Fast and Reliable Program Synthesis via User Interaction

Yanju Chen
yanju@cs.ucsb.edu
University of California, Santa Barbara
USA

Chenglong Wang
chenglong.wang@microsoft.com
Microsoft Research
USA

Xinyu Wang
xwangsd@umich.edu
University of Michigan
USA

Osbert Bastani
obastani@seas.upenn.edu
University of Pennsylvania
USA

Yu Feng
yufeng@cs.ucsb.edu
University of California, Santa Barbara
USA

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00129

***Abstract*—The performance of programming-by-example systems varies significantly across different tasks and even across different examples in one task. The key issue is that the search space depends on the given examples in a complex way. In particular, scalable synthesizers typically rely on a combination of machine learning to prioritize search order and deduction to prune search space, making it hard to quantitatively reason about how much an example speeds up the search. We propose a novel approach for quantifying the effectiveness of an example at reducing synthesis time. Based on this technique, we devise an algorithm that actively queries the user to obtain additional examples that significantly reduce synthesis time. We evaluate our approach on 30 challenging benchmarks across two different data science domains. Even with ineffective initial user-provided examples for pruning, our approach on average achieves a 6.0× speed-up in synthesis time compared to state-of-the-art synthesizers.**

I. Introduction

Due to its potential to significantly improve both programmer productivity and software correctness, programming by examples (PBE) has recently received significant attention from researchers. In most domains, the number of possible programs is enormous, making explicit enumerative search intractable. Thus, modern synthesizers aggressively prune the search space using logical deduction [1, 2, 3, 4, 5], as well as use machine learning to bias the search towards programs that are more likely to satisfy the specification [6, 3, 7, 8].

Despite the progress that has been made, synthesizers still face performance challenges that inhibit their usability. We consider two performance metrics:

- **Scalability:** This metric is the synthesis time for a set of IO examples. Many synthesizers [4, 5, 2, 9] are designed to optimize scalability since it is key to enabling rapid responses to the user. However, there is often a long tail of programs that still take significant time to synthesize.
- **Reliability:** This metric is the variance in synthesis times across different IO examples specifying the same program. Even if a synthesizer is fast given one set of examples, it may not be so for a different set of examples specifying the same program. Enhancing reliability is important in making PBE more usable to a broad audience beyond those who have the expertise to provide "good" examples. However, existing state-of-the-art example generation techniques [10, 11, 12] are not designed with this objective in mind and are sensitive to the initial choice of IO examples.

Intuitively, scalability corresponds to minimizing expected synthesis time across IO examples, whereas reliability corresponds to minimizing variance in synthesis time across IO examples. While the two are closely related, it is possible for a scalable synthesizer to be unreliable and vice versa.

We propose FAERY[1], a novel programming-by-example framework that leverages user interaction to improve both scalability and reliability. Given an initial set of user-provided examples, FAERY performs deduction-guided enumerative search to find a program that satisfies these examples. If FAERY is not able to find a correct program within a short amount of time, it selects an additional input example and queries the user to obtain the corresponding output. Then, FAERY takes this new IO example and continues its search. This query is selected to most reduce the (expected) synthesis time, thereby improving scalability. It also improves reliability since it makes the synthesis time less sensitive to the initial IO examples provided by the user.

Choosing additional examples to achieve both high scalability and high reliability is a challenging problem. Our approach significantly improves both scalability and reliability. Intuitively, it does so by choosing a second input to query that most improves expected synthesis time. This strategy naturally improves scalability. Importantly, it also naturally improves reliability: choosing an input to query that significantly prunes the search space directly reduces the dependence of synthesis time on the user-provided example, thereby improving reliability.

In our setting, the primary challenge is how to choose a query that most reduces synthesis time. Our strategy is to

[1]Fast And rEliable pRogram sYnthesis.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00129

estimate the reduction for each query in a set of candidates and choose the one that maximizes this estimate. To do so, we need to solve two key problems: First, we need to quantitatively estimate how much a query reduces synthesis time; Second, it must be estimated quickly, since time taken by the estimation procedure is part of the overall synthesis time.

Because state-of-the-art synthesizers [4, 7, 3] typically rely on a combination of machine learning to prioritize search order and deduction to prune search space, to address the first problem in a realistic setting, we devise a stylized model of the synthesizer for which we can rigorously reason about the synthesis time. In this model, the synthesizer employs a stochastic search strategy that combines a machine learning model to prioritize the search order with a deduction engine to prune parts of the search space. To address the second, we devise an algorithm that approximates the expected synthesis time using random samples from the search space. Intuitively, this algorithm estimates the expected fraction of the search space pruned by the query.

We instantiate FAERY on two data science domains, table transformations and JSON tree transformations, and evaluate its effectiveness on 30 challenging benchmarks. Our results show that even with a single user query, FAERY achieves 2.4× speed-up in the table transformation domain and 9.5× speed-up in the JSON transformation domain compared to the state-of-the-art synthesizers [4, 13]. In addition, we show that FAERY can reduce the variation in running time across different IO examples for a fixed benchmark. With just a single user query, FAERY reduces this variation in performance for benchmarks in both domains, demonstrating that users who cannot provide high quality IO examples can still use FAERY to solve challenging tasks.

In summary, we make the following key contributions:

- We propose and formalize the problem of improving the overall speed and reliability of a program synthesizer by leveraging user interaction (Section II).
- We propose a novel interactive synthesis algorithm that addresses this problem. Our algorithm uses a stylized model of the synthesizer to quantify synthesis time, and uses an efficient statistical estimator for it (Section III).
- We implement our approach in a tool called FAERY (Section IV), and empirically demonstrate its benefits compared to other state-of-the-art synthesizers (Section V).

II. PROBLEM FORMULATION

We consider syntax-guided synthesis [14], where the specifications are provided as input-output (IO) examples [9]. That is, given a domain-specific language L and examples E, the synthesis problem is to find a program in L that satisfies every example $(e_{in}, e_{out}) \in E$. Because state-of-the-art synthesizers [5, 3, 4] typically combine statistical models (to guide the search) with deduction engines (to prune infeasible programs), in this paper, we are concerned with synthesis techniques that combine deduction-based pruning with search prioritization using a statistical policy [6].

Our goal is to leverage user interaction to obtain additional IO examples that help speed up synthesis. In particular, we query the user on an additional input example from a predefined set of candidate inputs to obtain the desired output. Then, the deduction engine can use the additional IO example to prune a significantly larger portion of the search space, compared to only using initial examples E. For simplicity, we restrict to the case where our algorithm acquires exactly one new additional IO example, and it is straightforward to generalize the algorithm with multiple interactions.

To formalize this problem, we devise a stylized model of how our synthesizer works, which captures its key features yet is sufficiently simple that we can reason explicitly about its running time. Then, our goal is to select an additional input example that most reduces the (expected) running time of this stylized synthesizer. As long as our model is sufficiently realistic, the same strategy should also work well for our actual synthesizer.

A. Preliminaries

We assume the domain-specific language (DSL) L is in a context-free grammar (V, Σ, R, S), where V is the nonterminals, Σ is the terminals, R is the productions, and S is the start symbol. A *partial program* $P \in \mathcal{P}$ is a sequence $P \in (\Sigma \cup V)^*$ such that $S \overset{*}{\Rightarrow} P$ (i.e., P can be derived from S via a sequence of productions). A nonterminal in P is a *hole*; P is *complete* (denoted $P \in \overline{\mathcal{P}}$) if it does not have any holes. A production $r \in R$ is *valid* for P if it can be used to fill the left-most hole in P; we denote the resulting partial program by $P' = \textsc{Fill}(P, r)$ whenever $P \overset{r}{\Rightarrow} P'$ — here either P' is complete, or there is some production $r \in R$ that is valid for P. Next, $\mathcal{P}$ is a lattice with partial order $\sqsubseteq$, where $P' \sqsubseteq P$ if and only if P' can be derived from P in L (i.e., $P \overset{*}{\Rightarrow} P'$). In this case, P' is a *refinement* of P; if P' is complete, it is a *completion* of P.

Example 1: Consider the following partial program P: `map(◯, +1)` and production $r \equiv \bigcirc \rightarrow$ `reverse`$(\bigcirc)$. In this case, $\textsc{Fill}(P, r)$ yields the following partial program P': `map(reverse(◯), +1)`.

Given a set of IO examples E and DSL L, a complete program $P \in \overline{\mathcal{P}}$ *satisfies* E (denoted $P \models E$) if $e_{out} = [\![P]\!]e_{in}$ for all $(e_{in}, e_{out}) \in E$, where $[\![\cdot]\!]$ is the concrete semantics. The *synthesis problem* for E is to find $P \in \overline{\mathcal{P}}$ such that $P \models E$; we call such a P a *solution* for E.

Deduction engine. Given a set of IO examples E, a partial program P is *feasible* if there exists a completion P' of P such that $P' \models E$. We consider a *deduction engine* that checks whether P is feasible for E using abstract semantics $[\![\cdot]\!]^\#$ that overapproximates $[\![\cdot]\!]$. In particular, $[\![\cdot]\!]^\#$ maps a concrete input e_{in} to an abstract value $\hat{e}_{out} = [\![P]\!]^\# e_{in}$, which is a set of concrete values satisfying the soundness condition $(P' \textit{ is completion of } P) \Rightarrow ([\![P']\!]e_{in} \in [\![P]\!]^\# e_{in})$, This property of $[\![\cdot]\!]^\#$ enables us to use $[\![\cdot]\!]^\#$ to prune the search space. In particular, given an IO example $e = (e_{in}, e_{out})$ and a partial program P, we say e *prunes* P if $e_{out} \notin [\![P]\!]^\# e_{in}$, which we denote by $P \not\models e$. Given a set of IO examples E,

we say E prunes P if there exists an example $e \in E$ such that $P \not\models e$, which we denote by $P \not\models E$.

Notation. In general, we use E to denote a set of IO examples where each example $e = (e_{in}, e_{out})$, E_{in} to denote a set of inputs e_{in}, E_{out} to denote a set of outputs e_{out}, and $\hat{E}_{out}$ to denote a set of abstract outputs $\hat{e}_{out}$. In addition, we use $\hat{E}$ to denote a set of input-abstract-output (IAO) examples $\hat{e} = (e_{in}, \hat{e}_{out})$—i.e., where $\hat{e}_{out} = [\![P]\!]^{\#} e_{in}$ for some $P \in \mathcal{P}$.

Example 2: Consider the following input-output example in list manipulation:

$$e_{in} : [1, 3, 5, 2, 4] \mapsto e_{out} : [5, 4, 3]$$

Using the size of the list as the abstract domain [4], the partial program P: `reverse(map(`e_{in}`, ◯))` is infeasible (i.e., $P \not\models e$). In particular, no matter how we fill hole ◯, the resulting program cannot satisfy the given IO example for the following reason:

- The `map` construct applies a function (yet to be determined by the synthesizer) over every element of e_{in} and yields an output list whose length equals that of the input list e_{in}.
- The `reverse` construct reverses its input, making the size of the output list the same as its input.
- Since the output returned by `reverse` does not have the same size as the desired output e_{out}, we derive an inconsistency. i.e., $size(e_{in}) == size(e_{out}) \wedge size(e_{in}) == 5 \wedge size(e_{out}) == 3$ is UNSAT.

Several techniques from prior work (e.g., [3, 5, 4, 15]) can prove the infeasibility of such partial programs by using an SMT solver (provided specifications are given for the DSL constructs).

Statistical policy. We consider a *statistical policy* π used to prioritize the search order. Given a partial program P, it assigns probabilities $\pi(r \mid P)$ to productions $r \in R$ that can be used to fill the left-most hole in P. Let $\mathcal{Z} = \mathcal{P}^*$ be the space of sequences of programs. Then, we sample partial programs from the search space using π as follows:

Definition 2.1 (Rollout): Given a set of IO examples E, a *rollout* $\zeta \in \mathcal{Z}$ is a random sequence of partial programs $\zeta = (P_1, ..., P_n)$ such that (i) $P_1 = S$ is the start symbol of L, (ii) $P_{i+1} = \textsc{Fill}(P_i, r_i)$ is the partial program constructed by sampling $r_i \sim \pi(\cdot \mid P_i)$ and using it to fill the left-most hole in P_i , (iii) $P_i \models E$ for all $i < n$, and (iv) letting $P_\zeta = P_n$, either $P_\zeta \in \overline{\mathcal{P}}$ or $P_\zeta \not\models E$.

Example 3: According to Example 2, the partial program `reverse(map(`e_{in}`, ◯))` is a terminal state as it gets pruned by the deduction engine. Thus, the following sequence corresponds to a rollout:

$(S, S \to \bigcirc),\ (\bigcirc, \bigcirc \to \texttt{reverse}(\bigcirc)),$
$(\texttt{reverse}(\bigcirc), \bigcirc \to \texttt{map}(\bigcirc, \bigcirc)),$
$(\texttt{reverse}(\texttt{map}(\bigcirc, \bigcirc)), \bigcirc \to e_{in}), (\texttt{reverse}(\texttt{map}(e_{in}, \bigcirc)), \varnothing).$

We use $Z(E) \subseteq \mathcal{Z}$ to denote the rollouts for E. Intuitively, a rollout is a single sequence of samples from the search space that terminates in a program P_ζ that is either a solution to the synthesis problem for E or that is pruned by deduction using E. Note that we sample a rollout ζ with probability

$$p(\zeta \mid E) = \pi(r_1) \cdot \pi(r_2 \mid P_1) \cdot ... \cdot \pi(r_n \mid P_{n-1}),$$

i.e., sequentially sample each r_n conditioned on P_{n-1}. We prove that $p(\zeta \mid E)$ is a probability distribution over $Z(E)$—i.e., it is properly normalized; see Appendix A for a proof.

Theorem 2.2: We have $\sum_{\zeta \in Z(E)} p(\zeta \mid E) = 1$.

B. Model of Synthesis Running Time

Our stylized model is designed to balance faithful approximation of a real synthesizer such as Neo [4] with a tractable quantification of running time. In particular, it captures two critical components of state-of-the-art synthesizers [4, 2]: (i) the use of a statistical policy π to determine which parts of the search space to prioritize, and (ii) a deduction engine used to prune the search space. The main approximation is that the stylized synthesizer does not perform an enumerative search according to the probabilities of π; the running time of such a search is hard to quantify because it depends on the rankings of the predicted probabilities of different partial programs rather than directly using the probabilities.

Instead, using the $\textsc{Fill}(P_i, r_i)$ procedure described earlier in Section II-A, the stylized synthesizer randomly samples i.i.d. partial programs P according to π until it samples one such that $P \models E$. In each iteration, if P is a complete program and satisfies examples E according to the deduction engine, we return it as a solution. Otherwise, the current partial P may be infeasible thus gets pruned by deduction. In that case, the synthesizer has to keep sampling new partial programs.

Definition 2.3 (Running Time): The *running time* ρ of the stylized synthesizer is the random variable with distribution $p(\rho \mid E)$ that counts the number of times the deduction engine is invoked on some partial program P to check whether $P \models E$.

C. Fast and Reliable Synthesis

Our goal is to maximize the speed and reliability of the synthesizer. Our primary objective is speed; as we show in our experiments, reliability improves as a byproduct of using interaction to improve speed. We formalize these two metrics and our problem below.

Metrics. We assume given a distribution $p(E)$ over the *initial* IO examples E given by the user; then, the joint distribution over running time ρ and E is $p(E, \rho) = p(\rho \mid E) \cdot p(E)$.

Definition 2.4 (Expectation and Variability of Running Time): The *expected running time* is $\mu = \mathbb{E}_{p(E,\rho)}[\rho]$, and the *variability* is $\sigma^2 = \mathbb{E}_{p(E,\rho)}[(\rho - \mu)^2]$.

In other words, expected running time and variability are the mean and variance of running time ρ, respectively. Then, our synthesizer is fast if it has low expected running time, and is reliable if it has low variability. When there is no ambiguity between the running time ρ and the expected running time μ, we simply refer to μ as the running time.

Remark 2.5: The expected running time is essentially the expected pruning power defined in Section II-B, except (i) it measures the fraction *not* pruned instead of the fraction pruned, (ii) it considers the entire search space instead of the unexplored search space, which is a reasonable approximation since we only explore a small fraction of the search space

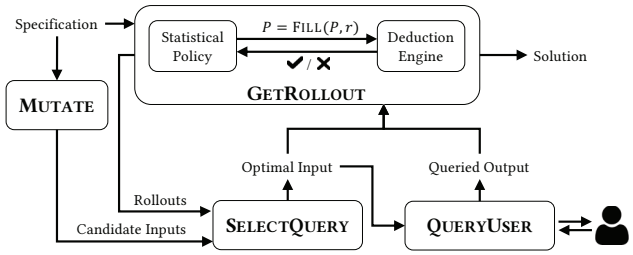

Fig. 1: Overview of stylized synthesizer with user interaction.

before querying the user, and (iii) it weights partial programs by their probability according to π. In particular, we show later that it equals the expected fraction of partial programs not pruned along a single rollout from π.[2]

Therefore, our goal is essentially to minimize the (expected) running time μ.

Problem formulation. Given 1) initial input-output examples E, 2) candidate inputs E'_{in} generated by mutations over $e_{in} \in E_{in}$, and 3) a stylized synthesizer that is composed by a search strategy guided by statistical policy π, and a deduction engine, we want to select $e'_{in} \in E'_{in}$ such that (e'_{in}, e'_{out}) minimizes the (expected) running time of the stylized synthesizer. Here, e'_{out} is obtained from the user via interaction.

III. Interactive Synthesis Algorithm

In this section, we describe our interactive synthesis algorithm on top of the stylized synthesizer described in Section II-B. First, we briefly explain each of its components. Then we formally give the estimation of expected running time of the interactive synthesis algorithm. Based on that, we further give a detailed discussion of the SELECTQUERY procedure, which is one of the main contributions of this paper. Since MUTATE is orthogonal to the main idea of this paper, we defer its detailed discussion Section IV.

A. Interactive Synthesis Algorithm

Figure 1 shows the overview design of our interactive synthesis algorithm. Given the initial input examples, FAERY first invokes the MUTATE subroutine to generate candidate input examples through mutations. Since the initial IO examples may not be good enough to significantly prune the search space, FAERY may optionally invokes the SELECTQUERY subroutine to select an additional input (from the candidate pool generated by MUTATE) whose corresponding output is obtained by interacting with the user through the QUERYUSER procedure. The queried output example together with the corresponding input examples will be an additional IO example that can be utilized to speed up synthesis. To reduce the number of interactions, SELECTQUERY must choose additional examples in a way that *minimizes* the expected running time.

We summarize the interactive algorithm in Algorithm 1. Formally, the stylized synthesizer samples i.i.d. rollouts ζ until it samples one such that $P_\zeta \models E$. In each iteration (lines 13–21), it samples a rollout ζ by calling GETROLLOUT (line 13).

[2]See Theorem 3.5.

Algorithm 1 Interactive Stylized Program Synthesis

Input: Initial IO Examples E, Candidate Input Examples E'_{in}
Output: Solution P_ζ

```
1:  procedure GETROLLOUT(E)
2:      P ← S;   ζ ← [P]
3:      while true do
4:          r ~ π(· | P)
5:          P ← FILL(P, r)
6:          if P ⊭ E or P ∈ P̄ return ζ
7:          ζ ← ζ ∪ [P]
8:      end while
9:  end procedure

10: procedure SYNTHESIZE(E, E'_in)
11:     Z ← []
12:     while true do
13:         ζ ← GETROLLOUT(E)
14:         if P_ζ is complete and P_ζ satisfies E then return P_ζ
15:         else Z ← Z ∪ [ζ]
16:         end if
17:         if |Z| = N then
18:             e'_in ← SELECTQUERY(E, E'_in, Z)
19:             e'_out ← QUERYUSER(e'_in)
20:             E ← E ∪ {(e'_in, e'_out)}
21:         end if
22:     end while
23: end procedure
```

If P_ζ is a complete program and satisfies examples E, we return it as a solution (line 14). Otherwise, the algorithm adds ζ into Z (line 15). GETROLLOUT procedure initializes P to be the start symbol (line 2), and then iteratively expands P by sampling $r \sim \pi(\cdot \mid P)$ and using it to fill the left-most hole in P. At each step, it may return if P is infeasible or complete.

Interaction. Note that FAERY will initialize an interaction as soon as the number of rollouts $|Z|$ reaches some hyperparameter $N \in \mathbb{N}$ (line 17). In practice, we found that performance is not very sensitive to the number of rollouts used to compute the optimal query. Instead, how to select e'_{in} from E'_{in} is the key challenge to achieving good performance.

We focus on a single interaction since this case has the highest payoff in terms of improvement in performance for a fixed amount of user effort; However, if the user is interested in providing additional IO examples, our algorithm for selecting queries e'_{in} could simply be applied multiple times.

With interaction, the algorithm calls SELECTQUERY (line 18) to obtain additional input example $e'_{in} \in E'_{in}$ based on the set of rollouts Z sampled so far and the precomputed set of candidate input examples E'_{in}. Then it queries the user by QUERYUSER (line 19) to obtain corresponding output for a new IO example (e'_{in}, e'_{out}) and add it to the initial examples (line 20).

Finally, a potential issue is that additional IO examples might change the set of solutions—i.e., we may have $p \models E$

but $p \not\models E \cup \{(e'_{in}, e'_{out})\}$. For simplicity, we assume that the user will answer the query (i.e., e'_{out}) that is consistent with her original intent and the set of solutions does not change. As we show later in Section V-A, in practice, most of our tasks are significantly constrained by the initial IO examples and this assumption holds.

B. Estimation of Expected Running Time

Since the goal of SELECTQUERY subroutine is to pick a candidate input example that minimizes the expected running time, here we first show how we estimate it with a single user interaction. Given a set of initial IO examples E, we design our synthesis algorithm to minimize the *conditional* expected running time $\mu(E) = \mathbb{E}_{p(\rho|E)}[\rho]$. Note that the expected running time is $\mu = \mathbb{E}_{p(E)}[\mu(E)]$; thus, this approach minimizes μ. As we show in our experiments, it also helps reduce the variability σ^2.

To this end, we show how we make the running time of the stylized synthesizer tractable to compute. First, we devise a formula for its running time $\mu_0(E)$ when there is no interaction—i.e., $N = \infty$ of line 17 in Algorithm 1. Based on this formula, we devise a formula for the running time $\mu(E)$ that queries the user on a single additional input example e'_{in}. Thus, our goal is to compute e'_{in} that minimizes this formula.

Running time with no interactions. Essentially, computing the running time corresponds to counting how many times line 6 in Algorithm 1 is called. In particular, for a single rollout $\zeta = (P_1, ..., P_n)$, this line is executed n times; thus, the running time of the stylized synthesizer is $\sum_{i=1}^{M} |\zeta_i|$, where M is the (random) number of rollouts sampled until a solution is found, and $|\zeta_i|$ is the length of ζ_i.

Lemma 3.1: For IO examples E, the running time of the stylized synthesizer with no interactions is $\mu_0(E) = \ell(E)/\tau^*(E)$, where $\ell(E) = \mathbb{E}_{p(\zeta|E)}[|\zeta|]$ is the expected rollout length and $\tau^*(E) = \mathbb{P}_{p(\zeta|E)}[P_\zeta \models E]$ is the probability of a solution. Note that different statistical policies may compute $\tau^*(E)$ differently; in our algorithm we make no assumption about its specific form.

We give a proof in Appendix A. Note that $\ell(E)$ depends on E since a rollout proceeds until $P_\zeta \not\models E$ (or $P_\zeta \in \overline{\mathcal{P}}$).

Running time with a single interaction. Next, we consider the running time of our stylized synthesizer when it issues at most one single user query. Suppose the initial IO examples are E, and the synthesizer queries the user to obtain a single additional IO example (e'_{in}, e'_{out}). Then, this case is almost equivalent to running the synthesizer with no interactions on IO examples $E \cup \{(e'_{in}, e'_{out})\}$—i.e., $\mu(E) = \mu_0(E \cup \{(e'_{in}, e'_{out})\})$. However, there is a chance that the synthesizer finds a solution before it queries the user; our formula accounts for this possibility.

Lemma 3.2: Given a set of IO examples E, the running time of the stylized synthesizer with a single interaction is

$$\mu(E) = \alpha(E) + \beta(E) \cdot \mu_0(E \cup \{(e'_{in}, e'_{out})\}),$$

where $e'_{in} = \text{SELECTQUERY}(E, E'_{in}, Z)$ is the selected query, $e'_{out} = \text{QUERYUSER}(e'_{in})$ is the user response, and $\alpha(E)$ and $\beta(E)$ are positive constants independent of e'_{in} and e'_{out}. We define $\alpha(E)$ and $\beta(E)$ along with a proof in Appendix B.

Optimal query. Now, given a set of candidate inputs E'_{in}, we can establish a formula characterizing the optimal query e^*_{in} that most reduces running time. One challenge is handling the fact that we do not know the corresponding outputs $e'_{out} = \text{QUERYUSER}(e'_{in})$ for inputs $e'_{in} \in E'_{in}$. To address this issue, we assume we know the distribution $p(e'_{out} \mid e'_{in})$ over outputs e'_{in} for the input e'_{out}.

Then, the optimal input example to query that minimizes the expected running time of the stylized synthesizer is

$$e^*_{in} = \underset{e'_{in} \in E'_{in}}{\arg\min}\, \mathbb{E}_{p(e'_{out}|e'_{in})}[\mu(E)]. \tag{1}$$

We have the following; see Appendix C for a proof:

Theorem 3.3: We have $e^*_{in} = \arg\min_{e'_{in} \in E'_{in}} J(e'_{in}; E)$, where

$$J(e'_{in}; E) = \mathbb{E}_{p(e'_{out}|e'_{in})}[\ell(E \cup \{(e'_{in}, e'_{out})\}].$$

Thus, we want to choose e'_{in} that minimizes the rollout length $\ell(E \cup \{(e'_{in}, e'_{out})\}$ in expectation over $p(e'_{out} \mid e'_{in})$. The key challenge in computing e^*_{in} is that we do not know the distribution $p(e'_{out} \mid e'_{in})$, which is the probability that the user responds e'_{out} when queried on a given input example $e'_{in} \in E'_{in}$. Assuming the user has a target program $P \in \overline{\mathcal{P}}$ in mind, and we know the probability $p(P)$, we could decompose this probability as

$$p(e'_{out} \mid e'_{in}) \propto \sum_{P \in \mathcal{P}} \mathbb{1}(e'_{out} = [\![P]\!] e'_{in}) \cdot \mathbb{1}(P \models E) \cdot p(P),$$

where $\mathbb{1}$ is the boolean predicate function[3]. In other words, the probability they have in mind P restricted to complete programs consistent with the initial IO examples E and programs that evaluate to e'_{out} on input e'_{in}. A natural choice for $p(P)$ would be the probability of sampling P using π.

However, since the original synthesis problem was to compute $P \in \overline{\mathcal{P}}$ that satisfies E, drawing a single sample P such that $P \models E$ is computationally infeasible. Instead, our algorithm uses heuristics to approximately optimize $J(e'_{in}; E)$; we discuss these heuristics in Section III-C.

Therefore, we reduce the intractable expected running time to a tractable objective based on expected rollout length. We'll then elaborate how SELECTQUERY subroutine computes the objective and selects an optimal query.

C. SELECTQUERY Subroutine

Algorithm 2 shows the design of the SELECTQUERY subroutine, which selects an input example $e'_{in} \in E'_{in}$ that optimizes $J(e'_{in}; E)$. Its inputs are the initial IO examples E, a precomputed set of candidate input examples E'_{in}, and a set of rollouts Z sampled so far, and it outputs an example $e'_{in} \in E'_{in}$. The key challenge is that we do not know the distribution $p(e'_{out} \mid e'_{in})$. To address this issue, as shown by Figure 2,

[3] A boolean predicate function $\mathbb{1}(A)$ is commonly defined as $\mathbb{1}(A) = \begin{cases} 1 & \text{if } A \\ 0 & \text{if } \neg A \end{cases}$.

Algorithm 2 Query Selection

Input: Initial IO Examples E, Candidate Input Examples E'_{in} and Rollouts Z
Output: Optimal Query $\arg\min_{e'_{in} \in E'_{in}} J^{\mathcal{A}}_{\text{abs}}(e'_{in})$

1: **procedure** SELECTQUERY(E, E'_{in}, Z)
2: **for** $e'_{in} \in E'_{in}$ **do**
3: $\hat{E}'_{out} \leftarrow [\![P]\!]^{\#} e'_{in} \mid \zeta \in Z,\ P \in \zeta,\ P \models E]$
4: **for** $\hat{e}'_{out} \in \hat{E}'_{out}$ **do**
5: $\ell((e'_{in}, \hat{e}'_{out})) \xleftarrow{} \frac{1}{|Z|} \sum_{\zeta \in Z} |\mathcal{T}(\zeta, (e'_{in}, \hat{e}'_{out}))|$
6: **end for**
7: $J^{\mathcal{A}}_{\text{abs}}(e'_{in}) \leftarrow \mathcal{A}\left\{\lambda \hat{e}'_{out}.\ell((e'_{in}, \hat{e}'_{out})); \hat{E}'_{out}\right\}$
8: **end for**
9: **return** $\arg\min_{e'_{in} \in E'_{in}} J^{\mathcal{A}}_{\text{abs}}(e'_{in})$
10: **end procedure**

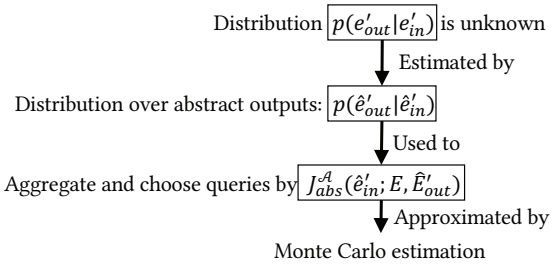

Fig. 2: Workflow of SELECTQUERY.

our algorithm considers abstract outputs—i.e., $p(\hat{e}'_{out} \mid e'_{in})$. Then, for each candidate input $e'_{in} \in E'_{in}$, our algorithm approximates $p(e'_{out} \mid e'_{in})$ to be the set $\hat{E}'_{out}$ of possible abstract outputs $[\![P]\!]^{\#} e'_{in}$ of sampled partial programs P (line 3). Then, it computes the rollout length $\ell(E \cup \{(e'_{in}, \hat{e}'_{out})\})$ for each $\hat{e}'_{out} \in \hat{E}'_{out}$ using the Monte Carlo estimate below (line 5). Next, based on this estimate, it evaluates a variant $J^{\mathcal{A}}_{\text{abs}}(e'_{in}; E, \hat{E}'_{out})$ of the objective $J(e'_{in}; E)$ by aggregating over $\hat{e}'_{out} \in \hat{E}'_{out}$ (line 7). Finally, it selects the query e'_{in} that minimizes this objective (line 9). We describe these steps in detail below.

Distribution over abstract outputs. Since we do not know the output e'_{out} of each candidate input example $e'_{in} \in E'_{in}$ in advance, as a heuristic, we instead consider abstract outputs $\hat{e}'_{out}$. We can sample $\hat{e}'_{out}$ for e'_{in} by sampling *partial* programs $P \in \mathcal{P}$ consistent with the initial IO examples E, and evaluating $\hat{e}'_{out} = [\![P]\!]^{\#} e'_{in}$. To sample such a P, we sample a rollout $\zeta = (P_1, ..., P_n)$ according to $p(\zeta \mid E)$; by definition, $P_i \models E$ for all $i < n$, so each P_i is a valid sample. We denote the distribution over abstract outputs sampled this way by $p(\hat{e}'_{out} \mid e'_{in})$.

Furthermore, we can easily extend deduction to handling such input-abstract-output (IAO) examples. In particular, we let $P \models \hat{e}$ if $[\![P]\!]^{\#} e_{in} \sqsubseteq \hat{e}_{out}$—i.e., we assume P can be pruned unless it is provably consistent with $\hat{e}$. This definition extends to sets of IAO examples $\hat{E}$ in the obvious way—i.e., $P \not\models \hat{E}$ if there exists $\hat{e} \in \hat{E}$ such that $[\![P]\!]^{\#} \not\models \hat{e}$; it also works with combinations of IO examples and IAO examples.

Then, we replace the objective $J(e'_{in}; E)$ with the following:

$$J_{\text{abs}}(e'_{in}; E) = \mathbb{E}_{p(\hat{e}'_{out} | e'_{in})}[\ell(E \cup \{(e'_{in}, \hat{e}'_{out})\}],$$

where in $J_{\text{abs}}(e'_{in}; E)$ the expectation is computed over $p(\hat{e}'_{out} \mid e'_{in})$.

Aggregating over abstract outputs. A potential shortcoming of $J_{\text{abs}}(e'_{in}; E)$ is that because $p(\hat{e}'_{out} \mid e'_{in})$ is a heuristic, it might no longer yield good queries. To improve robustness to this choice, we consider a more general objective that aggregates over possible abstract outputs $\hat{E}'_{out}$ for each $e'_{in} \in E'_{in}$ using a given operator $\mathcal{A}$. In particular, given a set $\hat{E}'_{out}$ of abstract outputs for each candidate input $e'_{in} \in E'_{in}$, we replace $J_{\text{abs}}(e'_{in}; E)$ with $J^{\mathcal{A}}_{\text{abs}}(e'_{in}; E, \hat{E}'_{out})$, which is given by: $J^{\mathcal{A}}_{\text{abs}}(e'_{in}; E, \hat{E}'_{out}) = \mathcal{A}\left\{\lambda \hat{e}'_{out}.\ell(E \cup \{(e'_{in}, \hat{e}'_{out})\}); \hat{E}'_{out}\right\}$.

In other words, $\mathcal{A}$ is a function that aggregates the loss function $\ell(E \cup \{(e'_{in}, \hat{e}'_{out})\})$ over $\hat{e}'_{out} \in \hat{E}'_{out}$. Possible choices of $\mathcal{A}$ include the expectation over the empirical distribution Uniform($\hat{E}'_{out}$), the best- or worst-case over $\hat{E}'_{out}$.

Finally, our algorithm uses the set $\hat{E}'_{out}$ of abstract outputs constructed using the partial programs in the sampled rollouts $\zeta \in Z$ that are consistent with E in conjunction with the above procedure (note that all partial programs in ζ will be consistent except for the last one P_ζ).

Monte Carlo estimation. Next, we estimate $\ell(E \cup \{\hat{e}'\})$ in $J^{\mathcal{A}}_{\text{abs}}(e'_{in}; E, \hat{E}'_{out})$ using sampling—i.e., given i.i.d. samples $Z = (\zeta_1, ..., \zeta_k)$ from $p(\zeta \mid E \cup \{\hat{e}'\})$, we use the estimate $\ell(E \cup \{\hat{e}'\}) \approx \frac{1}{|Z|} \sum_{\zeta \in Z} |\zeta|$.

However, computing this estimate independently for each e' is computationally expensive due to the need to sample rollouts ζ for each candidate input $e'_{in} \in E'_{in}$ and each abstract output $\hat{e}'_{out}$ for e'_{in}. Instead, we show how we can estimate $\ell(E \cup \{\hat{e}'\})$ based on samples from $p(\zeta \mid E)$.

Definition 3.4 ($\hat{e}$-prefix): Given a set E of IO examples, a rollout $\zeta = (P_1, ..., P_m)$ sampled from $p(\zeta \mid E)$, and an IAO example $\hat{e}$, the *$\hat{e}$-prefix* of ζ is $\mathcal{T}(\zeta, E, \hat{e}) = (P_1, ..., P_n)$ such that (i) $P_i \models E$ for all $i < n$, and (ii) either $P_\zeta \in \overline{\mathcal{P}}$ or $P_\zeta \not\models e$.

Note that such an n must exist, since if it is not satisfied for any $n < m$, it must hold for $n = m$—in particular, if $P \not\models E$, then by definition we have $P \not\models E \cup \{\hat{e}\}$. Then, we have the following result; see Appendix D for a proof:

Proposition 3.5: Given IO examples E and IAO example $\hat{e}$,

$$\ell(E \cup \{\hat{e}\}) = \mathbb{E}_{p(\zeta|E)}[|\mathcal{T}(\zeta, E, \hat{e})|].$$

In other words, we can express $\ell(E \cup \{\hat{e}'\})$ in terms of samples from $p(\zeta \mid E)$ (instead of $p(\zeta \mid E \cup \{\hat{e}'\})$). In particular, given i.i.d. samples $Z = (\zeta_1, ..., \zeta_k)$ from $p(\zeta \mid E)$, we can use $\ell(E \cup \{\hat{e}'\}) \approx \frac{1}{|Z|} \sum_{\zeta \in Z} |\mathcal{T}(\zeta, E, \hat{e}')|$. Thus, rather than drawing new samples ζ from $p(\zeta \mid E \cup \{(e'_{in}, \hat{e}'_{out})\})$ for each candidate input $e'_{in} \in E'_{in}$ and each abstract output $\hat{e}'_{out} \in \hat{E}'_{out}$ corresponding to e'_{in}, we can draw them once and compute $\mathcal{T}(\zeta, E, (e'_{in}, \hat{e}'_{out}))$.

[4]For simplicity, we here use intersection rather than implication for comparing abstract outputs of the example.

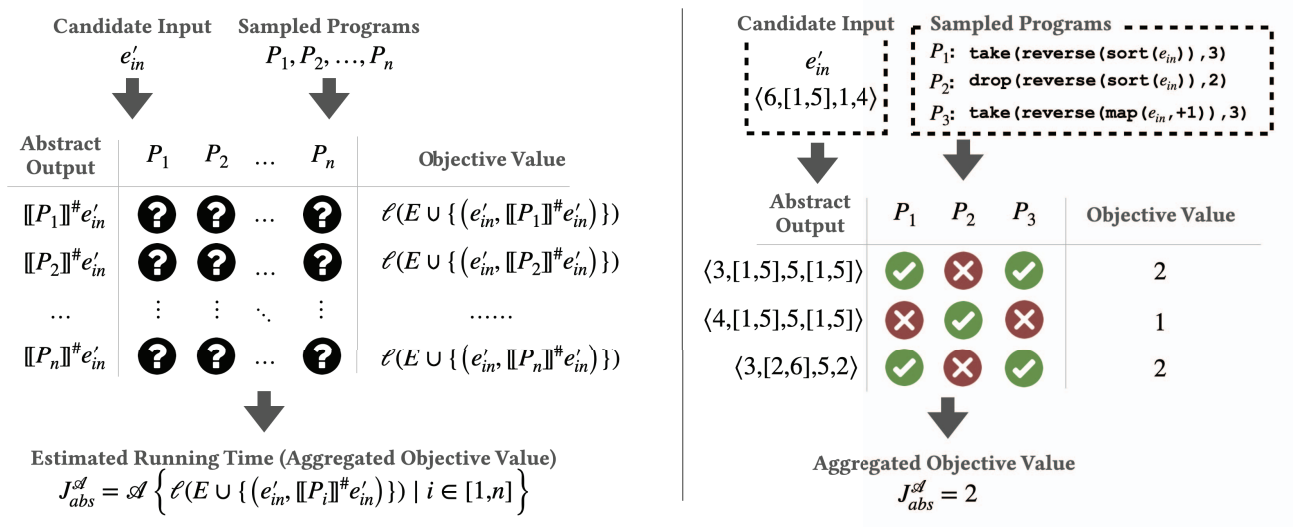

Fig. 3: Estimating the running time of an input e'_{in} by aggregating the losses from all abstract outputs. Left: Algorithmic view of running time estimation, where a question mark denotes consistency checking result; Right: A concrete example of running time estimation[4].

Example 4: We now show how one estimates the running time of a given candidate input e'_{in}. Following Example 2 FAERY first computes the abstract values $(\hat{e}_{in}, \hat{e}_{out})$ for the original example $e_{in} : [1,3,5,2,4] \mapsto e_{out} : [5,4,3]$ whose abstract values are:

$$\hat{e}_{in} : \langle 5, [1,5], 1, 4\rangle \mapsto \hat{e}_{out} : \langle 3, [3,5], 5, 3\rangle$$

where every abstract value is composed by four abstract domains: list length, value range (computed by $[min(l), max(l)]$ given a list l), first value, and last value. Then given the following candidate partial programs:

$$P_1 : \texttt{take}(\texttt{reverse}(\texttt{sort}(e_{in})), 3)$$
$$P_2 : \texttt{drop}(\texttt{reverse}(\texttt{sort}(e_{in})), 2)$$
$$P_3 : \texttt{take}(\texttt{reverse}(\texttt{map}(e_{in}, +1)), 3)$$

where all the programs (written in list manipulation DSL similar to previous works [6, 16]) are feasible given the initial example and P_1 is the intended solution, FAERY needs to pick an additional example that has a higher chance of pruning P_2 and P_3. Here `drop` (resp. `take`) truncates (resp. preserves) the last (first) n elements of a given list and returns it, where n denotes the parameter; `sort` rearranges the list in an ascending order while `reverse` flips the list backward.

As shown in Figure 3, for a given candidate input $e'_{in} : [1,3,5,3,2,4]$ and its corresponding abstract value $\hat{e}'_{in} : \langle 6, [1,5], 1, 4\rangle$, our goal is to estimate the objective value $J(e'_{in}; E)$ using the sampled partial programs P_1, P_2 and P_3. For example, abstractly evaluating P_2 on e'_{in} (i.e., $[\![P_2]\!]^{\#}e'_{in}$) yields its abstract output $\hat{e}'^{(2)}_{out}$ to be $\langle 4, [1,5], 5, [1,5]\rangle$ — i.e., the output list must contain 4 elements with the head elements being 5 and last element falls into interval $[1,5]$. FAERY evaluates all sampled programs on e'_{in} and fill in the cells as follows:

- ✓means that $[\![P_j]\!]^{\#}e'_{in}$ is guaranteed to be consistent with $\hat{e}'^{(i)}_{out}$—i.e., we have $[\![P_j]\!]^{\#}e'_{in} \sqsubseteq \hat{e}'^{(i)}_{out}$
- ✗means that $[\![P_j]\!]^{\#}e'_{in}$ may be inconsistent with $\hat{e}'^{(i)}_{out}$—i.e., we have $[\![P_j]\!]^{\#}e'_{in} \not\sqsubseteq \hat{e}'^{(i)}_{out}$

According to the Figure 3 (right), P_1 and P_2 generate different (and inconsistent) abstract outputs. Then, we compute the objective value of e'_{in} for each abstract output $\hat{e}'^{(i)}_{out}$—i.e., $\ell(E \cup \{(e'_{in}, \hat{e}'^{(i)}_{out})\})$, which is the sum of ✓ in the corresponding row. Finally, we obtain the score of e'_{in} by using $\mathcal{A}$ to aggregate this objective value over the abstract outputs of all sampled programs—e.g., $\mathcal{A}$ may take the maximum over these objective values.

Whereas Algorithm 2 operates over sampled rollouts $\zeta \in Z$, in this example, we operate over the "flattened" Z—i.e., the set of sampled partial programs $\mathcal{P}_Z = \{P \mid \zeta \in Z,\ P \in \zeta\}$. These two approaches are equivalent—Algorithm 2 flattens Z on line 3 to construct the abstract outputs $\hat{E}'_{out}$; also, the sum in line 5 equals (up to constants) $|\{P \in \mathcal{P}_Z \mid P \models (e'_{in}, \hat{e}'_{out})\}|$, which is exactly $\ell(E \cup \{(e'_{in}, \hat{e}'_{out})\})$ in Figure 3. Finally, we count ✓ instead of ✗ since we are estimating running time instead of pruning power (see Theorem 2.5).

IV. IMPLEMENTATION

We describe its procedure for generating candidate input examples, and the available aggregation functions $\mathcal{A}$ in FAERY.

Candidate inputs. Our algorithm assumes a given set of candidate input examples E'_{in}. FAERY generates E'_{in} using a standard mutation procedure. First, for a given kind of input data structure (e.g., a list, tree, table, etc.), we provide a DSL that encodes common operations over the data structure. For instance, for the table transformation domain, our DSL (as shown in Table III) contains operations such as deleting a

column, inserting a row, swapping the order of two rows, etc. Then, given the initial input example e'_{in}, E'_{in} is obtained by executing $\phi(e_{in})$ where ϕ is randomly sampled program from this DSL.

Aggregation function. Recall that in Algorithm 2, the SELECTQUERY subroutine relies on a function $\mathcal{A}$ to aggregate scores over the possible abstract outputs $\hat{e}'_{out}$ for each $e'_{in} \in E'_{in}$. We have implemented the following possible $\mathcal{A}$:

- The MAX strategy estimates the pruning power of a candidate input by:

$$\mathcal{A}(\hat{E}'_{out}) = - \max_{\hat{e}'_{out} \in \hat{E}'_{out}} \ell(\hat{e}'_{out}),$$
$$where \;\; \ell(\hat{e}'_{out}) = \ell(E \cup \{(e'_{in}, \hat{e}'_{out})).$$

  Intuitively, this strategy is an optimistic lower bound on the pruning power of e'_{in}—namely, the largest number of programs pruned by any possible abstract output.
- The EXPECTED strategy estimates the pruning power of a candidate input by computing the expected objective value across different abstract outputs:

$$\mathcal{A}(\hat{E}'_{out}) = - \sum_{\hat{e}'_{out} \in \hat{E}'_{out}} \frac{\ell(\hat{e}'_{out}) \cdot (\sum_{\zeta \in Z} |\zeta| - \ell(\hat{e}'_{out}))}{\sum_{\zeta \in Z} |\zeta|}.$$

  Intuitively, this strategy sums the number of programs pruned by each abstract output, weighted by the estimated probability that the queried output is consistent with the abstract output.

V. EVALUATION

We evaluated FAERY by conducting systematic experiments that are designed to answer the following research questions:

- Q1. **Scalability.** How does FAERY perform compared to state-of-the-art synthesis tools?
- Q2. **Reliability.** Can FAERY *consistently* reduce synthesis time for different initial examples?
- Q3. **Effectiveness.** How effective are the different aggregation functions described in Section IV?

A. Experimental Setup

For our core experiments, we use FAERY with the MAX strategy. We instantiate FAERY on two important domains in data science: (i) data wrangling in R, and (ii) JSON transformations using the JQ [17] library. To compare FAERY with existing tools, we adopt the original DSL used in NEO and MORPHEUS [4, 3] for domain (i), and designed a variant DSL of JQ for domain (ii). All experiments are conducted on an Intel Xeon(R) computer with an E5-2640 v3 CPU and 16G of memory, running the Ubuntu 18.04 with a timeout of 10 minutes.

State-of-the-art tools. In data wrangling domain, we compare to NEO [4], a state-of-the-art synthesis tool that is designed for this domain. To allow a fair comparison, we instantiate FAERY with the same DSL and specifications used by NEO. Furthermore, we use NEO's bigram model to prioritize the search. For the JSON transformations domain, we compare against TRINITY [13]. We also used the same DSL and abstract semantics for both tools.

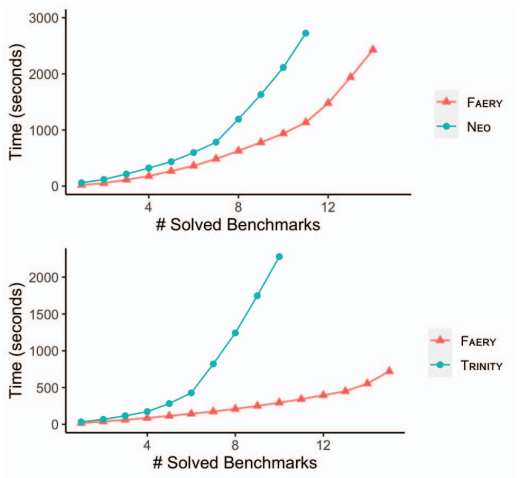

Fig. 4: Comparison between FAERY and: 1) NEO (up) on data wrangling tasks, and 2) TRINITY (down) on JSON transformation tasks; x-axis shows the number of benchmarks solved; y-axis shows the cumulative time taken.

Benchmark selection. For data wrangling, we use 15 of the most difficult benchmarks in prior work [4], where difficulty is measured in terms of NEO's synthesis time; we focus on difficult benchmarks since the goal of FAERY is to improve performance on benchmarks in the long tail in terms of scalability.

For JSON transformations, we have collected 15 challenging JSON transformation tasks from StackOverflow (i.e., a post with an IO example and a response with the desired program)—namely, tasks where the solution is a program with at least (i) 10 AST nodes, (ii) two higher-order components, and (iii) a mapping or aggregation function.

Number of interactions. We show that both data-wrangling and JSON transformation domains are highly constrained, and typically a single example is sufficient to significantly constrain the search space to just a few (and often a single) solution; the challenge is finding even a single solution. In particular, based on a study of the *full* 50 benchmarks for the data wrangling domain in NEO, there are 47 (94%) of them where the user-intended solution can be precisely quantified by the single user-provided input-output example. Similarly, for the JSON transformation domain, for 13 out of 15 (87%) benchmarks, the user-intended solution can be precisely quantified by the single user-provided input-output example. The benefit of interaction for these domains is almost entirely to improve scalability and reliability.

B. Scalability

We design automated evaluation and a user study to measure the scalability performance of FAERY. In automated evaluation, answers to user queries are directly supplied, while in the user study the tool waits for a real participant's response.

Results. Figure 4 (left) shows results for data wrangling. As we can see, FAERY significantly outperforms NEO in terms of both synthesis time and the number of benchmarks solved. In particular, FAERY solves 14 benchmarks with an average running time of 174 seconds, whereas NEO solves just 11 benchmarks with an average running time of 211 seconds.

Next, Figure 4 (right) shows results for JSON transformations. We observe an even greater speed-up in this domain: FAERY solves all 15 benchmarks with an average running time of 48 seconds, whereas TRINITY solves 11 benchmarks with an average running time of 22 seconds. Both results show that FAERY can dramatically speed up search compared to state-of-the-art synthesis techniques.

User interaction. We note that it is easy for end users to label the output examples for our selected input examples. On average, it took 10 seconds for a user to answer a query (i.e., provide the desired output for an input), which is less than 5% of the total synthesis time. See Section F for details of the user study.

C. Reliability

To evaluate the reliability of FAERY, we create variants of the existing synthesis tasks and compare the performance between different tools on these tasks. Specifically, we mutate synthesis tasks by *replacing* the initial IO example with each of the candidate IO examples produced by the MUTATE subroutine, and ask each tool to solve them. In the data wrangling (resp., JSON transformation) domain, we compare the performance of FAERY with NEO (resp., TRINITY); we also compare to an ablation "FAERY with RANDOM" that randomly selects a candidate input example to query the user.

Results. We show the performance of different tools on each benchmark across different initial IO example in Figure 5. The box captures the quartiles of the distribution in performance across different IO examples. For both domains, FAERY with MAX performs reliably on all benchmarks—in particular, its synthesis time is low in terms of both mean and variance. The baselines perform well on some benchmarks (sometimes even slightly better than FAERY with MAX), but have huge mean and/or variance on others—i.e., in z1, z5-7, and z10-13. There are a few reasons why FAERY with MAX may perform slightly worse: a) **Mutation strategy:** The candidate input examples produced by our mutation algorithm are equally effective, which can result in similar end-to-end performance for any query (e.g., z12 and z13). b) **Quality of initial examples:** Having high-quality initial examples reduces the benefit of interaction, and adding an extra example to the synthesizer may even introduce more overhead than time saved by extra pruning, thereby increasing synthesis time (e.g., z4, z8, z14 and z15). c) **Approximations:** Our MAX strategy uses approximations that rely on assumptions about the structure of the abstract search space, which may not always hold. Other strategies make different assumptions that perform better in specific cases—e.g., EXPECTED performs much better than MAX on z4 and z6 (but worse on the remaining benchmarks).

Strategy	Data Wrangling	JSON Transformation
FAERY (MAX)	99s	105s
FAERY (RANDOM)	142s	168s
NEO/TRINITY	170s	174s

TABLE I: Variability* of synthesizers.

Benchmark		MAX	EXPECTED
Data Wrangling	#solved	14/15	12/15
	avg. time	174s	228s
	avg. speed-up	2.4×	1.4×
JSON Transformation	#solved	15/15	11/15
	avg. time	48s	36s
	avg. speed-up	9.5×	8.2×

TABLE II: Effectiveness of different aggregation functions.

In addition, note that FAERY with RANDOM often outperforms NEO (e.g., r5 and z12). These results demonstrate the promise of using user interaction to reduce synthesis time; by using a more intelligent strategy, FAERY with MAX further improves performance by a significant margin. Next, in Table I, we show the variability for each strategy averaged across benchmarks in a domain (we take the square root so the units are seconds). As can be seen, FAERY with MAX is significantly more reliable than the baselines—e.g., 42% better than NEO on data wrangling and 40% better than TRINITY on JSON transformations.

Finally, Table II shows the performance of FAERY using the different aggregation strategies in Section IV—i.e., MAX and EXPECTED. Both strategies are effective. Specifically, in both domains, the MAX strategy outperforms EXPECTED favorably in terms of both running time (9.5× v.s. 8.2× speed-up in the JSON transformation client; 2.4× v.s. 1.4× in the data wrangling domain) and the number of benchmarks being solved (15 v.s. 11 benchmarks in the JSON transformation domain; 14 v.s. 12 benchmarks in the data wrangling client).

In summary, FAERY achieves high reliability compared to the baselines with a random or fixed strategies. In particular, when the initial user-provided examples are ineffective, FAERY is still efficient by leveraging the pruning power of the extra examples. While both strategies are effective, the MAX strategy outperforms EXPECTED favorably in terms of both running time and the number of benchmarks being solved.

D. Threats to Validity

Validity of user response. Even though additional user response provides more potentially useful information for problem solving, a user's familiarity towards context of the problem that she's working on still plays a key role to the performance of FAERY– i.e., mistakes or inconsistency made between the user's inputs could create additional challenges to the synthesizer. Thus, to mitigate this in the user study, each participant is asked to complete a tutorial that reinforces them about the intention and context of the problem. We elaborate more detail in Section F.

* We report the square root of the average across benchmarks.

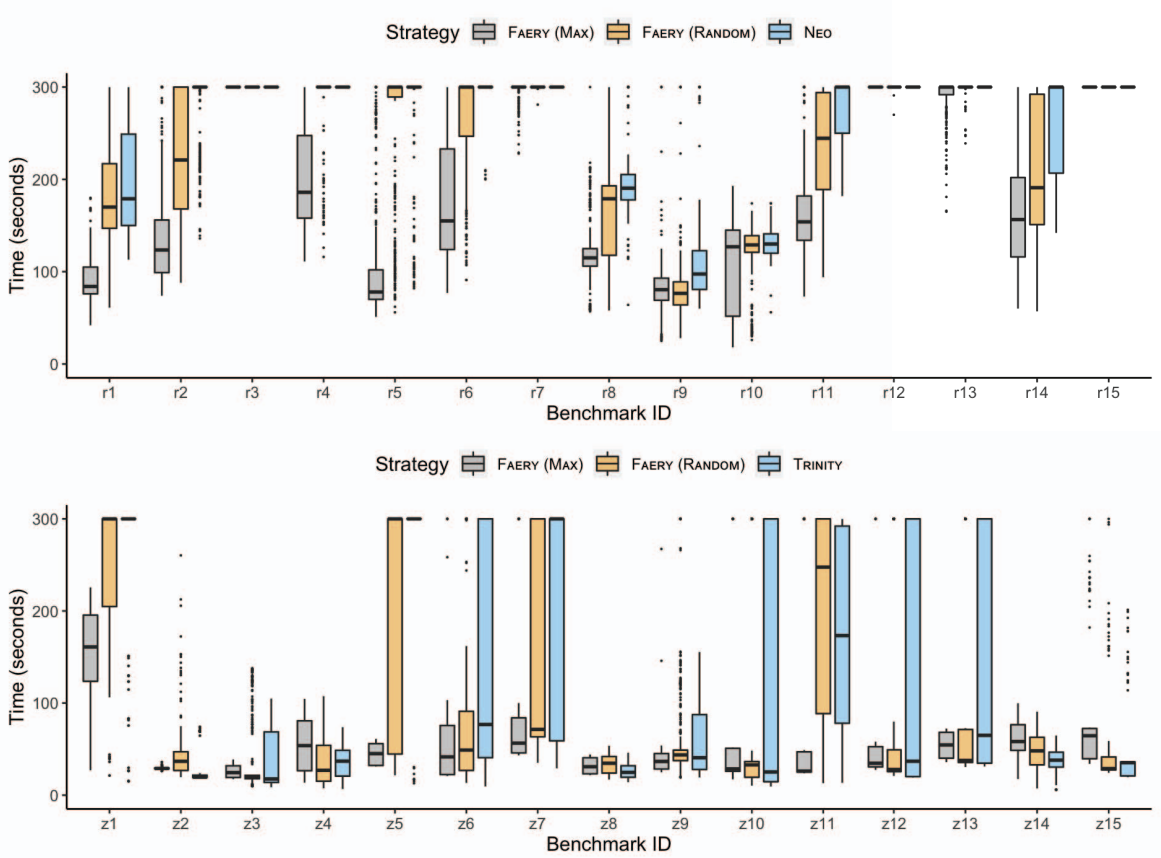

Fig. 5: Box plots measuring reliability on data wrangling benchmarks (top) and JSON transformation benchmarks (bottom). The boxes capture quartiles (omitting outliers, which are shown separately as points) across different initial IO examples for each benchmark. The black horizontal line is the mean (lower mean implies better scalability), and the height of the box captures variability (lower variability implies better reliability).

Quality of mutation. Generation of the additional candidate inputs is done by MUTATE, which is then used by SELECT-QUERY for user interaction. Effectiveness of the candidates for improving the synthesis performance given valid user response may still drop – that is, the quality of MUTATE is nontrivial to quantify and optimize. To mitigate this, we increase the size of the candidate pool generated using MUTATE so as to reduce the chance of queries that are ineffective or difficult to answer.

VI. Related work

A. Program Synthesis

There has been significant interest in automatically synthesizing programs from high-level user intent [18, 9, 19, 6, 20, 15, 21, 22]. Techniques geared towards programmers often utilize complex specifications, such as program sketches [18, 23, 24, 25] or types [19, 26], possibly in conjunction with test cases [27, 28] or logical specifications [23, 20]. In contrast, techniques geared towards end-users (i.e., non-experts) rely on IO examples [9, 21, 5, 29], natural language [30, 31, 32, 33], or both [34, 35]. While we have focused on IO examples, the high-level ideas can in principle, be applied to a broad classes of specifications, as long as some notion of refinement of the specification is available (analogous to querying additional IO examples).

B. Deduction-Based Pruning

We build on a line of work using deduction to prune the search space [19, 3, 4, 13, 2, 15]—e.g., using types and type-directed reasoning to prune infeasible partial programs [19, 15, 27, 36, 37], or using lightweight program analysis to do so [2, 3, 4, 13]. Concretely, BLAZE [2] uses abstract interpretation to build a compact version space representation capturing the space of all feasible programs; MORPHEUS [3], NEO [4] and TRINITY [13] use logical specifications of DSL constructs to derive feasibility conditions that are checked with an SMT solver; and SCYTHE [5] and VISER [38] use deductive reasoning to compute approximate results of partial programs to check feasibility. Our approach queries the user on the input example that maximizes the pruning power of the deduction engine. The deduction engine FAERY uses is similar

to NEO [4]; however, it can in principle be used with other deductive reasoning techniques.

C. Interactive Synthesis

Existing interactive synthesis systems focus on obtaining extra examples to disambiguate user intent[12, 11, 39, 40, 10, 41, 42, 43, 44, 45, 46]—e.g., Mayer et al. [12] and Wang et al. [11] randomly mutate existing inputs until a distinguishing input is found, and query the user on this example. Laich et al. [39] generates additional examples to address ambiguity issues, and Pu et al. [40] develops an approach to select small and representative subsets of examples. Ji et al. [10] improves these approaches by selecting the *optimal input* that minimizes the number of user queries needed to resolve ambiguity. In contrast, we are interested in problems where interaction is not needed to disambiguate, but to speed up synthesis. Importantly, techniques for tackling the former problem fundamentally cannot address the latter. In particular, they all rely on being able to sample multiple correct (concrete) programs. However, if we can identify even a single correct concrete program, then we have already solved our problem. In addition, we provide a theoretical framework to quantify the running time of deduction-guided synthesis.

VII. CONCLUSION

We have proposed a novel programming-by-example framework that leverages user interaction to improve both scalability and reliability. While we have focused on leveraging our theoretical analysis to facilitate user interaction, we believe our techniques have the potential to help improve our understanding the underlying synthesis search space. As we have demonstrated in our evaluation, the reliability of synthesis tools remains a key challenge, and our approach serves as a first step towards to quantifying the performance properties of synthesizers.

ACKNOWLEDGMENT

We thank the anonymous reviewers for the helpful feedback. This material is based upon work partially supported by a Google Faculty Research award, Ethereum Foundation academic award, NSF 1908494 and DARPA N66001-22-2-4037. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the funding source.

REFERENCES

[1] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2006, pp. 404–415.

[2] X. Wang, I. Dillig, and R. Singh, "Program synthesis using abstraction refinement," in *Proc. Symposium on Principles of Programming Languages*. ACM, 2018, pp. 63:1–63:30.

[3] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," in *Proc. Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 422–436.

[4] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," in *Proc. Conference on Programming Language Design and Implementation*, 2018, pp. 420–435.

[5] C. Wang, A. Cheung, and R. Bodik, "Synthesizing highly expressive sql queries from input-output examples," in *Proc. Conference on Programming Language Design and Implementation*. ACM, 2017, pp. 452–466.

[6] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," in *Proc. International Conference on Learning Representations*. OpenReview, 2017.

[7] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica, "Autopandas: neural-backed generators for program synthesis," *PACMPL*, vol. 3, no. OOPSLA, pp. 168:1–168:27, 2019.

[8] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," in *ICLR*, 2019.

[9] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proc. Symposium on Principles of Programming Languages*. ACM, 2011, pp. 317–330.

[10] R. Ji, J. Liang, Y. Xiong, L. Zhang, and Z. Hu, "Question selection for interactive program synthesis," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, 2020, pp. 1143–1158.

[11] C. Wang, A. Cheung, and R. Bodík, "Interactive query synthesis from input-output examples," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017, pp. 1631–1634.

[12] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. G. Zorn, and S. Gulwani, "User interaction models for disambiguation in programming by example," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, 2015, pp. 291–301.

[13] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig, "Trinity: an extensible synthesis framework for data science," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 1914–1917, 2019.

[14] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-guided synthesis*. IEEE, 2013.

[15] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output exam-

ples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 229–239.

[16] Y. Chen, C. Wang, O. Bastani, I. Dillig, and Y. Feng, "Program synthesis using deduction-guided reinforcement learning," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 587–610.

[17] S. Dolan, "JQ: a lightweight and flexible command-line json processor," http://https://stedolan.github.io/jq/, 2018.

[18] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, 2006, pp. 404–415.

[19] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program synthesis from polymorphic refinement types," *Proc. Conference on Programming Language Design and Implementation*, pp. 522–538, 2016.

[20] J. Bornholt and E. Torlak, "Synthesizing memory models from framework sketches and litmus tests," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, 2017, pp. 467–481.

[21] O. Polozov and S. Gulwani, "Flashmeta: a framework for inductive program synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, 2015, pp. 107–126.

[22] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. International Conference on Software Engineering*. ACM/IEEE, 2010, pp. 215–224.

[23] E. Torlak and R. Bodík, "A lightweight symbolic virtual machine for solver-aided host languages," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 530–541.

[24] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum, "Learning to infer graphics programs from hand-drawn images," in *Advances in neural information processing systems*, 2018, pp. 6059–6068.

[25] M. I. Nye, L. B. Hewitt, J. B. Tenenbaum, and A. Solar-Lezama, "Learning to infer program sketches," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 2019, pp. 4861–4870. [Online]. Available: http://proceedings.mlr.press/v97/nye19a.html

[26] A. Miltner, S. Maina, K. Fisher, B. C. Pierce, D. Walker, and S. Zdancewic, "Synthesizing symmetric lenses," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–28, 2019.

[27] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. Reps, "Component-Based Synthesis for Complex APIs," in *Proc. Symposium on Principles of Programming Languages*. ACM, 2017, pp. 599–612.

[28] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[29] K. Ellis and S. Gulwani, "Learning to learn programs from examples: Going beyond program structure," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 1638–1645. [Online]. Available: https://doi.org/10.24963/ijcai.2017/227

[30] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "SQLizer: Query Synthesis from Natural Language," in *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2017, pp. 63:1–63:26.

[31] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. Radev, "Syntaxsqlnet: Syntax tree networks for complex and cross-domain text-to-sql task," in *Proceedings of EMNLP*. Association for Computational Linguistics, 2018.

[32] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, 2018, pp. 1643–1652. [Online]. Available: https://www.aclweb.org/anthology/D18-1192/

[33] R. Shin, M. Allamanis, M. Brockschmidt, and O. Polozov, "Program synthesis and semantic parsing with learned code idioms," in *NeurIPS*, 2019.

[34] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," 2019.

[35] Y. Chen, R. Martins, and Y. Feng, "Maximal multi-layer specification synthesis," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 2019, pp. 602–612.

[36] P. Osera and S. Zdancewic, "Type-and-example-directed program synthesis," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 619–630.

[37] J. Frankle, P. Osera, D. Walker, and S. Zdancewic, "Example-directed synthesis: a type-theoretic interpretation," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 802–815.

[38] C. Wang, Y. Feng, R. Bodík, A. Cheung, and I. Dillig, "Visualization by example," *PACMPL*, vol. 4, no. POPL, pp. 49:1–49:28, 2020. [Online]. Available:

https://doi.org/10.1145/3371117

[39] L. Laich, P. Bielik, and M. Vechev, "Guiding program synthesis by learning to generate examples," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=BJl07ySKvS

[40] Y. Pu, Z. Miranda, A. Solar-Lezama, and L. Kaelbling, "Selecting representative examples for program synthesis," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 4161–4170. [Online]. Available: http://proceedings.mlr.press/v80/pu18b.html

[41] D. Drachsler-Cohen, S. Shoham, and E. Yahav, "Synthesis with abstract examples," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 254–278.

[42] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. D. Millstein, "Flashprofile: a framework for synthesizing data profiles," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 150:1–150:28, 2018.

[43] O. Bastani, X. Zhang, and A. Solar-Lezama, "Synthesizing queries via interactive sketching," *arXiv preprint arXiv:1912.12659*, 2019.

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

[45] Z. Zhou, M. T. Tang, Q. Pan, S. Tan, X. Wang, and T. Zhang, "Intent: Interactive tensor transformation synthesis," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3526113.3545653

[46] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, "Interactive program synthesis by augmented examples," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 627–648. [Online]. Available: https://doi.org/10.1145/3379337.3415900