comparing with minimax branch and the error rate of EpsSy are bounded. On the practical side, we implement these two approaches and our evaluation shows that both algorithms outperform the existing strategies significantly.

Appendices

The appendix can be found in the supplementary material [1] and consists of two parts. The first part discusses the relation between optimal question selection and optimal decision tree, and the second part gives the proofs for all other theorems in the paper.

Acknowledgements

We thank Yican Sun and the anonymous PLDI reviewers for valuable feedback on this work. This work is supported in part by the National Key Research and Development Program of China under Grant No. 2017YFB1001803, National Natural Science Foundation of China under Grant Nos. 61922003 and 61672045.

References


