Choose, Don't Label: Multiple-Choice Query Synthesis for Program Disambiguation

CELESTE BARNABY, University of Texas at Austin, USA
DANNY DING, University of Texas at Austin, USA
OSBERT BASTANI, University of Pennsylvania, USA
IŞIL DILLIG, University of Texas at Austin, USA

High-level specifications of code are inherently ambiguous, and prior systems have explored interactive techniques to help users clarify their intent and resolve such ambiguities. However, most existing approaches elicit supervision through labeled examples, which are often error-prone and may fail to capture user intent. This paper introduces a new active learning paradigm for program disambiguation based on *multiple-choice queries*. In this paradigm, the system presents a small set of high-level behaviors as multiple-choice options, and the user simply selects the intended one. Technically, each answer option corresponds to a Hoare triple that characterizes a cluster of semantically similar candidate programs. This formulation enables formal reasoning about the informativeness and interpretability of queries, and supports systematic construction of optimal queries. Building on this insight, we develop a new active learning algorithm and implement it in a tool called Socrates, which automatically synthesizes informative multiple-choice queries for program disambiguation. We evaluate Socrates across four domains spanning both symbolic and neurosymbolic settings and show that it produces intuitive, easy-to-answer queries and achieves efficient convergence. Most importantly, Socrates identifies the intended program *more reliably* than existing methods, while maintaining competitive runtime performance.

CCS Concepts: • **Software and its engineering → Software verification**.

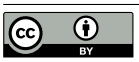
Additional Key Words and Phrases: Program Synthesis, Program Verification, Active Learning, Neurosymbolic Synthesis

ACM Reference Format:
Celeste Barnaby, Danny Ding, Osbert Bastani, and Işıl Dillig. 2026. Choose, Don't Label: Multiple-Choice Query Synthesis for Program Disambiguation. *Proc. ACM Program. Lang.* 10, PLDI, Article 201 (June 2026), 25 pages. https://doi.org/10.1145/3808279

1 Introduction

Recent advances in program synthesis and large language models have made it increasingly practical to generate code from high-level intent. Yet such intent is often underspecified or open to interpretation, giving rise to multiple plausible solutions that are semantically distinct. To address this ambiguity, prior work has explored active learning strategies that identify the desired program through targeted queries posed to the user [4, 24, 33, 34, 38]. These approaches, however, predominantly rely on users to label concrete inputs, which has two key limitations. First, manual input labeling can be cumbersome and error-prone in structure-rich domains, such as those involving

Authors' Contact Information: Celeste Barnaby, University of Texas at Austin, , USA, celestebarnaby@utexas.edu; Danny Ding, University of Texas at Austin, , USA, dingyy@utexas.edu; Osbert Bastani, University of Pennsylvania, , USA, obastani@seas.upenn.edu; Işıl Dillig, University of Texas at Austin, , USA, isil@cs.utexas.edu.



ACM 2475-1421/2026/6-ART201
https://doi.org/10.1145/3808279

large tables or nested data structures. Second, most existing methods assume a fixed set of inputs over which the synthesized program will be evaluated and only guarantee correctness over those.

This paper presents a new approach to active learning for code generation, where users answer high-level multiple-choice questions instead of labeling specific inputs. As shown in Figure 1, these questions let users choose among distinct behavioral patterns: for example, whether the program should return one row per user, one per user–date combination, or one per original record. By replacing tedious example annotation with intuitive questions about high-level behavior, this approach allows users to express semantic intent more directly, while also guiding the synthesizer to reason over broader behaviors. Furthermore, this approach offers stronger correctness guarantees than most prior methods [4, 33, 34] and yields more accurate user responses in practice.

Query: If the input table contains multiple rows for the same user, the program should return:

(a) one row per user
(b) one row per (user, date) pair
(c) one row per original record

Fig. 1. Example multiple-choice query.

At the heart of our approach is a method for partitioning the program space into a small number of semantic equivalence classes. Each class is defined by a shared precondition ϕ and a unique postcondition ψ_i, yielding a structured query $Q = (\phi, \psi_1, \ldots, \psi_k)$. Here, ϕ describes a class of inputs (e.g., *table containing multiple rows for the same user*), whereas each ψ_i specifies a possible property of the corresponding output (e.g., *returning one row per user*). This structure naturally induces a multiple-choice question: the precondition sets up the scenario under which the program's behavior should be evaluated, and the postconditions become the candidate answers. Thus, answering the multiple-choice query reduces to selecting which of several Hoare-style specifications $\{\phi\}\ P\ \{\psi_i\}$ represents the user's intent.

The key challenge is selecting structured queries that are both *informative* (i.e., best disambiguate programs) and *interpretable* (i.e., easily understood by users). To address this challenge, our method first identifies a region of the input space where a maximal number of candidate programs exhibit different behavior. This region is summarized as a precondition that strikes a good balance between discriminative power and simplicity. Once a precondition ϕ is fixed, our method clusters programs based on their semantic behavior under ϕ, ensuring that the clusters are as even as possible so that *any* user answer eliminates a large fraction of programs. Finally, each cluster's behavior is summarized by a concise postcondition that separates each group from all of the others. This yields a complete and mutually exclusive partition of candidate programs and can be directly translated into a multiple-choice query posed in natural language.

We implemented our approach in a new tool called Socrates and evaluated it on four domains: table transformations, JSON transformations, batch image editing, and image search. In our experiments, Socrates achieved perfect accuracy across all tasks, whereas prior active learning methods failed on up to 36% of benchmarks. Furthermore, it posed questions that users answered 38% more accurately, and achieved comparable or better efficiency than existing approaches in terms of query selection efficiency, user response time, and interaction rounds.

To summarize, this paper makes the following key contributions:

- **New paradigm for interactive synthesis.** We introduce a new interactive synthesis framework in which users answer multiple-choice questions about high-level program behavior, rather than labeling concrete inputs. Each question expresses a logical relation between inputs and outputs, allowing users to communicate intent at a semantic level (Section 3).

(a) Photo with Alice and friends. (b) Labeled output. (c) Alice among many people.

Fig. 2. Input–output demonstration (a,b) and a scenario (c) with high annotation burden.

- **Principled query synthesis algorithm.** We design a query synthesis algorithm that formulates user queries as an optimization problem over *informativeness* and *interpretability*, solved through a combination of semantic reasoning, clustering, and interpolation (Section 4).
- **End-to-end implementation and evaluation.** We realize these ideas in a new tool called Socrates (Section 5), and demonstrate its effectiveness across four diverse domains. Our evaluation shows that Socrates achieves perfect accuracy and consistently improves user response correctness, while maintaining comparable or better efficiency compared to example-based approaches (Section 6).

2 Motivating Scenario & Overview

This section illustrates our approach through a simple example inspired by ImageEye, a system for programmatic image manipulation from prior work [5]. ImageEye learns image editing programs from labeled examples, where each input is an original image and the output is its edited counterpart.

To illustrate the limitations of relying on labeled examples, consider a user, Alice, who wants to apply a privacy-preserving transformation to her photo collection: *blur the faces of all people except herself*. In ImageEye, this task can be achieved by a program that detects all faces, compares each against the target, and applies the `Blur` operator to non-matching faces (Program 1 in Figure 3a). Now, suppose Alice provides the input–output example in Figures 2a–2b, where her face remains clear while the others are blurred. Although this example captures her intent, it is ambiguous: several other programs, such as blurring all *non-smiling* faces or those with *non-blonde* hair, would produce the same output and therefore remain viable in the *hypothesis space* (shown in Figure 3a).

Existing methods. Prior approaches address such ambiguity in two main ways. The first is to present a candidate program to the user, who must inspect its source code or outputs and iteratively provide additional examples. This paradigm is both effort-intensive and prone to error. The second employs active learning to select the *most informative* example for labeling [4, 33, 34]. Although this strategy makes the interaction more systematic, it can select examples that are difficult to annotate, such as images containing many people with diverse attributes (e.g., Figure 2c), increasing both annotation time and the likelihood of mistakes. Moreover, because these methods only reason over a fixed set of inputs, their correctness guarantees are limited to that dataset and do not ensure that the resulting program reflects the user's intent on new data.

Our approach. Instead of asking users to label concrete examples, our method poses a multiple-choice question that highlights key differences among candidate programs. For the running example, Figure 3b describes a hypothetical image that contains people with different attributes (i.e., hair color, position, and expression), and asks which faces should remain unblurred. Each answer reflects a distinct interpretation of the user's intent and is designed to eliminate a corresponding subset of programs from the hypothesis space.

Hypothesis space $\mathcal{H}$

Program 1:
```
Is(Face) ∧ ¬FaceMatch(Alice) → Blur
```

Program 2:
```
Is(Face) ∧ ¬Smiling → Blur
```

Program 3:
```
Is(Face) ∧
¬HasAttribute(HairColor=blonde) → Blur
```

Program 4:
```
 Is(Face) ∧
¬HasAttribute(Location=leftmost) →
Blur
```

... and many other programs...

(a) Hypothesis space.

Query. Imagine a photo containing:

- **Alice**: (unblurred face in example) blonde, smiling, and *not* leftmost;
- **Person X**: not blonde, smiling, and *is* the leftmost person;
- **Person Y**: blonde, not smiling, and not the leftmost person.

Which people should remain *unblurred*?

(a) Only Alice.
(b) Alice and Person X.
(c) Alice and Person Y.
(d) None of the above.

(b) Multiple-choice query for running example

Fig. 3. Programs consistent with user's example (a) and the generated multiple-choice query (b)

To generate such a query, Socrates constructs a *text description* of a hypothetical image that makes candidate programs disagree. It first identifies *distinguishing predicates*, or interpretable conditions on object attributes where two programs produce different results. For example, for the first two programs in Figure 3a, a distinguishing predicate might state that "Alice is not smiling, but someone else is." From this set of predicates Φ, Socrates synthesizes a concise precondition ϕ that separates as many program pairs as possible while remaining easy to interpret. In our example, ϕ corresponds to the scene in Figure 3b, which depicts Alice and two others with attributes chosen to maximally differentiate the remaining candidates.

Given the discriminating scene ϕ, Socrates symbolically executes each program in $\mathcal{H}$ and groups those that produce the same result. Each group represents a distinct outcome under ϕ and serves as the basis for a multiple-choice answer. To keep the query concise, Socrates merges these groups into at most four balanced clusters so that any user selection eliminates a roughly equal share of the remaining candidates. Each cluster is then summarized by a *separator*, a logical condition distinguishing it from the others, which is rendered in natural language (using an LLM) to form the answer choices, as shown in Figure 3b.

3 Problem Formulation

In this section, we formalize the problem addressed in the rest of this paper.

3.1 Query Space

Given a finite hypothesis space $\mathcal{H}$ of candidate programs consistent with the user's initial specification, the goal is to identify the intended program $P^* \in \mathcal{H}$ through structured logical queries:

$$Q = (\phi, \psi_1, \ldots, \psi_k),$$

where ϕ is a precondition describing a family of inputs, and ψ_i is a postcondition describing a distinct program behavior for that input family. The pair (ϕ, ψ_i) is called a *scenario*. When the user selects option i, all programs inconsistent with ψ_i are eliminated, thereby refining $\mathcal{H}$.

For a query to be meaningful, its scenarios must partition the hypothesis space $\mathcal{H}$ relative to ϕ. That is, each program in $\mathcal{H}$ should fall into exactly one scenario, ensuring that the user's answer can be used to unambiguously refine $\mathcal{H}$. We capture this requirement as follows.

Definition 3.1. **(Valid query).** Given hypothesis space $\mathcal{H}$, a query $Q = (\phi, \psi_1, \ldots, \psi_k)$ is *valid* if

$$\textbf{(Mutual exclusion)} \quad \forall P \in \mathcal{H}.\ \forall i \neq j.\ \neg\big(\models \{\phi\}\, P\, \{\psi_i\} \wedge \models \{\phi\}\, P\, \{\psi_j\}\big).$$
$$\textbf{(Coverage)} \quad \forall P \in \mathcal{H}.\ \exists i.\ \models \{\phi\}\, P\, \{\psi_i\}.$$

Mutual exclusion ensures that no program in $\mathcal{H}$ can satisfy two different scenarios with the same precondition, so the answer is unambiguous. Coverage ensures that every program in $\mathcal{H}$ is consistent with at least one scenario, ensuring that one of the answers is correct. Now, given query $Q = (\phi, \psi_1, \ldots, \psi_k)$, for each i, define $\mathcal{H}_{\phi,i} = \{\, P \in \mathcal{H} \mid \{\phi\}\, P\, \{\psi_i\} \,\}$ to be the programs in $\mathcal{H}$ that satisfy (ϕ, ψ_i). By Definition 3.1, $\{\mathcal{H}_{\phi,i}\}_{i=1}^{k}$ is a complete partition of $\mathcal{H}$ (i.e., they are disjoint and cover $\mathcal{H}$). If the user selects answer ψ_i, we can refine $\mathcal{H}$ to $\mathcal{H}_{\phi,i}$.

3.2 Query Selection Problem

At a high level, the *query selection problem* is to choose a query Q that balances two goals. The first is *disambiguation power*: each possible answer should eliminate a large portion of the hypothesis space. The second is *interpretability*: the pre- and postconditions should form a question that users can easily understand and answer.

To formalize the problem, we assume two predicate universes: a precondition universe $\mathcal{U}^-$ and a postcondition universe $\mathcal{U}^+$. Each universe defines a finite set of atomic predicates that can be combined to form candidate pre- and postconditions, and we assume that these universes are literal-closed (i.e., $\forall a \in \mathcal{U}$, we also have $\neg a \in \mathcal{U}$). To keep queries interpretable, we restrict both pre- and postconditions to be *cubes*, that is, conjunctions of atoms drawn from their respective universes. Then, the query space is

$$\mathcal{Q} = \{Q = (\phi, \psi_1, \ldots, \psi_k) \mid \phi \in \textsc{Cube}(\mathcal{U}^-),\ \psi_i \in \textsc{Cube}(\mathcal{U}^+),\ Q \text{ is valid}\},$$

where validity is as in Definition 3.1 and $\textsc{Cube}(\mathcal{U})$ is the space of cubes over predicate universe $\mathcal{U}$.

Next, *disambiguation power* captures the efficacy of a query Q at pruning the hypothesis space.

*Definition 3.2 (***Disambiguation power).** Let $Q = (\phi, \psi_1, \ldots, \psi_k)$ be a query, and let $\mathcal{H}_{\phi,i}$ denote the set of programs in $\mathcal{H}$ that satisfy ψ_i under ϕ. Then the disambiguation power of Q is defined as:

$$\mathrm{DP}(Q) = \min_{i \in [1,k]} \frac{\sum_{j \neq i} |\mathcal{H}_{\phi,j}|}{|\mathcal{H}|} = \left(1 - \max_{i \in [1,k]} \frac{|\mathcal{H}_{\phi,i}|}{|\mathcal{H}|}\right)$$

Intuitively, for any answer provided by the user, we can eliminate at least $\mathrm{DP}(Q)$ fraction of programs from the hypothesis space. Thus, we want to select a query that maximizes disambiguation power. However, we also want to ensure the query is interpretable. We define the *complexity* $\mathbb{C}(Q)$ of a query Q based on the syntactic complexity of ϕ and the ψ_i's (e.g., the number of atoms in these logical formulas), reflecting how difficult the query is to interpret. Now, we have:

Definition 3.3. **(Query selection problem)** Given $\mathcal{U}^-, \mathcal{U}^+, \mathcal{H}$, and a hyperparameter $\lambda > 0$ balancing disambiguation power and complexity, the *query selection problem* is to compute

$$Q^* = \mathrm{argmax}_{Q \in \mathcal{Q}} \left[\mathrm{DP}(Q) - \lambda \cdot \mathbb{C}(Q)\right].$$

3.3 Decomposed Problem Formulation

Directly solving the query selection problem from Section 3.2 is computationally challenging since it requires jointly optimizing over pre- and postconditions. This combined search space is too large

Algorithm 1 Main Active Learning Loop

Require: Initial hypothesis space $\mathcal{H}$; predicate universes $\mathcal{U}^-, \mathcal{U}^+$
Ensure: A semantically unique program $P^* \in \mathcal{H}$

```
while true do
    if NumUnique(H) = 1 then return P* ∈ H
    for all distinct pairs (P_i, P_j) ∈ H × H do
        Φ(i, j) ← GetDistinguishing(P_i, P_j, U⁻)
    φ ← GetBestPrecondition(Φ)
    if φ = ⊥ then U⁻ ← RefinePredicates(H, U⁻); continue
    Q ← GenerateQuery(φ, H, U⁺)
    ψ ← QueryUser(Q)
    H ← { P ∈ H | {φ} P {ψ} }
```

for exhaustive evaluation to be feasible. Furthermore, optimizing interpretability at the level of the entire query makes it difficult to isolate and control the complexity of individual conditions. To address these challenges, we adopt a decomposed formulation. We first select a satisfiable precondition ϕ that captures semantically meaningful behavioral distinctions while remaining interpretable. Then, we construct postconditions ψ_i conditioned on ϕ. To formalize this decomposed problem, we first define the *disambiguation power* of a precondition ϕ in isolation as the number of (unordered) program pairs in the hypothesis space $\mathcal{H}$ that it distinguishes:

$$\mathrm{DP}_{\mathrm{pre}}(\phi) = \left|\{(P_1, P_2) \mid P_1, P_2 \in \mathcal{H} \wedge \forall x.\, \phi(x) \Rightarrow P_1(x) \neq P_2(x)\}\right|.$$

*Definition 3.4 (***Decomposed Query Selection Problem***).* Given $\mathcal{U}^-, \mathcal{U}^+, \mathcal{H}$, and hyperparameters λ_{pre} and λ_{post}, the *Decomposed Query Selection Problem* is to compute

$$\phi^* \in \mathrm{argmax}_{\phi \in \textsc{Cube}(\mathcal{U}^-)} \left[\mathrm{DP}_{\mathrm{pre}}(\phi) - \lambda_{\mathrm{pre}} \cdot \mathbb{C}(\phi)\right] \tag{1}$$

$$(\psi_1^*, \ldots, \psi_k^*) \in \mathrm{argmax}_{(\psi_1,\ldots,\psi_k) \in \textsc{Cube}(\mathcal{U}^+)^k} \left[\mathrm{DP}(\phi^*, \psi_1, \ldots, \psi_k) - \lambda_{\mathrm{post}} \cdot \sum_{i=1}^{k} \mathbb{C}(\psi_i)\right]. \tag{2}$$

We note that this decomposed formulation is not equivalent to the joint objective in Definition 3.3: fixing the precondition before optimizing postconditions may exclude queries that would score higher under simultaneous optimization. However, the decomposition offers two practical advantages that justify this trade-off. First, it makes the optimization tractable by replacing a single search over the combined space of all $(\phi, \psi_1, \ldots, \psi_k)$ tuples with two smaller, sequential subproblems. Second, it allows interpretability to be controlled at the level of individual conditions, since the complexity of the precondition and postconditions can be penalized independently. Intuitively, a precondition ϕ that distinguishes many candidate programs naturally enables informative queries: when ϕ induces diverse program behaviors, it becomes easier to construct postconditions that divide the hypothesis space evenly, leading to high disambiguation power.

4 Active Learning Algorithm

Algorithm 1 presents our top-level procedure for solving the decomposed query selection problem. Given an initial hypothesis space $\mathcal{H}$ and predicate universes $\mathcal{U}^-$ and $\mathcal{U}^+$, the algorithm iteratively interacts with the user until all remaining programs in $\mathcal{H}$ are semantically equivalent. Each iteration begins by identifying preconditions under which pairs of programs $P_i, P_j \in \mathcal{H}$ exhibit

Algorithm 2 GetDistinguishing$(P_1, P_2, \mathcal{U}^-)$

Require: Programs P_1, P_2; universe $\mathcal{U}^-$
Ensure: Set C of distinguishing cubes over $\mathcal{U}^-$
1: $\varphi \leftarrow$ WeakestPre(assert$(P_1(x) \neq P_2(x))$
2: $D \leftarrow$ DNF(φ); $C \leftarrow \varnothing$
3: **for all** clause $d \in D$ **do**
4: $\quad C \leftarrow C \cup$ FindImplyingCubes$(d, \mathcal{U}^-)$
5: **return** C

Algorithm 3 RefinePredicates$(\mathcal{H}, \mathcal{U}^-)$

Require: Hypothesis space $\mathcal{H}$; universe $\mathcal{U}^-$
Ensure: Expanded universe $\mathcal{U}^-$
1: **for all** distinct pairs $(P_1, P_2) \in \mathcal{H} \times \mathcal{H}$ **do**
2: $\quad \varphi \leftarrow$ WeakestPre(assert$(P_1(x) \neq P_2(x))$
3: $\quad$ **for all** $\alpha \in$ Atoms(φ) **do**
4: $\quad\quad$ **if** Admissible(α) **then** $\mathcal{U}^- \leftarrow \mathcal{U}^- \cup \{\alpha\}$
5: **return** $\mathcal{U}^-$

Fig. 4. Auxiliary procedures for finding distinguishing predicates and refining predicate universe. The Admissible procedure subjects each atom to an application-specific admissibility test and may exclude predicates based on syntactic or semantic complexity.

different behaviors (line 4). Concretely, it computes a set of *distinguishing predicates* $\Phi(i, j)$ over $\mathcal{U}^-$, where each $\varphi \in \Phi(i, j)$ is a sufficient condition for the outputs of P_i and P_j to differ – that is, $\forall x.\ \varphi(x) \Rightarrow P_i(x) \neq P_j(x)$. This ensures that φ captures an entire region where the two programs are semantically incompatible, so any input satisfying φ witnesses their disagreement. To obtain these predicates, the algorithm first derives the weakest precondition under which P_i and P_j differ, then extracts its prime implicants [45] over $\mathcal{U}^-$. This subroutine is summarized in Algorithm 2.

Next, Algorithm 1 invokes GetBestPrecondition to compute a symbolic precondition ϕ that maximizes the objective in Eq. (1) from Definition 3.4. If no predicate in $\mathcal{U}^-$ satisfies this objective, GetBestPrecondition returns $\perp$, triggering RefinePredicates (Algorithm 3) to extend $\mathcal{U}^-$ with new atoms and restart the iteration (line 6). Once a valid precondition ϕ is obtained, the algorithm calls GenerateQuery to synthesize postconditions that jointly optimize Eq. (2) in Definition 3.4. The resulting query $Q = (\phi, \psi_1, \ldots, \psi_k)$ is then translated into natural language using an LLM and presented to the user (line 8). If the user selects answer j, their intended program P must satisfy the Hoare triple $\{\phi\}P\{\psi_j\}$, and all programs violating this triple are removed from $\mathcal{H}$ (line 9). The process repeats until all remaining programs in $\mathcal{H}$ are semantically equivalent (line 2). The following subsections describe precondition synthesis and query generation in more detail.

Algorithm 1 satisfies two key guarantees. The first says that the query selected at each step solves the Decomposed Query Selection Problem (Definition 3.4); the second says that, assuming the user responds correctly, the algorithm converges to the correct program.[1]

Theorem 4.1. *(Optimality of Query Selection). At each iteration, the query $(\phi, \psi_1, ..., \psi_k)$ generated by Algorithm 1 solves the Decomposed Query Selection Problem in Definition 3.4.*

Theorem 4.2. *(Correctness of Active Learning). Given a finite hypothesis space $\mathcal{H}$ containing the ground-truth program, Algorithm 1 always terminates and returns a program $P^* \in \mathcal{H}$ that is semantically equivalent to the user's intended program (under the assumption that the user provides correct answers to each query).*

4.1 Precondition Synthesis

We now explain the GetBestPrecondition procedure for finding a single predicate ϕ that optimizes the first objective (Eq. 1) in Definition 3.4. Recall that our goal is to find a conjunction of predicates over $\mathcal{U}^-$ that will differentiate as many program pairs as possible, but with a complexity penalty. To find such a precondition, our method utilizes the distinguishing predicates Φ precomputed

[1]Proofs of all theorems are provided in the Appendix of the extended version of the paper [8].

Given

$\mathcal{U}^- = \{A_1, \ldots, A_n\}$ and $\Phi(i,j) = \langle \Phi_{ij}^1, \ldots, \Phi_{ij}^{K_{ij}} \rangle$ is an ordered list of distinguishing predicates for program pair P_i, P_j.

Decision Variables

- *Atom selectors* $a_t \in 0, 1$ for $t \in [n]$, where each $a_t = 1$ indicates that atom A_t is selected.
- *Predicate selectors* $d_{ij}^k \in \{0,1\}$ for $i < j$ and $k \in [K_{ij}]$, where $d_{ij}^k = 1$ indicates that predicate Φ_{ij}^k is selected for pair (P_i, P_j).
- *Distinguished-pair flags* $p_{ij} \in \{0,1\}$ for $i < j$, where $p_{ij} = 1$ indicates that pair (P_i, P_j) is distinguished.

Hard Constraints

$$\textbf{(DP)} \quad p_{ij} \Longleftrightarrow \bigvee_{k=1}^{K_{ij}} d_{ij}^k, \qquad \textbf{(IA)} \quad d_{ij}^k \Longrightarrow \bigwedge_{t \in \text{atoms}(\Phi_{ij}^k)} a_t \qquad \textbf{(SAT)} \quad a_t \Rightarrow A_t$$

Objective

$$\max \sum_{i<j} p_{ij} \; - \; \lambda_{\text{pre}} \sum_{t=1}^{n} a_t$$

Synthesized Precondition

$$\phi(x) = \bigwedge_{t:a_t=1} A_t(x), \tag{3}$$

Fig. 5. Optimization modulo theory formulation of GetBestPrecondition

for each program pair in Algorithm 1. Specifically, rather than performing a blind search over all cubes in $\mathcal{U}^-$, we leverage Φ to restrict attention to conjunctions that are known to be useful. Each distinguishing predicate in $\Phi(i,j)$ guarantees that P_i and P_j behave differently on this input region and thereby specifies exactly which atoms of $\mathcal{U}^-$ must be included for the precondition to distinguish this pair. Because each predicate in $\Phi(i,j)$ is a sufficient condition for P_i and P_j to differ, the semantic reasoning is already handled in Algorithm 1. The remaining task, addressed in GetBestPrecondition, is to find a consistent subset of these witnesses and combine their atoms into a single global precondition.

We encode GetBestPrecondition as an Optimization Modulo Theory (OMT) problem in Figure 5. Our formulation introduces three families of variables: atom selectors a_t, which determine which atomic predicates from $\mathcal{U}^-$ are included in the final conjunction; predicate selectors d_{ij}^k, which track which distinguishing predicate is used as a witness for program pair (P_i, P_j); and pair flags p_{ij}, which indicate whether the pair (P_i, P_j) is successfully distinguished. Constraints (DP)–(SAT) tie these variables together. First, (DP) ensures that a pair is marked as distinguished iff at least one distinguishing predicate for that pair is selected. Next, (IA) enforces that if a predicate Φ_{ij}^k is chosen (meaning d_{ij}^k is assigned to true), then all atoms that occur in it are also selected. Finally, (SAT) ties the atom-selection variables to their corresponding predicates by asserting $a_t \Rightarrow A_t$ for every atom $A_t \in \mathcal{U}^-$. These implications define the synthesized precondition $\phi(x) = \bigwedge_{t:a_t=1} A_t(x)$, with the hard constraints collectively ensuring that ϕ is always satisfiable.

The objective function (shown under Objective in Figure 5) trades off disambiguation power and complexity: the first term (sum over all p_{ij}'s) rewards maximizing coverage by distinguishing as many pairs as possible, while the second term (sum over a_t's) penalizes the number of atoms selected to bias the solution toward simpler and more interpretable preconditions. After solving this OMT instance, we obtain the precondition ϕ by conjoining exactly those atoms A_t whose corresponding indicator a_t is true.

Theorem 4.3 (Correctness of GetBestPrecondition). *GetBestPrecondition solves Eq. (1) in Definition 3.4; i.e., letting $\phi(x)$ be as in Eq. (3) and $\mathbb{C}(\psi)$ be the number of conjuncts in ψ, then*

$$\phi \in \operatorname{argmax}_{\psi \in \mathit{Cube}(\mathcal{U}^-)} \left[\mathrm{DP}_{\mathrm{pre}}(\psi) - \lambda_{\mathrm{pre}} \cdot \mathbb{C}(\psi)\right].$$

Example 4.4. Consider the following hypothesis space of ImageEye programs:

P_1 := `Is(Face) ∧ HasAttribute(HairColor=brown) → Brighten`

P_2 := `Is(Face) ∧ Smiling → Brighten`

P_3 := `Find(Is(Face), Is(Guitar), Above) → Brighten`

Here, P_1 brightens all faces with brown hair, P_2 brightens all smiling faces, and P_3 brightens all faces that are above guitars. Assuming the input image contains two objects $\{x_1, x_2\}$, suppose that GetDistinguishing generates the following constraints for each pair of programs:

$\Phi_{1,2}$ = {`HasLabel(`x_1`, face) ∧ HasHairColor(`x_1`, brown) ∧ ¬HasExpression(`x_1`, smiling)`,
`HasLabel(`x_1`, face) ∧ HasHairColor(`x_1`, blonde) ∧ HasExpression(`x_1`, smiling)`, }

$\Phi_{1,3}$ = {`HasLabel(`x_1`, face) ∧ HasHairColor(`x_1`, brown) ∧ ¬HasLabel(`x_2`, guitar)`,
`HasLabel(`x_1`, face) ∧ HasHairColor(`x_1`, blonde) ∧ HasLabel(`x_2`, guitar) ∧ Above(`x_1`, `x_2`)`}

$\Phi_{2,3}$ = {`HasLabel(`x_1`, face) ∧ HasExpression(`x_1`, smiling) ∧ ¬HasLabel(`x_2`, guitar)`,
`HasLabel(`x_1`, face) ∧ ¬HasExpression(`x_1`, smiling) ∧ HasLabel(`x_2`, guitar) ∧ Above(`x_1`, `x_2`)`}

Here, choosing the red or the blue constraints maximizes distinguished program pairs. Since the red constraints contain fewer unique atoms, they will be selected to generate the precondition:

`HasLabel(`x_1`, face) ∧ HasHairColor(`x_1`, brown) ∧`
`¬HasExpression(`x_1`, smiling) ∧ ¬HasLabel(`x_2`, guitar)`

This precondition means that the image contains two objects: the first is a face with brown hair that is not smiling, and the second is an object that is not a guitar.

4.2 Overview of Multiple-Choice Answer Generation

Now that we have computed the optimal precondition ϕ, we need to generate optimal answer choices for ϕ. At a high level, the precondition ϕ identifies the region of the input space on which the programs in $\mathcal{H}$ exhibit qualitatively different behaviors, and the answer choices should group these behaviors into a small set of mutually exclusive, collectively exhaustive scenarios that the user can reliably distinguish. Algorithm 4 summarizes how we generate these answer choices.

To start with, Algorithm 4 calls GroupBySP to partition the programs in $\mathcal{H}$ based on their output behavior under ϕ (line 2), producing equivalence classes

$$C_i = \{P \in \mathcal{H} \mid \mathrm{StrongestPost}(P, \phi) = \gamma_i\} \qquad (\forall i \in \{1, ..., N\}).$$

Here, StrongestPost denotes the strongest postcondition; thus, each C_i consists of all programs in $\mathcal{H}$ whose output behaviors are indistinguishable under ϕ. In principle, we could now directly translate each equivalence class C_i into an answer choice based on γ_i. However, this approach is undesirable for two reasons: (1) the number of distinct clusters N may be too large, resulting in an

Algorithm 4 GenerateQuery

Require: Precondition ϕ, hypothesis space $\mathcal{H}$, postcondition universe $\mathcal{U}^+$
Ensure: Query $(\phi, \psi_1, \ldots, \psi_k)$

```
1: while true do
2:     {C_1, ..., C_N} ← GroupBySP(H, φ)
3:     (B_1, ..., B_k) ← MergeClusters({C_i}, k, φ)
4:     for i = 1, ..., k do
5:         ψ_i ← ConstructSeparator(B_i, {B_j}_{j≠i}, U^+)
6:     if ∃ i ∈ {1, ..., k} with ψ_i = ⊥ then
7:         U^+ ← U^+ ∪ RefinePostPredicates(H, φ, U^+)
8:     else return (φ, ψ_1, ..., ψ_k)

9: procedure RefinePostPredicates(H, φ, U^+)
10:    return ⋃_{P∈H} { a ∈ Atoms(StrongestPost(P, φ)) | Admissible(a) }
```

impractically long list of answer choices, and (2) the postconditions γ_i are typically more complex than needed.

To address these challenges, Algorithm 4 then invokes MergeClusters to coarsen the initial fine-grained partition $\{C_1, \ldots, C_N\}$ into at most k disjoint bins $B_1, \ldots, B_k$, where k is a small constant (typically 3 or 4). Each bin B_i corresponds to answer choice i in the final multiple-choice query, such that selecting option i retains only the programs in B_i and eliminates all others. MergeClusters seeks to produce bins of roughly equal size so that each answer removes a comparable portion of the hypothesis space. Next, Algorithm 4 calls ConstructSeparator (line 5) to convert each bin B_i into a symbolic postcondition ψ_i over the predicate universe $\mathcal{U}^+$, ensuring that ψ_i is consistent with the behaviors in B_i and excludes all programs in other bins. If no such postconditions can be synthesized (meaning ConstructSeparator returns $\perp$ for at least one cluster), the algorithm calls RefinePostPredicates (defined in lines 9–10) to extend $\mathcal{U}^+$ with new atomic predicates derived from the strongest postcondition.

We next describe the MergeClusters and ConstructSeparator procedures in more detail. We start with ConstructSeparator because it is internally used by MergeClusters to evaluate the cost of candidate merges.

4.3 Separator Construction

The goal of ConstructSeparator (summarized in Algorithm 5) is to synthesize a postcondition that captures the behavior of programs in the target bin while excluding all others. Specifically, it takes as input the target bin B, the negative bins $\{B_1, \ldots, B_k\}$, and the atom universe $\mathcal{U}^+$, and aims to find a postcondition ψ that (1) holds for all behaviors represented by B, (2) rules out behaviors from every other bin B_i, (3) is a cube $\psi \in \mathsf{Cubes}(\mathcal{U}^+)$, and (4) contains as few atoms as possible. The first two conditions ensure that ψ is correct (i.e., it separates B from the other bins), and the last two conditions ensure that it is interpretable (i.e., the answer choices are understandable).

Algorithm 5 starts by constructing the *positive specification* $\Phi^+ = \bigvee_{P \in B} \mathsf{StrongestPost}(P, \phi)$ for the target bin B and, a *negative specification* $\varphi_j = \bigvee_{P \in B_j} \mathsf{StrongestPost}(P, \phi)$ for every other B_j. If $\Phi^+ \wedge \varphi_j$ is satisfiable for any j, then the behaviors of B and B_j overlap on at least one input. In this case, no separator ψ can distinguish B and B_j because ψ needs to hold for *all* B behaviors but reject *all* B_j behaviors. Thus, in this case, the procedure terminates with $\perp$ to indicate failure.

Algorithm 5 ConstructSeparator$(B, \{B_1, \ldots, B_k\}, \mathcal{U}^+)$

1: $\Phi^+ \leftarrow \bigvee_{P \in B} \mathsf{StrongestPost}(P, \phi)$ ▷ Positive spec for target bin
2: $\varphi_j \leftarrow \bigvee_{P \in B_j} \mathsf{StrongestPost}(P, \phi)$ ▷ Negative specs for each B_j
3: **if** $\exists j.\ \mathrm{SAT}(\Phi^+ \wedge \varphi_j)$ **then return** $\bot$
4: $\mathcal{A} \leftarrow \{\, a \in \mathcal{U}^+ \mid \mathrm{UNSAT}(\Phi^+ \wedge \neg a) \,\}$ ▷ All atoms implied by Φ^+
5: $S \leftarrow \{s_a \mid a \in \mathcal{A}\}$ ▷ Indicators s_a denoting that atom a is chosen
6: $C \leftarrow \emptyset$ ▷ Counterexamples
7: **while** true **do**
8: $\quad \psi \leftarrow \mathrm{MaxSAT}\big(\{\, \neg s_a \mid a \in \mathcal{A} \,\}, C\big)$ ▷ Returns $\psi \triangleq \bigwedge_{s_a=1} a$ with as few atoms as possible
9: $\quad$ **if** $\psi = \bot$ **then return** $\bot$
10: $\quad$ **if** $\forall j.\ \mathrm{UNSAT}(\varphi_j \wedge \psi)$ **then return** ψ
11: $\quad$ Choose j, m with $m \models (\varphi_j \wedge \psi)$
12: $\quad C \leftarrow C \cup \left\{ \bigvee_{a \in \mathcal{A},\ m \models \neg a} s_a \right\}$

Otherwise, the algorithm proceeds to construct a separator ψ that satisfies the four conditions mentioned earlier. Here, satisfying conditions (1) and (2) is straightforward: since the check on line 3 guarantees that $\Phi^+ \wedge (\bigvee_j \varphi_j)$ is unsatisfiable, the entailment $\Phi^+ \Rightarrow \neg(\bigvee_j \varphi_j)$ holds, and we can take ψ to be a Craig interpolant for this entailment – that is, $\Phi^+ \Rightarrow \psi$ and $\psi \Rightarrow \neg(\bigvee_j \varphi_j)$. Then, ψ satisfies (1) since $\Phi^+ \Rightarrow \psi$ by definition, and (2) since $\psi \Rightarrow (\bigvee_j \varphi_j)$ implies that $\psi \wedge \varphi_j$ is unsatisfiable for every j. In other words, any Craig interpolant satisfies the first two conditions, but we must ensure that the interpolant is also a cube and contains as few atoms as possible to satisfy conditions (3) and (4).

Hence, rather than using an off-the-shelf interpolation tool, our method searches for a *minimum cube interpolant* using a custom algorithm based on counterexample-guided inductive synthesis (CEGIS). Specifically, it treats each atom in $\mathcal{U}^+$ as a candidate building block, and incrementally constructs the interpolant by alternating between an *optimization* phase and a *verification* step. Given a set of counterexamples C, the algorithm first attempts to solve an optimization problem subject to C and then checks whether the resulting solution is indeed a valid interpolant. If not, it strengthens the specification C and solves a more constrained optimization problem. Because each step preserves optimality subject to an over-approximation of the true specification, the first solution found is guaranteed to be the optimal interpolant.

Query: Suppose you have an input table with 2 rows and 2 columns. If the value in cell $(1, 1)$ is -1 and the value in cell $(2, 1)$ is 0, which of the following is true of your output table?:

(a) The table has 2 rows
(b) The table has 1 row, and the value in cell $(1, 1)$ is 0.
(c) The table has 1 row, and the value in cell $(1, 1)$ is -1.

Fig. 6. Generated query.

In more detail, line 3 of Algorithm 5 first restricts the search space to atoms $\mathcal{A} = \{\, a \in \mathcal{U}^+ \mid \mathrm{UNSAT}(\Phi^+ \wedge \neg a) \,\}$ that are already entailed by the positive specification. This is valid because any cube interpolant must consist only of atoms implied by Φ^+. Then the CEGIS loop (lines 7–12) alternates between using MaxSAT to compute the optimal solution consistent with C (line 8) and searching for counterexamples (line 10). The MaxSAT problem is over the Boolean variables s_a defined on line 5, where s_a indicates whether atom $a \in \mathcal{A}$ is in ψ (i.e., ψ is defined as $\bigwedge_{s_a=1} a$). Then, the optimization problem is to minimize the number of atoms in ψ subject to C (line 6).

If the computed ψ successfully separates the positive and negative specifications (i.e., $\psi \wedge \varphi_j$ is unsatisfiable for all j), then it is returned as the solution (line 10). Otherwise, the solver produces a model $m \models (\varphi_j \wedge \psi)$ for some j, from which a new blocking clause $\bigvee_{a \in \mathcal{A},\, m \models \neg a} s_a$ is derived. This clause enforces that any solution must include at least one atom that contradicts the counterexample m, thereby eliminating ψ and other cubes that fail for the same reason. Finally, this clause is added to the set of constraints (line 12) and the CEGIS loop continues.

Theorem 4.5 (Correctness of ConstructSeparator). ConstructSeparator *returns the smallest cube* $\psi \in \mathsf{Cubes}(\mathcal{U}^+)$ *such that* $\Phi^+ \Rightarrow \psi$ *and* $\forall j.\, \mathrm{UNSAT}(\psi \wedge \varphi_j)$, *or* $\bot$ *if no such cube exists.*

Example 4.6. Consider the following hypothesis space of three R programs:

P_1 := `Mutate(sum := Col(1) + Col(2))`

P_2 := `Filter(Col(1) < 0) |> Mutate(sum := Col(1) + Col(2))`

P_3 := `Filter(Col(1) = 0) |> Mutate(sum := Col(1) + Col(2))`

Given an input table, P_1 computes a new column containing the sum of columns 1 and 2. P_2 (resp. P_3) performs the same mutation, but first removes all rows where the value in column 1 is less than 0 (resp. equal to 0). Since there are only three programs, each program is placed in its own bin B_i. Under precondition $\phi := \mathtt{cell}_{1,1} = -1 \wedge \mathtt{cell}_{2,1} = 0$ (where $\mathtt{cell}_{i,j}$ corresponds to the value in the ith row and jth column), and assuming that the input table contains 2 rows and 2 columns, the strongest postconditions are as follows:

$$\mathsf{StrongestPost}(P_1, \phi) = \mathtt{num\_rows} = 2 \wedge \mathtt{num\_columns} = 3 \wedge \mathtt{output\_cell}_{1,1} = \mathtt{cell}_{1,1} \wedge \mathtt{output\_cell}_{1,2} = \mathtt{cell}_{1,2} \wedge \cdots$$
$$\mathsf{StrongestPost}(P_2, \phi) = \mathtt{num\_rows} = 1 \wedge \mathtt{num\_columns} = 3 \wedge \mathtt{output\_cell}_{1,1} = \mathtt{cell}_{2,1} \wedge \mathtt{output\_cell}_{1,2} = \mathtt{cell}_{2,2} \wedge \cdots$$
$$\mathsf{StrongestPost}(P_3, \phi) = \mathtt{num\_rows} = 1 \wedge \mathtt{num\_columns} = 3 \wedge \mathtt{output\_cell}_{1,1} = \mathtt{cell}_{1,1} \wedge \mathtt{output\_cell}_{1,2} = \mathtt{cell}_{1,2} \wedge \cdots$$

These postconditions constrain the shapes of the output tables and the values therein. Suppose our atom universe $\mathcal{U}^+$ contains all of the atoms in the postconditions. To construct a separator for B_1, we must select interpolants that rule out B_2 and B_3. In this case, both B_2 and B_3 may be ruled out by the single atom `num_rows = 2`, since both P_2 and P_3 are guaranteed to filter a row. For B_2, we need to find the simplest constraint that rules out both B_1 and B_3, while capturing the behavior of B_2. Selecting `num_rows = 1` $\wedge$ $\mathtt{output\_cell}_{1,1} = \mathtt{cell}_{2,1}$ as the separator satisfies both constraints and is the simplest such predicate. Finally, for B_3, the simplest such predicate is `num_rows = 1` $\wedge$ $\mathtt{output\_cell}_{1,1} = \mathtt{cell}_{1,1}$. Thus, for this example, our method would pose the multiple-choice query shown in Figure 6.

4.4 Merging Clusters

We now discuss the MergeClusters procedure (Algorithm 6) for producing a partition of the hypothesis space into k clusters. This algorithm takes as input the initial *fine-grained partition* $C_1, \ldots, C_N$ (induced by the strongest postconditions γ_i) and produces *coarse-grained* partition $\{B_1, \ldots, B_k\}$ for a fixed k, which we represent by a *partition mapping* $\mathcal{F} : [N] \to [k]$, where $\mathcal{F}(i) = j$ indicates that cluster C_i is included in bin B_j. For a fixed precondition and hypothesis space, any partition mapping $\mathcal{F}$ naturally induces a multiple-choice query, defined as follows:

Definition 4.7 (Induced query). Given hypothesis space $\mathcal{H}$, precondition ϕ, and partition mapping $\mathcal{F}$, the *query induced by* $(\mathcal{F}, \phi^*)$ is $(\phi, \psi_1(\mathcal{F}), \ldots, \psi_k(\mathcal{F}))$, where

$$P_j(\mathcal{F}) = \{\, h \in \mathcal{H} \mid \exists\, i\, .\, \mathcal{F}(i) = j \ \wedge\ h \in C_i \,\}, \qquad \mathcal{N}_j(\mathcal{F}) = \left\{ N_r \;\middle|\; r \neq j,\ N_r = \bigcup\nolimits_{\mathcal{F}(i)=r} C_i \right\}$$

$$\psi_j(\mathcal{F}) := \textsc{ConstructSeparator}\big(P_j(\mathcal{F}),\, \mathcal{N}_j(\mathcal{F}),\, \mathcal{U}^+\big).$$

Algorithm 6 MergeClusters

Require: Clusters $C_1, \ldots, C_N$, number of answers k, precondition ϕ
Ensure: EvaluateObjective$(\mathcal{F}, \phi) \geq O_{\min}$, or $\mathcal{F}$ is optimal

function MergeClusters($\{C_i\}_{i=1}^N$, k, ϕ)
  $\mathcal{F} \leftarrow$ LbPartition($\{C_i\}_{i=1}^N$, k)
  $O_{\text{init}} \leftarrow$ EvaluateObjective$(\mathcal{F}, \phi)$
  $(\mathcal{F}, O_{\text{best}}) \leftarrow$ BranchAndBound($\{C_i\}_{i=1}^N$, $\varnothing$, O_{init}, ϕ)
  return $\mathcal{F}$
function LbPartition(($\{C_i\}_{i=1}^N$, k))
  for $i = 1$ to N **do**
    $\psi_i \leftarrow$ ConstructSeparator$(C_i, \{\cup_{j \neq i} C_j\}, \mathcal{U}^+)$
    $w_i \leftarrow |C_i| + \lambda\, \mathbb{C}(\psi_i)$
  return LB-BinPack-Solver($\{w_i\}_{i=1}^N$, k)

Intuitively, each bin B_j is represented by a separator ψ_j that distinguishes it from all others. Our goal is to generate an informative and interpretable query, which corresponds to generating a partition mapping that results in balanced clusters and interpretable separators, respectively.

The key challenge is that evaluating the quality of a partition requires computing its separators, but this makes the optimization problem computationally intractable: the space of possible partitions is exponential in N, and each candidate partition requires invoking ConstructSeparator, which is already solving an NP-hard problem. We address this challenge using a two-phase approach. In the first phase (LbPartition), we optimize a proxy objective that approximates the true objective, resulting in a *proxy partition* that is far more efficient to compute since it does not require explicitly constructing separators. Then, the second phase performs a branch-and-bound search starting from the proxy partition. Intuitively, since the proxy partition is a high-quality starting point, the branch-and-bound procedure can efficiently prune the search space.

The first phase is based on the insight that the quality of a partition can be estimated without computing all pairwise separators. Instead, we approximate the true objective by constructing, for each initial fine-grained cluster, a *one-vs-all* separator that distinguishes it from the union of all other clusters. This strategy provides a good signal about how easily each cluster can be separated from the rest, allowing us to *estimate* the cost of potential merges. Specifically, we formulate a proxy objective that takes into account both (1) how balanced a partition is, and (2) its estimated separator complexity cost, obtained by aggregating the one-vs-all separator costs.

In more detail, we formulate this proxy optimization problem as a *load-balanced bin-packing* task, where the goal is to distribute items of varying weights into a fixed number of bins so that no bin becomes disproportionately heavy. In our setting, each fine-grained cluster C_i plays the role of an item, and each answer option corresponds to a bin. The "weight" of each item reflects both the number of programs it contains and the complexity of its separator:

$$w_i \;=\; |C_i| + \lambda\, \mathbb{C}(\psi_i), \quad \text{where} \quad \psi_i = \textsc{ConstructSeparator}(C_i, \{\cup_{j \neq i} C_j\}, \mathcal{U}^+)$$

Here, $|C_i|$ penalizes clusters that contain many candidates and $\mathrm{C}(\psi_i)$ is the syntactic complexity of the one-vs-all separator distinguishing C_i from all other clusters. The optimization seeks a mapping of clusters to k bins that minimizes the load of the heaviest bin, thereby achieving two desirable properties: (i) no single answer option dominates the hypothesis space, and (ii) each option corresponds to clusters that can be separated using simple conditions. We let $\mathcal{F}$ denote the proxy partition mapping constructed by LbPartition.

In the second phase, our algorithm performs branch-and-bound search over the space of partition mappings with $\mathcal{F}$ as the starting point. First, the EvaluateObjective procedure in line 3 computes the induced query for $\mathcal{F}$, and then computes its true objective value O_{init} according to Definition 3.4. The branch-and-bound implementation is standard. It organizes the search space as a tree, where each internal node corresponds to a partial assignment of fine-grained clusters to bins, and each leaf represents a complete partition. It uses an admissible heuristic on the best achievable objective; for efficiency, it does not need to call ConstructSeparator. We provide details the Appendix of the extended version of the paper [8].

5 Implementation

We have implemented the proposed active learning technique as a new tool called Socrates, written in Python. Socrates uses the Z3 SMT solver [19] for checking satisfiability and solving optimization problems.

Translating queries to natural language. Our implementation leverages `gpt-4o` to translate logical queries into natural language (NL) descriptions. We utilize few-shot prompting [9], providing the LLM with a query along with a small set of in-context examples. Because this translation is not formally verified, a mistranslation could in principle cause the user to select an incorrect answer. In practice, however, we find this risk to be negligible because the queries are expressed in first-order logic over a small, well-typed predicate set – this is a setting where LLMs are highly reliable [6].

Instantiating in new domains. The design of Socrates is domain-agnostic and can be instantiated for different synthesis settings by providing three components: (1) a synthesizer for generating the initial hypothesis space, (2) an analysis engine for computing pre- and postconditions, and (3) the initial universes of pre- and postcondition predicates. Depending on the domain, the analysis engine may either apply standard invariant-generation techniques to reason about iterative or higher-order constructs (e.g., `fold`), or unroll these constructs to a fixed bound and compute pre- and postconditions on the resulting loop- and recursion-free programs. In the latter case, Socrates guarantees semantic equivalence only up to the chosen unrolling depth. For the domains used in our evaluation, we adopt this bounded-unrolling strategy and manually verify the correctness of the final synthesized program. The construction of the predicate universes is straightforward. Precondition predicates in $\mathcal{U}^-$ follow the shape of predicates already exposed by the DSL, while postcondition predicates in $\mathcal{U}^+$ encode possible effects of the program on the input. For example, in the image editing domain, $\mathcal{U}^-$ includes predicates such as `HasLabel(obj, Person)` and `HasRelation(obj_1, obj_2, NextTo)`, whereas $\mathcal{U}^+$ includes predicates such as `Blurred(obj)` and `Cropped(obj)` that describe observable output behavior. More generally, $\mathcal{U}^-$ captures properties of inputs that may appear in synthesized preconditions, while $\mathcal{U}^+$ captures properties of outputs or input-output relationships that may appear in answer choices.

Approximating the objective. While the algorithms described in Section 4 compute the optimal query as defined in Definition 3.4, our implementation employs two practical approximations to ensure tractability. First, following prior work [4, 34], we uniformly sample a subset of programs from the hypothesis space each round and compute queries that are optimal with respect to this subset, rather than over the full hypothesis space. Second, our implementation of MergeClusters invokes branch-and-bound search only when the solution produced by LbPartition exhibits complexity exceeding a threshold. These approximations preserve the intent of optimal query selection while keeping the computation efficient in practice.

6 Evaluation

In this section, we describe the results of our experimental evaluation, which aims to answer the following research questions:

- **RQ1: Accuracy.** How does Socrates compare against state-of-the-art active learning baselines in terms of accuracy?
- **RQ2: Interpretability.** How do users perform when answering the multiple-choice queries posed by Socrates compared to traditional input–output labeling questions?
- **RQ3: Efficiency.** How does Socrates compare against baselines in terms of efficiency (e.g., number of interaction rounds, query generation time)?
- **RQ4: Ablations.** How important are our key algorithmic ingredients in reducing active learning runtime and generating simple queries?

6.1 Application Domains

To address our research questions, we evaluate Socrates and all baselines on 157 tasks drawn from four domains studied in prior work: data wrangling [23], JSON transformations [15], batch image editing [5], and image search [6]. All of these domains involve rich input types, namely tables, trees, and images, and therefore provide a meaningful basis of evaluation for our approach. We describe each domain in more detail below.

Table transformations. Our first application domain, Wrangle, consists of 80 challenging *data wrangling* tasks considered in prior work [22, 23]. Each task involves transforming one or more input tables into a target table using a DSL that is inspired by R's `dplyr` and `tidyr` libraries. Typical transformations include reshaping data between "wide" and "long" formats, consolidating multiple tables, and performing grouped aggregations or joins.

JSON transformations. Our second application domain, Json, consists of 15 JSON transformation tasks from prior work [15]. Each benchmark involves converting hierarchical input trees into structurally distinct outputs using a domain-specific language that supports node creation, filtering, and restructuring through declarative tree combinators. These transformations capture a range of structural manipulations, such as field extraction, flattening, and conditional reorganization.

Batch image editing. Our third application domain, ImageEdit, involves image editing tasks considered in prior work [4, 5]. Unlike the two previous domains, these tasks are neurosymbolic and require synthesizing programs in a DSL that involves neural networks for image classification and segmentation. Each task specifies a high-level edit (e.g., "brighten faces of bride and groom" "crop all people playing guitar") that must be realized by composing various neural components with symbolic operators. These benchmarks come with collections of real-world images.

Image search. Our fourth and final application domain, ImageSearch, involves image search tasks from prior work [4, 6] where the goal is to filter a subset of images that have a certain property, such as "contain a dog and a cat next to each other," or "contain a person riding a bike while wearing a helmet." Similar to the previous domain, these tasks require synthesis in a neurosymbolic DSL that has neural constructs for image segmentation and object classification.

6.2 Baselines

To meaningfully evaluate our proposed approach, we compare Socrates against state-of-the-art active learning techniques from recent work. Because no single prior method applies uniformly across both symbolic and neurosymbolic domains, our evaluation considers two groups of baselines.

Purely symbolic domains. For purely symbolic domains (Wrangle and Json), we compare against **SampleSy** [34] and **LearnSy** [33]. SampleSy selects the input whose *worst-case* label would eliminate the largest fraction of the hypothesis space, while LearnSy generalizes this strategy by introducing a probabilistic model of program equivalence that estimates the *expected* information gain from labeling a particular input.

Neurosymbolic domains. For neurosymbolic domains (ImageEdit and ImageSearch), we compare Socrates against **SmartLabel** [4], which is the only prior technique that provides formal guarantees against eliminating the intended program in the presence of neural uncertainty. SmartLabel extends SampleSy's objective to neurosymbolic settings by incorporating conformal prediction [1], producing set-valued outputs that bound the true label with high confidence.

6.3 Experimental Setup and Methodology

We evaluate Socrates, along with all baselines and ablations, using the following methodology. We begin by generating two initial input–output examples and constructing a hypothesis space $\mathcal{H}$ of programs consistent with the examples, using either an enumerative synthesizer or an LLM-based agent. The hypothesis space always includes the ground-truth program P^*; however, it also includes many other programs that conform to the initial examples, but semantically differ from the ground truth. Once such a hypothesis space $\mathcal{H}$ is constructed, we perform active learning over $\mathcal{H}$, using an oracle that provides correct responses to each query. To account for potential variability from the initial random I/O examples, we repeat all experiments five times, each with a different seed, and report the mean and standard deviation of the outcomes.

Table 1 summarizes key statistics for our benchmark domains, including the number of tasks, average program size, and average cardinality of the initial hypothesis space. While Socrates does not require access to a predefined set of inputs on which the target program is evaluated, all of the baseline methods do. In particular, these baselines are parameterized by a finite *input space* from which candidate queries are drawn. Thus, Table 1 also shows the size of the input space used for evaluating the baselines. Specifically, for the ImageEdit and ImageSearch domains, we adopt the same input spaces used in prior work [4]. In contrast, for the Wrangle and Json domains, no standard input dataset is available; we construct synthetic input spaces by sampling schema-compatible inputs that satisfy the grammar of each domain.[2]

Table 1. Details about (1) the number of tasks, (2) average input space size, (3) initial hypothesis space size, (4) average program size, and (5) the percentage of program pairs in the initial hypothesis space that are semantically distinguishable.

Domain	#	|Inputs|	$\|\mathcal{H}\|$	AST size	% Pairs Dist.
Wrangle	80	75.0	50.0	14.6	58.1%
Json	15	375.0	95.6	11.7	86.4%
ImageEdit	37	271.7	1096.4	15.7	87.3%
ImageSearch	25	257.4	318.8	16.0	92.9%
Overall	157	244.8	390.2	14.5	73.2%

To further characterize the complexity of query generation, Table 1 also reports the fraction of program pairs in the initial hypothesis space that are semantically distinguishable. This metric captures how easily the hypothesis space can be pruned by informative queries: when many pairs are distinguishable, the system can more readily construct queries that expose behavioral differences among candidates, whereas greater semantic overlap means that many candidates

[2]To make the comparison fair and meaningful, we must choose the size of the input space carefully: including too few inputs hurts accuracy, while including too many increases the time needed to compute each query. Following findings from the HCI literature on user tolerance for interactive wait times [40], we select the largest input space for which all methods complete query generation within 10 seconds, which has been reported as the upper bound of acceptable response latency for sustained user engagement.

Table 2. Experimental results comparing accuracy of active learning techniques in symbolic domains.

Domain	Socrates	SampleSy	LearnSy
Wrangle	**100.0% ± 0%**	80.3% ± 4.3%	78.5% ± 5.4%
Json	**100.0% ± 0%**	62.7% ± 7.6%	64.0% ± 6.0%
Overall	**100.0% ± 0%**	77.5% ± 2.5%	76.2% ± 3.8%

Table 3. Experimental results comparing accuracy of active learning techniques in neurosymbolic domains.

Domain	Socrates	SmartLabel
ImageEdit	**100% ± 0%**	84.9% ± 2.4%
ImageSearch	**100% ± 0%**	86.4% ± 8.3%
Overall	**100% ± 0%**	85.5% ± 2.3%

behave identically on large parts of the input space and are therefore harder to separate. Overall, 73.2% of program pairs are distinguishable, although this fraction varies across domains, with tasks in the Wrangle domain having the lowest distinguishability rate (58.1%).

All experiments are executed on a 2022 MacBook Pro with an 8-core M2 processor and 8 GB of RAM, using a timeout limit of 300 seconds per task.

6.4 Evaluation of Accuracy

Our first research question investigates whether Socrates improves the accuracy of interactive program synthesis compared to existing active learning techniques. For each method, we measure the percentage of benchmarks for which the synthesized program is *semantically equivalent* to the ground-truth program P^*. A benchmark counts as solved only if the synthesized program is manually verified to behave identically to P^* on all possible inputs. Tables 2 and 3 report these results for the symbolic (Wrangle, Json) and neurosymbolic (ImageEdit, ImageSearch) domains. Across all domains, Socrates solves 100% of benchmarks. For the Wrangle and Json domains, SampleSy and LearnSy solve 77.5% and 76.2% benchmarks, respectively, and for the neurosymbolic domains, SmartLabel solves 85.5% of the benchmarks.

Failure analysis for baselines. Most baseline failures stem from their reliance on a fixed input set for checking equivalence. These methods iteratively refine the hypothesis space until all remaining candidates are observationally indistinguishable on that set and then return one randomly sampled program. However, observational equivalence on a finite input set does not imply true semantic equivalence, and the selected program may therefore diverge from the ground truth. As discussed in Section 5, the equivalence guarantee provided by Socrates is also not universal, due to the bounded unrolling used to compute pre- and postconditions. Despite this practical limitation, all programs synthesized by Socrates were manually verified to be semantically equivalent to the ground truth. In contrast, the baselines frequently produce programs that match the ground truth on all observed examples but behave differently on unseen inputs – an inherent limitation of any active-learning approach restricted to a fixed input set. Finally, SmartLabel sometimes fails for a distinct reason: although its conformal predictor provides statistical coverage guarantees, there remains a small probability that the ground-truth label falls outside the predicted confidence set. In the neurosymbolic domains, a small fraction of failures can be attributed to this cause.

Result for RQ1: Socrates converges to the ground-truth program in all experimental benchmarks, while the baselines fail to find the intended program for roughly 25% of the programs in the symbolic domains and 15% in the neurosymbolic domains.

6.5 Evaluation of Interpretability

To answer our second research question, we conducted a user study to evaluate interpretability. This study measures how easily users can understand and answer multiple choice (MC) queries compared to the input–output (I/O) queries produced by existing active learning techniques.

Setup. We recruited 18 participants, each of whom completed two active learning tasks per application domain: one using Socrates and one using a representative active learning baseline (SampleSy for Wrangle and Json, and SmartLabel for ImageEdit and ImageSearch). Participants were undergraduate and graduate students in computer science, all with prior programming experience but no familiarity with the evaluated systems.

To keep the total session length under 45 minutes and reduce cognitive fatigue, each participant was assigned tasks from three of the four domains. Each task was randomly drawn from a pool of five benchmarks, and both the order of domains and the assignment of tools were randomized to mitigate ordering effects. For each task, participants were first shown a short textual description and an I/O example, followed by a two-minute familiarization period during which they could review the materials and ask clarifying questions. They then answered all queries issued by the corresponding active learning tool. We recorded both the time to answer each query and whether the answer matched the ground truth.

Results. Table 4 summarizes the results of our user study. Across all domains, users answered MC questions slightly faster and 38% more accurately than I/O questions. The overall accuracy improvement is statistically significant ($p = 4.35 \times 10^{-5}$, paired-sample t-test). Using the same test, response time differences were statistically significant only in the Wrangle domain ($p = 0.043$), where users answered MC questions 38 seconds faster on average. This difference likely reflects the additional overhead of manually typing tables for I/O questions, which was considerably more time-consuming than selecting a multiple-choice answer. In the Json domain, users answered MC questions 44% more accurately and 10 seconds faster than I/O questions, suggesting that the structured presentation of MC queries helped users reason about hierarchical data more effectively. In the ImageEdit and ImageSearch domains, MC questions yielded 52% and 60% higher accuracy, respectively, despite slightly longer response times. For these domains, I/O questions required users to inspect images and identify objects, leading to frequent misidentifications or omissions. In contrast, MC questions provided textual descriptions and a small set of candidate options, which seems to have reduced ambiguity and human error. The Appendix of the extended version of the paper [8] illustrates representative errors users make when answering queries.

Table 4. User study comparing the interpretability of multiple-choice (MC) and I/O queries.

Domain	Time (s)		Accuracy (%)	
	MC	I/O	MC	I/O
Wrangle	**60.5**	98.8	**93.8**	86.7
Json	**59.9**	70.5	**77.8**	54.1
ImageEdit	45.5	**38.6**	**88.2**	57.9
ImageSearch	25.6	**21.2**	**92.3**	57.8
Overall	**48.2**	55.9	**88.5**	64.1

Result for RQ2: Users answered multiple-choice queries significantly more accurately than the input–output queries, while maintaining comparable response times.

6.6 Evaluation of Efficiency

Having established that Socrates improves both synthesis and query accuracy, we next examine whether this improvement comes at the cost of efficiency. We measure efficiency along two dimensions: (1) the number of interaction rounds required to converge, and (2) the average time needed to generate each query. Table 5 summarizes these results.

Across all domains, Socrates is competitive in terms of efficiency. In the Wrangle domain, it converges in roughly the same number of rounds as LearnSy and fewer than SampleSy, while maintaining lower average query-generation time. In the neurosymbolic domains (ImageEdit and ImageSearch), Socrates matches or outperforms SmartLabel in runtime and requires roughly

the same number of interaction rounds despite its stronger semantic guarantees. The only setting where Socrates is less efficient is the Json domain. However, this is explained by the fact that the baselines have the lowest accuracy in this domain: while they take approximately 3 fewer rounds of user interaction, their accuracy is less than 65% (compared to 100% of Socrates).

Our runtime measurements indicate that Socrates's computational cost is dominated by LLM inference (using `gpt-4o`), which is used to translate logical queries into natural language. To isolate this effect, Table 5 also reports results for Socrates-InstantLLM, which omits LLM processing time. This configuration is consistently faster than all baselines across every domain. Although a few Wrangle benchmarks exhibit outliers that inflate the mean, the median query-generation time for Socrates-InstantLLM remains under two seconds. It is worth noting that the cost of LLM inference primarily reflects model throughput and network variability, and it can be substantially reduced through improved deployment infrastructure [58] (e.g., provisioned throughput or specialized inference accelerators) without any modification to Socrates's core technique.

Impact of different components on runtime. Another interesting empirical question is where Socrates spends the majority of its query-generation time in practice. As noted above, LLM inference for translating queries into natural language accounts for a significant portion of runtime (roughly 42%), but this cost is not a fundamental algorithmic bottleneck. Figure 7 shows how the remaining components contribute to query-generation time.

Across all domains, computing distinguishing predicates accounts for 70.6% of runtime, precondition synthesis accounts for 11.5%, and postcondition generation accounts for 13.0%. This breakdown varies by domain. In the Wrangle domain, computing distinguishing predicates dominates runtime (81%). In the ImageEdit and ImageSearch domains, precondition synthesis becomes the primary cost, accounting for 60–66% of runtime. In the Json domain, query generation time is distributed more evenly across the three components.

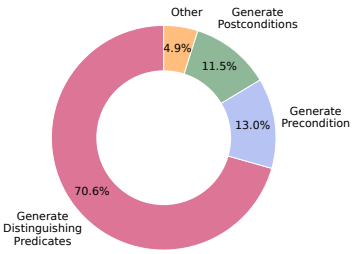


Fig. 7. Breakdown of active learning runtime, excluding LLM inference time.

This trend reflects the greater symbolic reasoning burden in the Wrangle domain, where programs compose multiple table transformations and exhibit the lowest semantic distinguishability (58.1%, Table 1). Furthermore, weakest precondition size is largest in this domain.

> **Result for RQ3:** Socrates achieves stronger semantic guarantees with only a marginal increase in rounds of user interaction and comparable or faster query computation time.

6.7 Ablation Studies

To quantify the contribution of each major algorithmic component in Socrates, we conduct ablation studies that remove or modify one component at a time while keeping others fixed. Specifically, we evaluate the following three variants:

- **Socrates-SimplePre**, which selects any distinguishing constraint between two non-equivalent programs as the precondition. This ablation isolates the impact of GetBestPrecondition.
- **Socrates-SimpleSep**, which computes a separator by taking the disjunction of the strongest postconditions within each cluster and then simplifying the resulting formula [20]. This ablation isolates the impact of ConstructSeparator.
- **Socrates-RandCluster**, which constructs roughly equal-sized clusters by assigning each program to a randomly chosen cluster. After forming the clusters, it constructs a multiple-choice

Table 5. Experimental results comparing Socrates against active learning baselines. For each technique, we report the averages of (1) number of interaction rounds, (2) time per round, and (3) total tool runtime.

Domain	Technique	# Rounds	Time per Round (s)	Total Tool Time (s)
Wrangle	Socrates	4.4 ± 0.1	7.8 ± 0.4	34.5 ± 2.3
	Socrates-InstantLLM	4.4 ± 0.1	5.7 ± 0.4	25.2 ± 1.8
	SampleSy	5.1 ± 0.3	9.6 ± 0.3	48.4 ± 2.0
	LearnSy	4.2 ± 0.1	8.5 ± 0.2	35.8 ± 1.9
Json	Socrates	5.6 ± 0.2	5.9 ± 0.5	33.4 ± 2.5
	Socrates-InstantLLM	5.6 ± 0.2	0.9 ± 0.0	5.1 ± 0.3
	SampleSy	2.6 ± 0.1	9.5 ± 0.8	25.2 ± 2.6
	LearnSy	3.4 ± 0.3	4.8 ± 0.2	16.3 ± 1.3
ImageEdit	Socrates	4.1 ± 0.6	3.1 ± 0.2	12.9 ± 2.3
	Socrates-InstantLLM	4.1 ± 0.6	0.9 ± 0.2	3.8 ± 0.9
	SmartLabel	3.7 ± 0.5	5.5 ± 0.6	20.6 ± 6.5
ImageSearch	Socrates	4.0 ± 0.6	2.9 ± 0.2	11.7 ± 2.9
	Socrates-InstantLLM	4.0 ± 0.6	0.7 ± 0.1	2.9 ± 0.7
	SmartLabel	4.5 ± 0.8	6.3 ± 2.2	27.4 ± 7.3

Table 6. Summary of ablation study results.

Variant	Key impact
Socrates-SimplePre	Increases the number of rounds by about 2×, while modestly improving query generation time and query complexity.
Socrates-SimpleSep	Nearly triples postcondition complexity, with minimal effect on other metrics.
Socrates-RandCluster	Almost doubles the number of rounds and increases postcondition complexity by more than 2.5×.

Table 7. Detailed ablation study results. The first row reports absolute values for Socrates. Remaining rows report percentage change relative to Socrates.

Variant	# Rounds	Time/Round (s)	Precond.	Postcond.
Socrates	4.4 ± 0.1	3.6 ± 0.3	21.4 ± 0.1	6.9 ± 0.0
Socrates-SimplePre	+102%↑	-61%↓	-33%↓	-39%↓
Socrates-SimpleSep	-2%	+11%	+0%	+191%↑
Socrates-RandCluster	+84%↑	+17%	+4%	+154%↑

question by calling ConstructSeparator, merging clusters as needed if a separator does not exist. This ablation isolates the impact of our MergeClusters procedure.

Tables 6 and 7 report the results of this ablation study. Specifically, Table 6 summarizes the key impact of disabling a given component, and Table 7 provides more detailed statistics about the averages of (1) the number of interaction rounds, (2) query generation time (excluding LLM processing time), (3) precondition complexity, and (4) postcondition complexity (measured by AST size). As we can see, disabling any component of Socrates degrades performance along at least one dimension relative to the full configuration. Socrates-SimplePre generates simpler queries quickly, but doubles the number of interaction rounds. This suggests that our OMT-based precondition

generation method plays an important role in reducing user effort. Socrates-SimpleSep leaves tool runtime essentially unchanged, but substantially increases postcondition complexity. This indicates that separator construction is important for keeping answer choices easy to understand. Finally, Socrates-RandCluster increases both the number of rounds and the complexity of the postconditions, showing that our clustering strategy helps form bins that are balanced and admit clean separators.

Result for RQ4: Our key algorithmic ingredients collectively improve the trade-off between query complexity, interaction rounds, and query-generation time.

7 Related Work

Active learning for program synthesis. Active learning refers to a class of techniques that strategically select data in order to maximize information gain. Originally developed in the context of machine learning [10, 11, 18, 48–50], active learning has recently been applied to *interactive program synthesis*, where it is used to disambiguate candidate programs. In this setting, the active learner typically queries the user to label an input from a pre-defined input space. For example, *FlashProg* [38] asks the user to label inputs on which candidate programs produce different outputs. Subsequent approaches [4, 12, 30, 33, 34] generalize this idea by formulating query selection as an optimization problem wherein the goal is to minimize the number of rounds of user interaction. For example, *SampleSy* and *SmartLabel* both employ greedy minimax algorithms, selecting the question whose worst answer will prune the greatest portion of programs. These methods iterate until all remaining programs agree on all queries in the input space. However, as demonstrated in Section 6.4, these methods may fail to identify the ground-truth program due to their restricted notion of equivalence.

An alternative line of work explores symbolic program disambiguation, where the active learner reasons over logical constraints rather than concrete inputs. In this setting, the system constructs a formula characterizing inputs on which candidate programs produce different outputs and invokes an SMT solver to find an input satisfying that formula. Several prior approaches [24, 32, 54] follow this pattern, using the resulting input as a new query for the user. Ramos et al. [47] extend this idea by formulating program disambiguation as a MaxSAT optimization problem that selects an input distinguishing the largest number of program pairs. However, their formulation assumes that program semantics can be fully captured in propositional logic. More broadly, while these kinds of input generation techniques work well for programs over simple data types like integers or strings, extending them to richer data (e.g., JSON documents or images) remains challenging: constructing a concrete input that satisfies a symbolic constraint is an open problem, especially when constraints involve complex structural or perceptual features. For example, generating an image that satisfies a logical predicate would require constraint-conditioned image synthesis, for which efficient and reliable methods do not yet exist [52].

User interaction models for program synthesis. Since providing input-output examples in the traditional programming-by-example (PBE) setting is often challenging [35, 39], prior work has proposed alternative interaction models that make it easier for users to specify their intent. Several approaches [42, 57] propose interfaces that visualize clusters of data in the input space, as an aid for selecting new input-output examples. Singh and Solar-Lezama [51] propose a method that allows users to specify their task using control-flow diagrams. Other interfaces [43, 44] present a candidate program, and let users annotate sub-expressions that should or should not be present in the solution. Drachsler-Cohen et al. [21] present an interactive synthesis system that communicates with the user through abstract examples – i.e., symbolic input–output pairs describing potentially

unbounded sets of examples. The user can either accept an abstract example or provide a concrete counterexample that invalidates it. While this design offers formal guarantees once the accepted abstractions cover the input space, it expects users to reason about symbolic patterns and to submit concrete instances that contradict them.

Natural language interaction in synthesis. In recent years, there has been much interest in leveraging large language models (LLMs) in synthesis systems. Many prior works allow users to specify their intent with natural language (NL) queries that are processed by an LLM [2, 13, 16, 29, 41, 53]. Since NL is inherently imprecise, some approaches combine NL specifications with other modalities, such as input-output examples [6, 14, 26, 31, 46] or demonstrations [36, 37]. While writing NL queries is often convenient, these approaches offer no guarantee that the synthesized code is correct across all inputs. In our work, we generate simple natural language queries that are designed to clarify ambiguities in the user's intent.

Predicate abstraction in program synthesis. Predicate abstraction is a classical technique for approximating program behavior within a finite domain of logical predicates [3, 25, 27]. In verification, frameworks such as CEGAR [17] iteratively refine these abstractions to rule out spurious counterexamples, and similar ideas have been applied to program synthesis [28, 55, 56]. In our setting, predicate abstraction serves a different role: it is used to synthesize structured, human-interpretable queries that can be formulated as multiple-choice questions.

8 Conclusion

This paper introduced a new paradigm for interactive program disambiguation based on multiple-choice queries, where users choose from a list of high-level behaviors instead of labeling concrete inputs. By formulating each question as a structured logical query over pre- and postconditions, our approach enables more direct communication of semantic intent and enables stronger correctness guarantees. We presented a principled algorithm that decomposes query selection into precondition synthesis and answer generation, realized through a combination of SMT-based optimization, clustering, and separator construction. Our implementation, Socrates, demonstrates that this paradigm leads to substantially higher synthesis accuracy and user response accuracy across both symbolic and neurosymbolic domains, while maintaining competitive efficiency. More broadly, this work highlights that replacing low-level annotation with structured semantic queries can make program disambiguation more practical both for users and synthesis systems. Looking ahead, as large language models are increasingly used for code generation, integrating structured disambiguation mechanisms like ours offers a promising direction for improving the reliability and interpretability of LLM-driven synthesis.

9 Data-Availability Statement

The artifact for this paper is available on Zenodo [7].

Acknowledgments

We would like to thank the members of the UTOPIA group, and the anonymous reviewers, for their help and feedback on this paper. This work was conducted in a research group supported by NSF awards CCF-1918889, CNS-2120696, CCF-2210831, CCF-2319471, CCF-2422130, CCF-2505865, CCF-2326576, and CCF-2403211, as well as a DARPA award under agreement HR00112590133 and a gift from Amazon.

References

[1] Anastasios N. Angelopoulos and Stephen Bates. 2023. Conformal Prediction: A Gentle Introduction. *Found. Trends Mach. Learn.* 16, 4 (March 2023), 494–591. doi:10.1561/2200000101

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 203–213. doi:10.1145/378795.378846

[4] Celeste Barnaby, Qiaochu Chen, Ramya Ramalingam, Osbert Bastani, and Isil Dillig. 2025. Active Learning for Neurosymbolic Program Synthesis. *arXiv preprint arXiv:2508.15750* (2025).

[5] Celeste Barnaby, Qiaochu Chen, Roopsha Samanta, and Işıl Dillig. 2023. ImageEye: Batch Image Processing Using Program Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 134 (jun 2023), 26 pages. doi:10.1145/3591248

[6] Celeste Barnaby, Qiaochu Chen, Chenglong Wang, and Isil Dillig. 2024. PhotoScout: Synthesis-Powered Multi-Modal Image Search. *arXiv preprint arXiv:2401.10464* (2024).

[7] Celeste Barnaby and Danny Ding. 2026. *Artifact for "Choose, Don't Label: Multiple-Choice Query Synthesis for Program Disambiguation"*. doi:10.5281/zenodo.19052770

[8] Celeste Barnaby, Danny Ding, Osbert Bastani, and Isil Dillig. 2026. Choose, Don't Label: Multiple-Choice Query Synthesis for Program Disambiguation. (2026). https://doi.org/10.48550/arXiv.2604.08792

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

[10] Nader H Bshouty. 1995. Exact learning boolean functions via the monotone theory. *Information and Computation* 123, 1 (1995), 146–153.

[11] Nader H Bshouty, Richard Cleve, Sampath Kannan, and Christino Tamon. 1994. Oracles and queries that are sufficient for exact learning. In *Proceedings of the seventh annual conference on Computational learning theory*. 130–139.

[12] Varun Chandrasekaran, Kamalika Chaudhuri, Irene Giacomelli, Somesh Jha, and Songbai Yan. 2020. Exploring connections between active learning and model extraction. In *29th USENIX Security Symposium (USENIX Security 20)*. 1309–1326.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[14] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 487–502.

[15] Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. 2024. Fast and Reliable Program Synthesis via User Interaction. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 963–975. doi:10.1109/ASE56229.2023.00129

[16] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2022. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875* (2022).

[17] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (Sept. 2003), 752–794. doi:10.1145/876638.876643

[18] Sanjoy Dasgupta. 2004. Analysis of a greedy active learning strategy. *Advances in neural information processing systems* 17 (2004).

[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[20] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Small formulas for large programs: on-line constraint simplification in scalable static analysis. In *Proceedings of the 17th International Conference on Static Analysis* (Perpignan, France) *(SAS'10)*. Springer-Verlag, Berlin, Heidelberg, 236–252.

[21] Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. 254–278. doi:10.1007/978-3-319-63387-9_13

[22] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435.

[23] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing

Machinery, New York, NY, USA, 422–436. doi:10.1145/3062341.3062351

[24] Margarida Ferreira, Miguel Terra-Neves, Miguel Ventura, Inês Lynce, and Ruben Martins. 2021. FOREST: An Interactive Multi-tree Synthesizer for Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part I* (Luxembourg City, Luxembourg). Springer-Verlag, Berlin, Heidelberg, 152–169. doi:10.1007/978-3-030-72016-2_9

[25] Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 191–202.

[26] Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2020. Interactive synthesis of temporal specifications from examples and natural language. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–26.

[27] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*. Springer-Verlag, Berlin, Heidelberg, 72–83.

[28] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[29] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[30] Di Huang, Rui Zhang, Xing Hu, Xishan Zhang, Pengwei Jin, Nan Li, Zidong Du, Qi Guo, and Yunji Chen. 2022. Neural program synthesis with query. *arXiv preprint arXiv:2205.07857* (2022).

[31] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

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

[33] Ruyi Ji, Chaozhe Kong, Yingfei Xiong, and Zhenjiang Hu. 2023. Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 103 (apr 2023), 29 pages. doi:10.1145/3586055

[34] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question selection for interactive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1143–1158. doi:10.1145/3385412.3386025

[35] Tak Yeon Lee, Casey Dugan, and Benjamin B. Bederson. 2017. Towards Understanding Human Mistakes of Programming by Example: An Online User Study. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces* (Limassol, Cyprus) *(IUI '17)*. Association for Computing Machinery, New York, NY, USA, 257–261. doi:10.1145/3025171.3025203

[36] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6038–6049.

[37] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2019. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*. 577–589.

[38] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 291–301.

[39] Brad A. Myers and Richard McDaniel. 2001. Chapter 3 - Demonstrational Interfaces: Sometimes You Need a Little Intelligence, Sometimes You Need a Lot. In *Your Wish is My Command*, Henry Lieberman (Ed.). Morgan Kaufmann, San Francisco, 45–III. doi:10.1016/B978-155860688-3/50004-X

[40] Jakob Nielsen. 1993. Response Times: The 3 Important Limits. https://www.nngroup.com/articles/response-times-3-important-limits/#:~:text=,not%20know%20what%20to%20expect

[41] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[42] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 150 (Oct. 2018), 28 pages. doi:10.1145/3276520

[43] Hila Peleg, Shachar Itzhaky, Sharon Shoham, and Eran Yahav. 2020. Programming by predicates: a formal model for interactive synthesis. *Acta Informatica* 57, 1 (2020), 165–193.

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

[45] Willard V Quine. 1959. On cores and prime implicants of truth functions. *The American Mathematical Monthly* 66, 9 (1959), 755–760.

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

[47] Daniel Ramos, Ines Lynce, Vasco Manquinho, and Ruben Martins. 2020. Program disambiguation using MaxSAT. *MaxSAT Evaluation 2020* (2020), 58.

[48] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B Gupta, Xiaojiang Chen, and Xin Wang. 2021. A survey of deep active learning. *ACM computing surveys (CSUR)* 54, 9 (2021), 1–40.

[49] Greg Schohn and David Cohn. 2000. Less is more: Active learning with support vector machines. In *ICML*, Vol. 2. Citeseer, 6.

[50] Burr Settles. 2009. Active learning literature survey. (2009).

[51] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 289–299. doi:10.1145/2025113.2025153

[52] Kota Sueyoshi and Takashi Matsubara. 2024. Predicated Diffusion: Predicate Logic-Based Attention Guidance for Text-to-Image Diffusion Models. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 8651–8660. doi:10.1109/CVPR52733.2024.00826

[53] Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF international conference on computer vision*. 11888–11898.

[54] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1631–1634. doi:10.1145/3035918.3058738

[55] Xinyu Wang, Greg Anderson, Isil Dillig, and Kenneth L McMillan. 2018. Learning abstractions for program synthesis. In *International Conference on Computer Aided Verification*. Springer, 407–426.

[56] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[57] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. doi:10.1145/3379337.3415900

[58] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. 2024. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294* (2024).

Received 2025-11-07; accepted 2026-04-03