Visualization Question Answering using Introspective Program Synthesis

Yanju Chen
Department of Computer Science
University of California Santa Barbara
USA
yanju@cs.ucsb.edu

Xifeng Yan
Department of Computer Science
University of California Santa Barbara
USA
xyan@cs.ucsb.edu

Yu Feng
Department of Computer Science
University of California Santa Barbara
USA
yufeng@cs.ucsb.edu

Abstract

While data visualization plays a crucial role in gaining insights from data, generating answers over complex visualizations from natural language questions is far from an easy task. Mainstream approaches reduce data visualization queries to a semantic parsing problem, which either relies on expensive-to-annotate supervised training data that pairs natural language questions with logical forms, or weakly supervised models that incorporate a larger corpus but fail on long-tailed queries without explanations. This paper aims to answer data visualization queries by automatically synthesizing the corresponding program from natural language. At the core of our technique is an abstract synthesis engine that is bootstrapped by an off-the-shelf weakly supervised model and an optimal synthesis algorithm guided by *triangle alignment constraints*, which represent *consistency* among natural language, visualization, and the synthesized program.

Starting with a few tentative answers obtained from an off-the-shelf statistical model, our approach first involves an *abstract synthesizer* that generates a set of sketches that are consistent with the answers. Then we design an instance of optimal synthesis to complete one of the candidate sketches by satisfying common type constraints and maximizing the consistency among three parties, i.e., natural language, the visualization, and the candidate program.

We implement the proposed idea in a system called Poe that can answer visualization queries from natural language. Our method is fully automated and does not require users to know the underlying schema of the visualizations. We evaluate Poe on 629 visualization queries and our experiment shows that Poe outperforms state-of-the-arts by improving the accuracy from 44% to 59%.

This work is licensed under a Creative Commons Attribution 4.0 International License.

PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523709

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Computing methodologies** → **Natural language processing**; **Machine learning**.

Keywords: Program Synthesis, Visualization, Machine Learning, Natural Language Processing

ACM Reference Format:
Yanju Chen, Xifeng Yan, and Yu Feng. 2022. Visualization Question Answering using Introspective Program Synthesis. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523709

1 Introduction

Due to the prevalence of non-trivial visualization tasks across different application domains, recent years have seen a growing number of libraries that aim to automate complex visualization tasks. Despite all these efforts, data visualization still remains a daunting task that requires considerable expertise.

As many end-users typically lack the expertise to write complex queries in declarative query languages such as SQL or R programs, techniques that can answer visualization queries from natural language (NL) descriptions are more compelling. However, because natural language is inherently ambiguous, mainstream NL-based techniques try to achieve high precision by training the system on a specific semantic parser [6] where the question is translated to a logical form that can be executed against the visualization to retrieve the correct denotation. Unfortunately, semantic parsers heavily rely on supervised training data that pairs natural language questions with logical forms, but such data is very expensive to annotate. Although recent state-of-the-arts [20] slightly mitigate this challenge through weak supervision without explicitly annotating data with logical forms, their performance is far from satisfactory [20, 22] due to the quality and quantity of the training data required to infer the hidden logical connections for deriving the answers.

In this paper, we provide an *introspective program synthesis* technique and its implementation in a tool called Poe, for synthesizing data visualization queries from natural language. Our key insight is based on a synergistic integration of statistical model and logic-based reasoning shown in Figure. 1. Specifically, Poe starts with answers from an off-the-shelf

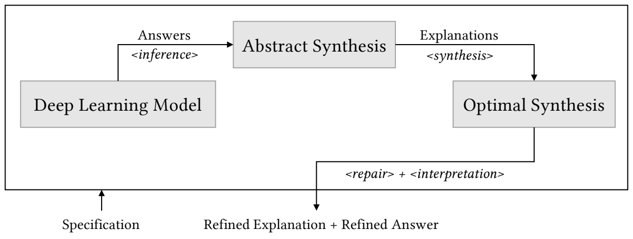


Figure 1. Framework overview.

statistic model that is trained through weak supervision. Since such a model only relies on pairs of question-answer instead of explicit logical programs, it significantly reduces the effort of labeling data thus achieves better performance through a large corpus [20]. However, in the case of long-tailed queries, the statistical model may still generate wrong answers. This is where our key insight comes from: even though the statistical model generates a wrong answer that is derived from a sequence of hidden inference steps represented by neural network, part of the hidden steps may still be sensible since they are learnt from a large corpus. But we can not access the hidden inference steps from the neural network since it is trained directly from question-answer pairs. To get an *interpretable explanation* that deciphers the answer of a statistical model, we leverage a synthesis procedure to generate programs that are consistent with the specification, which contains a visualization query and its answer. Because the original answer may be wrong, the generated programs may all be problematic. Here, each program can be viewed as an explanation for the decision, which contains *partial correct* derivations to the correct answer. After that, Poe further turns this into an *optimal synthesis* problem whose goal is to pick a candidate program and refine it into a concrete program that is likely to be correct.

There are two caveats we need to conquer in this project. First, for each candidate answer proposed by the statistic model, there could be multiple programs that are consistent with the specification and generating each program is slow since it has to solve a non-trivial synthesis problem. Second, even with a set of programs as the explanations of the answer, we still need to define an objective function that guides the optimal synthesis to obtain the desired solution.

To address the first caveat, we design an *abstract synthesizer* whose job is to generate the *most general partial programs* that are consistent with the specification. Here, we prefer partial programs that are most general because 1) they are faster to find, and 2) they offer a *compact representation* of the explanation (i.e., search space). To mitigate the second caveat, we leverage a *multi-modal optimal synthesis procedure* whose objective function is to encode fine-grained semantic constraints that are difficult to learn by off-the-shelf statistical models. In particular, Poe encodes 1) a novel *triangle alignment* constraint that denote *semantic consistency* among three parties, namely, natural language, visualizations, and candidate programs; 2) well-typed constraints that are enforced by the semantics of the DSL.

To evaluate the effectiveness of our technique, we evaluate Poe on 629 visualization benchmarks and compare it against VisQA [22], the state-of-the-art synthesizer for visualization queries. Our experiment shows that Poe outperforms VisQA by improving the accuracy from 44% to 59%. Our ablation study clearly demonstrates the benefits of our abstract synthesizer and optimal synthesis using triangle alignments.

To summarize, this paper makes the following key contributions:

- We identify and present a new type of program synthesis problem in visualization question answering, where a deep learning model's (potentially noisy) output is used as specification to synthesize programs that explain the model's behavior, which is dubbed as *introspective program synthesis*.
- We describe an abstract program synthesis technique for quickly inducing the search space given *noisy* specifications from a deep learning model's output.
- We describe an optimal program synthesis technique for finding programs that best match the consistency constraints implied between natural language questions and visualizations.
- We implement our approach in an end-to-end system called Poe and evaluate it on 629 visualization question answering tasks of different types. In particular, we show that our approach improves the state-of-the-art performance from 44% to 59%.

2 Overview

In this section, we give an overview of our approach with the aid of a simple motivating example.

2.1 A Motivating Example

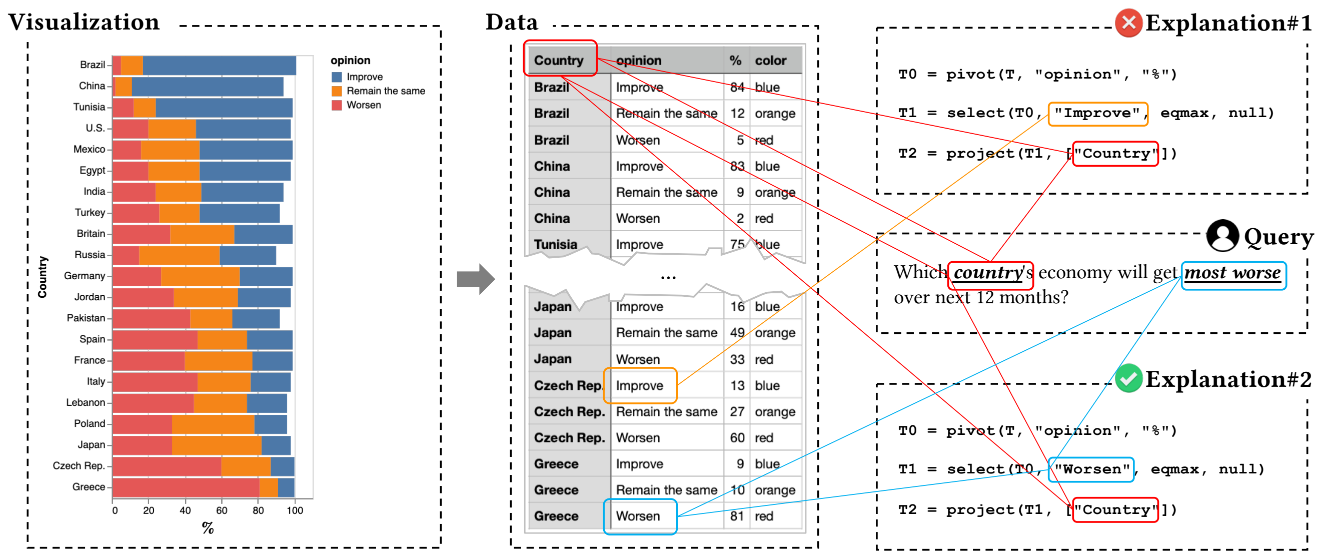
Figure. 2 (left) shows a stacked bar chart that represents the opinions for future economic growth for different countries. Here, Alice describes her query in natural language:

> *"Which country's economy will get most worse over next 12 months?"*

By reading the visualization on the length of the *red* bar for every country, human beings can locate the correct answer: *"Greece"*, because it has the longest bar that represents the opinion of *"Worsen"*, which corresponds to the keyword *"most worse"* from the query.

To automate data visualization tasks, weakly-supervised approaches [20] employ neural programming that mimics the above procedure by *directly* estimating the probability of each potential answer extracted from the visualization. For example, a typical output ranking (by probability) from such models would look like:

(0.78, Brazil), (0.67, Japan), (0.55, Greece), ...



Country	opinion	%	color
Brazil	Improve	84	blue
Brazil	Remain the same	12	orange
Brazil	Worsen	5	red
China	Improve	83	blue
China	Remain the same	9	orange
China	Worsen	2	red
Tunisia	Improve	75	blue
Japan	Improve	16	blue
Japan	Remain the same	49	orange
Japan	Worsen	33	red
Czech Rep	Improve	13	blue
Czech Rep.	Remain the same	27	orange
Czech Rep.	Worsen	60	red
Greece	Improve	9	blue
Greece	Remain the same	10	orange
Greece	Worsen	81	red

```
T0 = pivot(T, "opinion", "%")
T1 = select(T0, "Improve", eqmax, null)
T2 = project(T1, ["Country"])
```

Which *country's* economy will get *most worse* over next 12 months?

```
T0 = pivot(T, "opinion", "%")
T1 = select(T0, "Worsen", eqmax, null)
T2 = project(T1, ["Country"])
```

Figure 2. A motivating example on data of opinions for future economic growth for different countries. Left: A visualization of stacked bar chart for illustrating the data distribution; Middle: The corresponding table format of the data; Right: Example checking semantic consistency between three parties: data, query and explanation. Explanation#1 doesn't fit since no keyword in the query shares similar meaning with *Improve* in the data and `Improve` in the explanation; Explanation#2 satisfies semantic consistency.

where each tuple is composed by a candidate answer and its corresponding probability estimation. Compared to approaches based on semantic parsing that require additional labeling of intermediate logical forms, weakly-supervised approaches save the efforts of manual labeling by skipping the logical forms and moving directly from query to answer, thus benefiting from a larger source of available training data. However, it becomes non-trivial to track and fix problematic answers proposed by these models, since weakly-supervised approaches do not utilize intermediate logical forms that give hints about the implicit reasoning process. For example, according to the above output ranking, the correct solution *"Greece"* has a lower probability than *"Brazil"*. However, because the model does not generate logical forms to *explain* the answers, it is difficult to figure out which one is the correct answer.

To address this, Poe employs a two-staged program synthesis procedure to refine the candidate answers immediately proposed from weakly-supervised models. First, for candidate answers, Poe generates potential explanations (i.e., abstract programs) using an abstract program synthesis algorithm. Then, Poe tries to refine the explanations based on information from the data and user-provided query by optimal synthesis techniques. Finally, Poe proposes the most promising candidate answer based on the newly refined ranking.

$$
\begin{array}{lll}
\langle \mathit{Table} \rangle & ::= & \text{project}(\ \langle \mathit{Table} \rangle, \langle \mathit{ColList} \rangle\ ) \\
 & \mid & \text{select}(\ \langle \mathit{Table} \rangle, \langle \mathit{BoolOp} \rangle, \langle \mathit{ColInt} \rangle, \langle \mathit{ConstVal} \rangle\ ) \\
 & \mid & \text{pivot}(\ \langle \mathit{Table} \rangle, \langle \mathit{ColInt} \rangle, \langle \mathit{ColInt} \rangle\ ) \\
 & \mid & \text{aggregate}(\ \langle \mathit{Table} \rangle, \langle \mathit{ColList} \rangle, \langle \mathit{AggrOp} \rangle, \langle \mathit{ColInt} \rangle\ ) \\
\langle \mathit{AggrOp} \rangle & ::= & \text{count} \mid \text{min} \mid \text{max} \mid \text{sum} \mid \text{mean} \\
\langle \mathit{BoolOp} \rangle & ::= & < \mid <= \mid == \mid >= \mid > \mid != \mid \text{eqmax} \mid \text{eqmin}
\end{array}
$$
$$\langle \mathit{Table} \rangle \in \textbf{tables},\quad \langle \mathit{ConstVal} \rangle \in \textbf{constants}$$
$$\langle \mathit{ColInt} \rangle \in \textbf{columns},\quad \langle \mathit{ColList} \rangle \in \textbf{columns}^n$$

Figure 3. Syntax of a toy DSL for data wrangling.

2.2 Explanation Generation

To reason about the visualization, without loss of information from data, Poe applies a visualization-to-table conversion procedure similar to previous work [22] to obtain a compact representation, as shown in Figure. 2 (middle). To explain the candidate answers using program synthesis, we first introduce a simple domain-specific language (DSL) for common data wrangling tasks. As shown in Figure. 3, the DSL supports a subset of relational algebra such as projection (`project`) and selection (`select`) with aggregation (`aggregate`), as well as pivoting (`pivot`) from typical data wrangling tasks.

The abstract synthesis engine of Poe can explain the candidate answers by looking for DSL programs that generate the corresponding answers. In particular, for a given table T (converted from its visualization) and the proposed top-k candidate answers $A_0, A_1, ..., A_k$, Poe treats them as multiple programming-by-example (PBE) problems where the input example is T and the output example is A_i, one for each candidate answer as shown below:

$$(T, A_0), (T, A_1), (T, A_2), \ldots$$

where A_0 =*"Brazil"*, A_1 =*"Czech Rep."*, A_2 =*"Greece"*, etc., and synthesizes their corresponding DSL programs. For example, for A_0 =*"Brazil"*, there can be multiple explanations:

```
project(select(T, "%", ==, 84), ["Country"])
project(select(pivot(
  T, "opinion", "%"), "Improve", eqmax, null), ["Country"])
...
```

and for A_2 =*"Greece"* the explanations would look like:

```
project(select(pivot(
  T, "opinion", "%"), "Worsen", eqmax, null), ["Country"])
...
```

Instead of directly synthesizing the above concrete programs, which may not be scalable in practice, Poe synthesizes *abstract programs* that are *consistent* with their corresponding IO examples. So the explanations for A_0 =*"Brazil"* would look like:

```
project(select(T, ⋄, ⋄, ⋄), ⋄)
project(select(pivot(T, ⋄, ⋄), ⋄, ⋄, ⋄), ⋄)
...
```

and similar to A_2 =*"Greece"*:

```
project(select(pivot(T, ⋄, ⋄), ⋄, ⋄, ⋄), ⋄)
...
```

where ⋄ denotes a hole in the program *yet to be determined.* Such an abstract program can be further refined to concrete programs by filling up the holes. Thus, each of them represents a broader search space of *concrete programs.*

Strategically, since the program

```
project(select(pivot(T, ⋄, ⋄), ⋄, ⋄, ⋄), ⋄)
```

satisfies at least 2 of the examples, i.e., (T, A_0) (where A_0 =*"Brazil"* which corresponds to the country with the highest *"Improve"* opinion) and (T, A_2) (where A_2 =*"Greece"* which corresponds to the country with highest *"Worsen"* opinion), it's included as one of the potential abstract programs. Besides, Poe seeks to expand the bag of such abstract programs. For example, the following program

```
project(select(T, ⋄, ⋄, ⋄), ⋄)
```

also satisfies multiple examples (e.g., (T, A_0) and (T, A_1) so it's also included.

As a result, Poe's abstract synthesis procedure constructs a bag of abstract programs that satisfy the top-k examples:

```
project(select(pivot(T, ⋄, ⋄), ⋄, ⋄, ⋄), ⋄)
project(select(T, ⋄, ⋄, ⋄), ⋄)
...
```

and provides it to the optimal synthesis for further refinement.

2.3 Answer Refinement

Given the list of program sketches above, Poe's optimal synthesis engine fills in the holes by combination of type-directed synthesis and multi-modal information from the original data and query. In particular, Poe infers constraints from the original data and query and encode them as objectives that guide the optimal synthesis procedure.

Note that the query from the user has two keywords highlighted automatically[1], i.e., *"country"* and *"most worse"*. Poe composes constraints from different guiding principles in practice. For example, semantic consistency should be maintained among three parties, namely data, query and explanation, which we denote by *triangle alignment.* In particular for the keyword *"country"* in the query, triangle alignment produces constraints that ensure the existence of table contents that have similar meanings with *"country"*, as well as existence of similar DSL constructs in the explanation programs.

Figure. 2 (right) depicts the meaning of semantic consistency via triangle alignment. For a concrete program refined from the bag of abstract programs such as:

```
project(select(pivot(
  T, "opinion", "%"), "Improve", eqmax, null), ["Country"])
```

we can find `Country` as an argument provided to `project` and *"Country"* as a column name in the original table. However, the semantic consistency for *"most worse"* is broken since we cannot find any language construct in the program that is similar to it, even though *"Worsen"* as an opinion in the original table builds up the similarity connection between the data and the query. If we switch the language construct that causes the inconsistency from *"Improve"* to *"Worsen"*, the resulting program:

```
project(select(pivot(
  T, "opinion", "%"), "Worsen", eqmax, null), ["Country"])
```

now satisfies the semantic consistency, where `Worsen` from the program now connects with *"Worsen"* in the query and *Worsen* in the data. Actually, this turns out to be the exact program that best executes the user intent and generates the desired answer *"Greece"*.

Besides triangle alignment, Poe also encodes other guiding principles as soft constraints into an optimal synthesis problem and generates a ranking list of preferences of concrete programs in accordance to how well they fit into different constraints. Eventually, Poe executes the top-ranked program and returns the refined answer.

3 Preliminaries and Problem Statement

In this section, we first provide some background that will be used throughout the paper. After that, we describe the architecture of our introspective synthesis algorithm and explain

[1]Keyword discovery can be approached by a template-based method or by data-driven methods (e.g., TFIDF weighting).

each of its components in detail. However, because both the abstract synthesis and optimal refinement are the main contributions of this paper, we defer a detailed discussion to Section. 4 and Section. 5, respectively.

3.1 Preliminaries

DSL. We assume a domain-specific language L specified as a context-free grammar $L = (V, \Sigma, R, S)$, where V, Σ denote non-terminals and terminals respectively, R is a set of productions, and S is the start symbol.

Partial Program. A *partial program (or abstract program)* 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). We refer to any non-terminal in P as a hole $\diamond$, and we say that P is *complete* if it does not contain any holes.

Given a partial program P containing a hole $\diamond$, we can fill this hole by replacing $\diamond$ with the right-hand-side of any grammar production r of the form $\diamond \rightarrow e$. We use the notation $P \overset{r}{\Rightarrow} P'$ to indicate that P' is the partial program [2] obtained by replacing the first occurrence of $\diamond$ with the right-hand-side of r, and we write $\textsc{Fill}(P, r) = P'$ whenever $P \overset{r}{\Rightarrow} P'$.

Example 1. Consider the following partial program P:

```
project(⋄, ⋄)
```

and production $r \equiv \diamond \rightarrow$ `select(⋄,⋄,⋄,⋄)`. In this case, $\textsc{Fill}(P, r)$ yields the following partial program P':

```
project(select(⋄, ⋄, ⋄, ⋄), ⋄)
```

Deduction Engine. Motivated by prior work [11, 15, 16] in deductive synthesis, we assume access to a *deduction engine* that can determine whether a partial program P is *feasible* with respect to specification ϕ. To make this more precise, we introduce the following notion of feasibility.

Definition 1 (Feasible Partial Program)**.** Given a specification ϕ and language $L = (V, \Sigma, R, S)$, a partial program P is said to be *feasible* with respect to ϕ if there exists any complete program P' such that $P \overset{*}{\Rightarrow} P'$ and $P' \models \phi$.

In other words, a feasible partial program can be refined into a complete program that satisfies the specification. We assume that our deduction engine over-approximates feasibility through *abstract semantics*. That is, if P is feasible with respect to specification ϕ, then the feasibility check should report that P is feasible but not necessarily vice versa. Note that almost all deduction techniques used in the program synthesis literature satisfy this assumption [15–17, 23, 43].

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

$$e_{in} : [74, 39, 40, 53, 89, 10] \mapsto e_{out} : [78, 80, 106]$$

[2]We also call P' as the refinement of P.

We use the length of the list as the abstract domain [15]. Thus, the partial program P: `reverse(map(`e_{in}`, ⋄))` is infeasible (i.e., $P \not\models e$). In other words, the program won't satisfy the given IO example, no matter how we fill hole $\diamond$, because:

- The `map` construct takes as input a function (yet to be determined by the synthesizer) and applies it over every element of e_{in}, which yields an output list with equal length to that of the input list e_{in}.
- The `reverse` construct reverses the order of elements of its input, which makes no changes to its length; thus, the output list has the same length with the input list.
- Since the output returned by `reverse` does not have the same length as the desired output e_{out}, we derive an inconsistency, i.e., $size(e_{in}) == size(e_{out}) \wedge size(e_{in}) == 6 \wedge size(e_{out}) == 3$ is UNSAT.

Statistical Model. We consider a *weakly supervised statistical model* π [20] used to prioritize the search order. Given a visualization I and its query Q, the model directly assigns probabilities $\pi(A|I, Q)$ to every candidate answer $A \in \mathcal{A}$.

Rendering Visualization as Table. For simplicity of the presentation, we will represent a visualization by its equivalent table format, which can be manipulated by existing DSLs for data wrangling or relational algebra. In particular, given a visualization I, we leverage an off-the-shelf procedure [22] to convert I into its tabular format. Please refer to Section. 6 for more details.

3.2 Introspective Program Synthesis

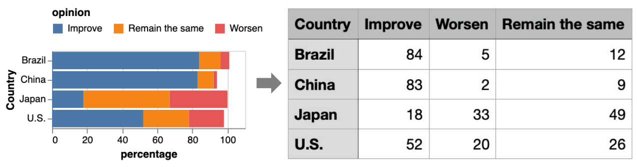
In this section, we state the problem of introspective program synthesis, as well as an overview of our proposed approach. At a higher level, our approach aims at boosting the performance of deep learning models in visualization question answering by explaining their predictions using programs and performing consistency refinements over the explanations, where we use *explanations*, *partial programs*, or *abstract programs* interchangeably. Because mainstream weakly supervised models that directly predict answers rather than generating intermediate logical forms, it is non-trivial for human beings to understand how the decisions are made and provide potential improvements. Our approach automates such a task by synthesizing and refining the answers using program synthesis. This makes our problem different from a typical PBE setting, where our specification is *noisy* in that 1) *not* all the predictions are correct, and 2) predictions may conflict with each other.

Example 3. Figure. 4 (right) shows a visualization query, where the user asks:

"Which country has highest Improve value?"

which expects the ground truth reasoning process to be similar to:

```
project(aggregate(I, null, max, ⋄), ["Country"])
```



Country	Improve	Worsen	Remain the same
Brazil	84	5	12
China	83	2	9
Japan	18	33	49
U.S.	52	20	26

Figure 4. Example tables showing how one can derive similar programs to get conflicting outputs.

where different hole fillings for ⋄ will result in different answers, namely "Brazil" (when ⋄="Improve") or "Japan" (when ⋄="Worsen" or ⋄="Remain the same").

Introspective Program Synthesis. Given 1) a visualization question answering task $\mathcal{T} = (I, Q)$ where I is the visualization and Q is the question in English, 2) a domain-specific language $L = (V, \Sigma, R, S)$, and 3) a *weakly supervised* deep learning model π that predicts top-k answers $\mathcal{A} = \pi(I, Q)$, the goal of *introspective program synthesis* is to find a *complete* program P such that $S \overset{*}{\Rightarrow} P$ and P optimizes the following objectives O:

$$\begin{aligned} P^* &= \arg\max_P J_{\mathcal{T},\pi}(P) \\ &= \arg\max_P \sum_{o \in O} \theta_o \cdot o(I, Q, \mathcal{A}, P), \end{aligned}$$

where P^* is the optimal program, J is a cumulative term of weighted objectives $o \in O$.

In particular, we leverage objectives O to solve a *multi-model* synthesis problem where O encodes 1) consistency properties among three parties, namely, the visualization, the question, and the program, and 2) *naturalness* of the program.

Key Insight. Given a weakly-supervised deep learning model π trained from a large corpus, Poe starts from the top-k answers of π. Our observation on many deep learning models indicates that, even though the model's *top* predictions may look different and sometimes may not even contain the correct answer, they share inherent semantics through *implicit reasoning processes*, which establish certain confidence drew from the training data. Therefore, our key insight is to unravel the implicit reasoning process by decompiling the answers of π while resisting fine-grained details that are error-prone due the limitation of noisy data.

Example 4. As shown in Figure. 4, given an question:

"Which country has highest Improve value?"

according to the above key insight, the following ordered predictions will be proposed by an off-the-shelf deep learning model [20]:

"Brazil", "Japan", "China", "U.S.", ...

since the first three answers can be explained by the following partial program:

```
project(aggregate(I, null, ⋄, ⋄), ["Country"])
```

while the answer of "U.S." can not be obtained because none of its values of the three opinions aligns with the maximum or minimum value which the program is able find. Thus, "Brazil", "Japan" and "China" share some inherent similarity from the perspective of how they are reasoned, even though they look unrelated on the surface.

System Overview. Figure. 5 shows the system workflow of Poe. Specifically, given a DSL $\mathcal{L}$, a visualization I, and a question Q, Poe first collects the top-k answers by querying the deep learning model π with the visualization task. Due to the noisiness of the answers, they will be sent to the *abstract synthesis* module to interpret the implicit reasoning process behind the answers.

Abstract Synthesis. Given the top-k noisy answers from the deep learning model as well as a DSL $\mathcal{L}$ for generating visualization query programs, the *abstract synthesis* module performs a *relaxed* version of deduction over the noisy answers to quickly converge to a *roughly* feasible search space, which is represented by a set of partial/abstract programs $\mathcal{P}$. We defer a detailed discussion of *abstract synthesis* to Section. 4.

Optimal Refinement. Since each abstract program $P \in \mathcal{P}$ can not be concretely executed to obtain the answer, Poe further invokes the *optimal refinement* procedure to generate a concrete program. In particular, the optimal refinement module is an instance of optimal synthesis whose goal is to optimize several objectives ranging over consistency among multiple parties as well as perplexity of the programs. Finally, the module will interpret the optimized program and return the final answer. We defer a detailed discussion of *optimal refinement* to Section. 5.

4 Abstract Program Synthesis with Noisy Specification

In this section, we describe a novel abstract synthesis algorithm that can efficiently quantify the relevant search space given noisy specification from the deep learning model.

Intuition. Due to the uncertainty of an off-the-shelf deep learning model, it may produce noisy answers that fail to capture the user intent. Therefore, before we generate the precise answer, we first need to efficiently quantify relevant search space that *explains* the outputs from the statistical model. However, this is quite challenging. As shown in Figure 6, given a set of input-output examples E, a naive way (at the left) is to generate a coarse-grained abstract program ⋄ that is consistent with all input-output examples. However, this option is useless because the search space also includes a huge amount of undesired programs. On the other extreme at the right, we can also perform fine-grained synthesis by synthesizing a concrete program P per each input-output

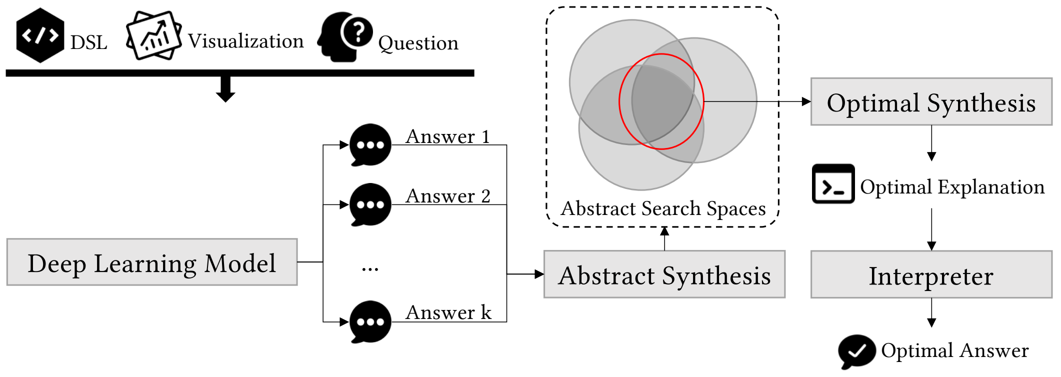


Figure 5. System workflow in Poe.

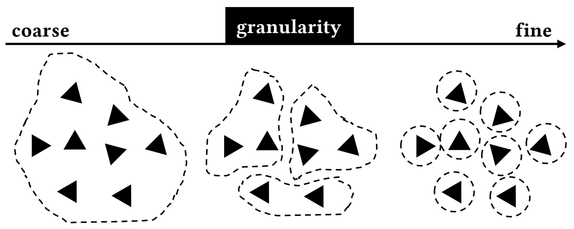


Figure 6. Different granularities that affect the algorithm search space. An input-output pair is denoted by a triangle.

example $e \in E$. However, the fine-grained option has at least two drawbacks: first, it requires invoking multiple instances of PBE (programming-by-example) tasks, which may not be feasible for the end user. Second, such a fine-grained option may also lead to overfitting, especially if none of the input-output examples matches the user intent.

Our Solution. Our goal is to compute a set of abstract programs that achieve a good balance between generality and specificness (The middle one in Figure. 6). In other words, our abstract programs should be relatively specific to provide sufficient information to derive the correct solution. In the meantime, they should also achieve certain degree of generality with information that go beyond the current input-output examples.

We first introduce an auxiliary function that will be used by the abstract synthesis algorithm.

Definition 2 (Relaxed Feasibility)**.** Given a partial program P as well as a set of IO examples E, we use the *CountConsist* function to count the number of examples in E that is consistent with program P:

$$\textsc{CountConsist}(P, E) = \sum_{\forall e \in E} \mathbb{1}(P \models e)$$

where $\mathbb{1}$ is the boolean predicate function[3].

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

Example 5. Consider the table shown in Figure. 4 (right) as input, and the following partial program P:

```
project(aggregate(I, null, ◇, ◇), ["Country"])
```

For the given set of model predictions as outputs:

"Brazil", "Japan", "China", "U.S."

Invoking CountConsist(P, E) will return 3. Because only "U.S." cannot be generated by any derivations of the partial program, which makes P consistent with three out of the four input-output examples.

Abstract Program Synthesis. Algorithm 1 shows the high-level structure of our synthesis algorithm, which takes as input a specification E that must be satisfied by the synthesized program, a domain-specific language with syntax $\mathcal{L}$, as well as a hyperparameter q that balances the generality and specificness, which we denote as a *balance coefficient*. The output of the AbsSynth procedure is either a set of partial/abstract programs $\mathcal{P}$ in the DSL or $\bot$, meaning that there is no DSL program that satisfies E.

Internally, our synthesis algorithm maintains a worklist data structures $\mathcal{W}$. The worklist $\mathcal{W}$ is a set of *abstract programs* that will eventually be returned by the procedure. In particular, the AbsSynth procedure initializes $\mathcal{W}$ with a single root node labeled with the start symbol S (line 2); thus, $\mathcal{W}$ initially contains an abstract program P that represents any syntactically legal DSL program.

In each iteration of the while loop (lines 3–16), we pick an abstract program P from $\mathcal{W}$ (line 5) and iteratively compute all possible refinements P' using the production rules defined by $\mathcal{L}$ (line 6). For each candidate refinement P^*, we invoke the CountConsist procedure to compute the number of examples N in E that is consistent with P^* (line 7). if n is greater than the threshold q, it means P^* is still too abstract thus requires further refinement. In this case, we add P^* to the worklist $\mathcal{W}$ (line 9) so that the program gets refined again in the near future. In the second case where n is no greater than q (line 10), it indicates that P^* is too specific and may lead to overfitting. In this case, we include P, which is the abstract program from which P^* is refined, to the worklist $\mathcal{W}$ (line 11). Finally, the algorithm terminates when the worklist $\mathcal{W}$ reaches a fixed-point (line 14). In other words, for all programs $P \in \mathcal{W}$, any refinement on P will lead to programs that are too specific (i.e., $\textsc{CountConsist}(P', E) \leq q$).

Algorithm 1 Abstract Synthesis with Noisy Specification

```
Input: DSL $\mathcal{L}$, IO Examples $E$, Balance Coefficient $q$
Output: Set of Partial Programs $\mathcal{P}$ or $\bot$
procedure AbsSynth($\mathcal{L}, E, q$)
    $\mathcal{W} \leftarrow \{\textsc{Root}(S)\}$
    while true do
        $\mathcal{W}' \leftarrow \mathcal{W}$
        $\mathcal{W} \leftarrow \mathcal{W} - \{P\}$
        for $P^* \in \{P' \mid \forall d \in \mathcal{L}, P \xrightarrow{d} P'\}$ do
            $n \leftarrow \textsc{CountConsist}(P^*, E)$
            if $n > q$ then
                $\mathcal{W} \leftarrow \mathcal{W} \cup \{P^*\}$
            else if $n \leq q$ then
                $\mathcal{W} \leftarrow \mathcal{W} \cup \{P\}$
            end if
        end for
        if $\mathcal{W} = \mathcal{W}'$ then return $\mathcal{W}$
        end if
    end while
end procedure
```

Example 6. Following Example. 5, for the same given IO examples, Algorithm. 1 iteratively finds out a set of feasible partial programs given the threshold $q = 3$. We go over the algorithm with a few concrete iterations:

1. The algorithm starts from a start symbol of hole $\mathcal{W} = \{\diamond\}$ and $P = \diamond$, which is feasible for all examples.
2. The algorithm derives P with well-typed production rules (line 6). For example, one of the P^* could be:
```
project(◇)
```
which is also feasible for all the IO examples (line 7), i.e., $n = 4$. In this case, since $n > q$, the above program is added to the worklist (line 8-9).
3. The derivation continues until P becomes:
```
project(aggregate(I, null, ◇0, ◇1), ["Country"])
```
From the previous example we know currently P satisfies only three of the IO examples, i.e., $n = 3$, but not sure whether it can be further refined, so we add P to the worklist and continue with the iteration.
4. The algorithm attempts to fill $\diamond_0$ with `max`, which yields:
```
project(aggregate(I, null, max, ◇1), ["Country"])
```
and finds out it's only feasible for IO with outputs of "Brazil" and "Japan", i.e., $n = 2$. In this case since $n \leq q$, the previous P (before derivation) is added.
5. The procedure continues until the worklist $\mathcal{W}$ reaches a fixed point. (line 14).

5 Explanation Refinement via Optimal Program Synthesis

In this section, we describe our algorithm for synthesizing the optimal explanations that best match the consistency constraints implied between natural language questions and visualizations. We first define a *relational operator* to formalize the optimal synthesis problem:

Near-Synonym Linguistic Engine. First, we assume access to a *linguistic engine* that can specifically determine whether two *linguistic units* are *near-synonyms* [13], which constitutes to one of the major constraints of triangle alignment. A call to the near-synonym linguistic engine $\textsc{NSyn}(r, s) \in [0, 1]$ returns the degree of two linguistic units r and s sharing common senses, where 1 indicates *identical* and 0 indicates *irrelevant*. In other words, a near-synonym linguistic engine tells the "similarity"[4] between linguistic units, e.g., words, phrases, etc..

Example 7. Consider the following words: "high", "highest", "low", we have:

$$\textsc{NSyn}(\text{“high”}, \text{“highest”}) > \textsc{NSyn}(\text{“high”}, \text{“low”})$$

which means "high" is more similar to "highest" than "low".

ILP Formulation. A 0-1 Integer Linear Programming (ILP) consists of a set of linear constraints C over boolean variables and an objective function c. The goal is to find an assignment such that all constraints are satisfied and the value of the objective function c is optimized.

Definition 3 (0-1 Integer Linear Programming)**.** The 0-1 ILP problem is defined as follows:

$$min\ c : \sum_j c_j x_j$$
$$s.t.\ \ C : \bigwedge_i \sum_j a_{i,j} x_j \ \Delta\ b_i,$$

with $\Delta := \{\leq, =, \geq\}$, $x_j \in \{0, 1\}$, and coefficients $c_j, a_{i,j},$ and b_i are all integers.

We formulate the problem of finding an optimal triangle alignment using 0-1 ILP. Specifically, constraints C encode mappings $\mathcal{M}$ among entities from three parties: the question Q, the program P, and the visualization I. The objective function expresses that we want to minimize the cost of the mappings. In what follows, we describe our encoding in more detail.

Domains. The domains contains entities from three parties. In particular, each question Q contains a set of linguistic units $w \in V_w$, each visualization consists of a set of cells $t \in V_t$, and each P has a set of holes $h \in V_h$ that need to be filled. Finally, we also have a set of abstract programs

[4]Note that similarity techniqes based on *distributional hypothesis* [19], e.g., word2vec [25] and glove [28], are observably not suitable for distinguishing synonyms and antonyms.

$P \in V_P$ generated by Algorithm 1. Formally, the *triangle alignments* among entities from three parties are encoded as the conjunctions of the following boolean variables:

Variables. The variables in our 0-1 ILP formulation correspond to all possible mappings among three parties:

- x_w^t: the boolean variable indicates a one-to-one mapping from a linguistic unit w from the question Q to a cell value t from the data source (i.e., visualization).
- y_h^t: the boolean variable indicates a one-to-one mapping from a hole h of an abstract program to a terminal of cell value t. I.e., the hole h is filled with terminal t.
- z_P^h: the boolean variable indicates a mapping from an abstract program P to a hole h. In other words, z_P^h evaluates to 1 if hole h belongs to abstract program P.
- u^P: The boolean variable indicates the abstract program P (chosen from Algorithm 1.) is used to derive the final solution.

Example 8. The optimal mapping for Explanation#2 from Figure. 2 given the following program P:

```
project(select(pivot(T, ◇0, ◇1), ◇2, ◇3, ◇4), ◇5)
```

can be represented by the following variables:

- $x_{\text{country}}^{\text{Country}} = true$, $x_{\text{most worse}}^{\text{Worsen}} = true$
- $y_{\diamond_5}^{\text{Country}} = true$, $y_{\diamond_2}^{\text{Worsen}} = true$
- $\forall i \in \{0, 1, 2, 3, 4, 5\}, z_P^{\diamond_i} = true$
- $u^P = true$

Observe that the number of variables used in the encoding grows quadratically for the number of words in the question Q as well as the number of holes in the abstract program. However, since the number of words and holes is usually small, our encoding introduces a manageable number of variables in practice.

Constraints. While the variables describe all possible mappings among entities from different parties, not all mappings can occur simultaneously. For example, we must enforce that any satisfying assignment to C corresponds to a mapping from entities in visualization v to holes in P. Furthermore, types also impose *hard constraints* that limit which variables in V can be mapped to which ones in H. We enforce these hard constraints by generating a system of linear constraints C as follows:

1. **Well-typed terminals.** If two parameters $h \in H$ and $t \in T$ are not compatible due to their types, the boolean variables where these parameters occur are always set to 0.

   $y_h^t = 0$ if the types of t and h are incompatible.

2. For each hole $h \in H$, we impose that there is *exactly one* terminal t that maps to h:

$$\forall h \in V_h, \sum_{\forall t \in V_t} y_h^t = 1$$

   Effectively, these constraints enforce that any solution of C corresponds to a surjective mapping.

3. In a similar way, we also impose that there is *exactly one* abstract program P that will be chosen:

$$\sum_{\forall P \in V_P} u^P = 1$$

   Furthermore, each hole h can belong to *exactly one* abstract program p:

$$\forall h \in V_h, \sum_{\forall P \in V_P} z_P^h = 1$$

4. For each entity $t \in V_t$, we impose that there is *at most one* mapping in the question Q that contains t:

$$\forall t \in V_t, \sum_{w \in V_w} x_w^t \leq 1$$

5. Finally, we ensure that a mapping only gets activated if its corresponding abstract program P is chosen:

$$\forall h \in V_h, \forall t \in V_t, \; -y_t^h + u^P + z_P^h \geq 1$$

Example 9. Following Figure. 2 and Example. 8, we can construct the corresponding constraint system by defining the set of holes V_h and set of cell values V_t, which are given by:

$$V_h = \{\diamond_i | i = 0, 1, \ldots\}, V_t = \{Country, opinion, \ldots\}.$$

Objective Function. We borrow the notion of perplexity from information theory to measure how common a candidate abstract program is observed. Given a program P, assuming we have function $\text{PPL}(P)$ that computes the perplexity of P using an off-the-shelf statistical model, then the goal of the objective function c in our ILP formulation is to find an *optimal* alignment with the lowest cost and perplexity. Specifically, we define the objective function c as follows:

$$\sum_{w \in V_w} \sum_{t \in V_t} (1 - \textsc{NSyn}(w, t)) \cdot x_w^t + \sum_{p \in V_P} \text{PPL}(P) \cdot u^P$$

Each mapping has an associated cost using linguistic distances defined at the beginning, and the perplexity score will bias the objective function to prefer more promising candidates.

Example 10. Following Example. 9, suppose eventually we want to find out the optimal explanation from the following two programs (denoted by P_1 and P_2):

```
1 project(select(pivot(
    T, "opinion", "%"), "Worsen", eqmax, null), ["Country"])
2 project(select(pivot(
    T, "opinion", "%"), "Improve", eqmax, null), ["Country"])
```

Note that the linguistic engine has the following returned scores:

$$\text{NSyn}(\text{Country}, \text{country}) = 1,$$
$$\text{NSyn}(\text{Worsen}, \text{most worse}) = 0.6,$$

with other scores not mentioned omitted (since they are mostly shared between the two programs and won't affect the final result), and the computed perplexity of both programs are $\text{PPL}(P_1) = 3.93$ and $\text{PPL}(P_2) = 3.99$. Both costs can be computed by:

$$cost(P_1) = (1-1)\cdot 1 + (1-0.6)\cdot 1 + 3.93 = 4.33,$$
$$cost(P_2) = (1-1)\cdot 1 + (1-0)\cdot 1 + 3.94 = 4.94.$$

Obviously P_1 has lower cost and the optimal synthesis will propose it as the optimal candidate explanation.

6 Implementation

We have implemented the proposed framework in a tool called Poe, which consists of approximately 6,000 lines of Python code. Poe is built on top of the Trinity [24] framework. In particular, our component specifications are expressed (Section. 3) in quantifier-free Presburger arithmetic. More specifically, we use a similar DSL for the data wrangling domain and the same specifications considered in prior work [16]. The linguistic engine is built using NLTK (with WordNet interface) [7] and spaCy [27]. In what follows, we elaborate other key implementation details.

Deep Learning Model. Given a visualization query in English as well as a table that corresponds to the visualization, Poe incorporates the pre-trained weakly supervised model from the TaPas tool, to generate the top-k answers as the starting point of the system.

Rendering Visualization as Table. Similar to VisQA [22] and Viser [42], Poe also needs to convert a visualization into its table representation, with additional visual properties attached, such as colors, shapes, etc.. In particular, Poe invokes the Vega-Lite [34] visualization tool to render the visualization from the benchmark specification with extra accessible rich internet application (ARIA) attributes [40], and retrieves them by parsing together with additional visual properties as a compact table. This reduces the complex visualization to its succinct tabular format that is amendable to existing data wrangling DSL.

Other Optimizations. Our implementation performs extra optimizations in addition to the algorithms presented in Sections. 4 and 5. First, following the Occam's razor principle, Poe explores abstract programs in increasing order of size. In the meantime, if the size of the candidate answers is a large number k, Poe may end up exploring many abstract programs. In practice, we have found that a better strategy is to exploit the inherent parallelism of our algorithm. Specifically, Poe uses multiple threads to search for abstract programs for different answers.

Our deduction engine is inspired by prior works [15, 16], whose core procedures include: (1) every DSL construct is attached with its abstract semantics in form of first-order formulas describing the input-output behavior, (2) the semantics of a partial program is computed by conjoining the side effects of each individual construct, and (3) an SMT query is issued to encode the consistency between the abstract semantics and the user intent via implication.

Motivated by the Neo system [15], our implementation of CountConsist performs an additional optimization over Algorithm 1: Since different partial programs may share the same SMT specification, Algorithm 1 ends up querying the satisfiability of the same SMT formula multiple times. Thus, our implementation memoizes the result of each SMT call to avoid redundant Z3 queries.

Finally, since using a "universal DSL" for all visualization queries may significantly increase the search space of the synthesizer, motivated by the Lift framework [1], Poe will refine the DSL constructs on-the-fly and filter out irrelevant or redundant constructs with respect to the query and the visualization. In particular, Poe starts with a smaller DSL with constants that are relevant to the question, and to ensure completeness it gradually increases the DSL constructs on-the-fly if it fails to find any feasible candidate programs using the current DSL.

Perplexity Computation. Recall that in Section. 5, our optimal synthesis relies on computing the perplexity of each candidate abstract program. Because perplexity measurement of an abstract P requires a background probability model of P, we adapt a similar statistical model from the Morpheus system [16], which uses a 2-gram model trained on 15,000 code snippets collected from Stackoverflow. Since Poe's search strategy always starts with an abstract program P derived from abstract synthesis, $\text{PPL}(P)$ is weighted slightly higher than $\text{NSyn}(P)$ in order to balance the objective function.

7 Evaluation

In this section, we describe the results for the experimental evaluation, which is designed to answer the following key research questions:

1. **RQ1. Performance**: How does Poe compare against state-of-the-art tools on visualization queries?
2. **RQ2. Effectiveness**: Can Poe rectify wrong answers proposed by other tools?
3. **RQ3. Explainability**: Does Poe synthesize explanations that well capture the question intentions and make sense to human end-users?

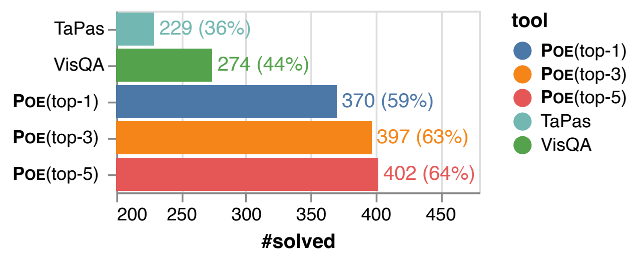


Figure 7. Performance comparison between the original pipeline from VisQA (baseline), TaPas and Poe.

4. **RQ4. Ablation**: How significant is the benefit of abstract synthesis (Section. 4) and optimal alignment (Section. 5)?

Benchmarks. We evaluate Poe on a total number of 629 visualization question-answering tasks used in VisQA [22]. Specifically, these tasks contain visualizations collected from different real-world data sources and non-trivial questions in natural language proposed by real users from Mechanical Turk. The types of questions cover including retrieval, aggregation, assertion, and comparison, etc.

Experimental Setup. To evaluate the effectiveness of Poe, we choose two state-of-the-arts, VisQA and TaPas [20]. In particular, TaPas leverages a weakly supervised model and provides an end-to-end way to directly predict the answer without explicitly generating logic forms, where Poe collects top-30 answers from TaPas as input to its abstract synthesis component. VisQA is an automatic pipeline for answering natural language questions about visualizations and it builds on top of Sempre [6], a question-answering system for relational data tables.

All experiments are performed on Amazon EC2 platform with a t3a.xlarge instance. The time limit for a single task is 5 mins. We set the balance coefficient $q = 3$ by default[5].

7.1 Comparison against State-of-the-Arts

To answer **RQ1**, we compare Poe against VisQA and TaPas on all the 629 VisQA benchmarks discussed earlier. We measure the total number of benchmarks solved, which is shown in Figure. 7. As we can see, within given time limit, Poe solves 370 (59%) benchmarks, whereas VisQA solves 274 (44%) and TaPas solves 229 (36%). By comparison, Poe solves 11% more benchmarks than VisQA and 23% than TaPas.

Additionally, we show more details of the comparison with respect to different questions types, as shown in Table. 1. Poe solves on average 35% (resp. 25%) more benchmarks across different types of questions compared to VisQA (resp. TaPas), and has a lower variance on performance of different types of questions, whereas TaPas only supports and is good at a restricted portion of questions. Thus, we believe these results answer **RQ1** in a positive way.

[5]Note that in practice q may need to be adjusted dynamically depending on the quality of candidate programs derived from abstract synthesis. For example, for some benchmarks the statistical model may not produce enough candidate answers and q needs to shrink accordingly so as to prevent generating programs that are too abstract.

Table 1. Comparison on number of benchmarks solved by different tools across different types of questions.

question type	total	VisQA (baseline)	TaPas	Poe (top-1)
retrieval	183 (29%)	101 (55%)	98 (54%)	**123 (67%)**
comparison	87 (14%)	50 (57%)	0 (0%)	**71 (82%)**
aggregation	253 (40%)	92 (36%)	119 (47%)	**161 (64%)**
other	106 (17%)	**31 (29%)**	12 (11%)	15 (14%)
total	629 (100%)	274 (44%)	229 (36%)	**370 (59%)**

7.2 Benefits of Optimal Alignment and Abstract Synthesis

To study the effectiveness of abstract synthesis and optimal alignment, we further perform an ablation study in which we compare Poe against two of its other variants:

- Poe^{O}: This variant *only* performs optimal synthesis on the full search space.
- Poe$^{\mathcal{A}}$: This variant *only* performs abstract synthesis-followed by an enumerative search to pick the first feasible concrete program.

The results from this evaluation are summarized in Table. 2 with given timeout of 5 mins. As we can see, without abstract synthesis procedure, Poe^{O} is still able to solve a certain number of benchmarks (357) since the consistency constraints provide very strong hints that greatly reduce the search space. While for Poe$^{\mathcal{A}}$ without optimal synthesis, majority of the synthesis calls are timed out. Without prioritization provided by optimal synthesis, Poe$^{\mathcal{A}}$ finds it difficult to reach the optimal solution quickly even after search space pruning. The full version of Poe combines both the benefits of the abstract synthesis and optimal synthesis and thus reaches the best results among the variants. Thus, we conclude that the ablation study provides positive evidence for **RQ4** and shows the necessity of both procedures.

7.3 Evaluation on Effectiveness

To answer **RQ2**, we specifically measures the *flip rate* of Poe over TaPas, i.e., the percentage of the benchmarks that Poe can *rectify* on top of TaPas. We compute flip rate of tool A

Table 2. Comparison between TaPas and different ablated variants of Poe.

variant	TaPas	Poe	Poe$^{\mathcal{A}}$	Poe$^{\mathcal{O}}$
solved	229	370	194	357
delta (%)	(+0%)	(+23%)	(-5%)	(+21%)
#timeout	-	36	586	58

over tool B according to the following equation:

$$FLIP(\frac{A}{B}) = \frac{|SUCC(A) \cap FAIL(B)|}{|FAIL(B)|}$$

where $SUCC$ returns the set of successfully solved benchmarks and $FAIL$ returns the set of failed benchmarks. Our results show that Poe has a flip rate of 39% over TaPas, which means it can successfully "fix" 39% of the benchmarks that TaPas fails to solve. In particular, for retrieval (resp. aggregation, comparison) type of questions, the flip rate is 36% (resp. 37%, 78%). In summary, we believe our proposed techniques in Poe are effective and thus **RQ2** is answered in a positive way.

7.4 A User Study on Explainability

To answer **RQ4**, we carry out a simple user study on a comparison of the usability and explainability between TaPas and Poe. The design of the user study is inspired by the one carried out by VisQA [22]. Specifically, 3 students with elementary background of data analytics are asked to use Poe and TaPas and perform the following evaluations given real-world visualizations (and their corresponding parsed tables):

- **Task 1 (Usability)**: Ask a question regarding the given visualization and evaluate which tool returns the accurate desired answers.
- **Task 2 (Explainability)**: Inspect the returned answer together with the explanation generated by Poe and tell whether the answer is well explained and aligns with the user intent.

In particular, 3~5 individual questions were asked in each task, and the participants were asked to make a choice between Poe and TaPas for each question based on the usability and explainability of the answers given by both tools.

As a result, the participants indicate in our results that Poe is demonstrating better usability than TaPas in that it solves more questions asked by users. Out of all the visualization question answering tasks they issued, Poe finds the correct explanations that well match their original intents of the questions in majority of the cases. Thus, for **RQ3**, we believe the user study provides positive evidence about the usability of Poe and explainability of the explanations generated.

7.5 Discussion

Like any other techniques, our approach also has its own limitations. Based on the result in Figure. 7, we manually inspect all these cases and notice that the issue is caused by the following reasons:

Timeout. Poe uses a timeout of 5 mins similar to previous works [15, 16]. As a result, 5% of difficult benchmarks do not terminate within the given timeout.

Incomprehensible Question. Since the natural questions from VisQA benchmarks are obtained from Amazon Mechanical Turk, some of the questions are found to be incomprehensible, e.g.:

- *"What is highestt change in income?"* – typo.
- *"In which year investors of all age groups took bigger risks?"* – "bigger" should be "biggest".
- *"Who has roughly 5 votes?"* – factual error; no one has 5 votes in the visualization.

Such benchmarks create difficulties for all of the tools we experiment on.

Fallback Strategy. Poe starts its core synthesis algorithm based on the top-k answers from TaPas. In some cases, the top-k answers may all be wrong and do not provide any hints to derive the correct solution. Then Poe has to leverage a simple fallback strategy to dynamically increase the size of k, which may lead to timeout.

Limitation of NLP Modules. Some of the benchmarks are found to be also challenging for the current NLP techniques that the tools depend on. For example:

- *"How many countries in Asia will have their economy improved based on majority votes?"* – requires a knowledge base backend for inferring the implication of *"countries in Asia"*.
- *"How many teams are in the Central Division?"* – requires alignment with entities from the visualization to the range of *"Central Division"*.
- *"What month has the least recorded weather?"* – requires aligning a implicit summary of more than one weather types to represent the weather before aggregation.

Despite the aforementioned limitations, our core technique is not restricted to visualization tasks. We anticipate that a similar idea can be instantiated to other tasks with multi-layer specifications (e.g., text, table, code, visual objects, etc.) that combine a top-down search procedure with an off-the-shelf statistical model. For instance, video understanding, structural-object queries, data wrangling, etc.

There are various directions that Poe can be extended to handle a broader spectrum of visualization queries. For example: 1) An extended version of the DSL that considers more data wrangling operations that cover more long-tailed queries in our benchmark; 2) A more sophisticated linguistic engine, 3) A better deep learning model trained from a better dataset.

8 Related Work

Machine Learning for Synthesis. There has been significant interest in automatically synthesizing programs given high-level specifications of different granularity [3, 8, 17, 18, 21, 30, 31, 38], such as program sketches [14, 38, 39], types [26, 30], logical forms [8, 39] and natural languages [9, 10, 35, 44, 46]. Recently, machine learning is extensively used for better prioritization of programs for search-based approaches [3, 5, 9, 10, 15]. On the other end, program synthesis techniques and formal methods are also used to provide rich and generalizable feedback for machine learning models [2, 4, 11, 36, 49]. For example, SQLizer [44] performs program repairs based on type-directed program synthesis; Concord [11] formalizes a generic deduction engine and performs deduction-guided off-policy sampling to enhance the reinforcement learning for program synthesis; Probe [4] utilizes guided bottom-up search to boostrap machine learning model for synthesis; Metal [37] uses graph-based models of reinforcement learning for synthesis with rewards from SMT solvers. Different from prior work [22, 44] that rely on a semantic parser whose training data is difficult and expensive to obtain, our approach focuses more on interpreting and rectifying the direct answers from weakly supervised machine learning models by synthesizing programs as explanations. Such models become increasingly popular due to the ease of obtaining training data.

Model Interpretability. Despite their popularity, machine learning models are often applied as black boxes. How to interpret the predicted results remains an important, yet challenging task. Ribeiro et al. [32] proposed a method to explain models by presenting representative individual predictions and their explanations. In deep learning, there are efforts to perturb the input to a neural network and visualize its influence to the output. Clark et al. [12] analyzed the attention mechanisms of pre-trained models and demonstrated syntactic information captured in these models. Petroni et al. [29] considered language models as knowledge bases. However, none of them can achieve rigorous explanation like what a synthesized program (e.g. generated by Poe) does.

Visualization / Table Question Answering. Semantic parsing of natural language queries to SQL has attracted increasing interest since the release of datasets like WikiSQL [48] and Spider [47]. Their leaderboards have been frequently updated by newly developed encoder and decoder architectures. For example, RAT-SQL [41] included schema encoding, schema linking, and feature representation in a unified relation-aware self-attention framework. Both autoregressive AST-based top-down (e.g., Yin and Neubig [45]) and bottom-up parsers (e.g., SMBOP [33]) have been proposed. Most of the those studies assume the existence of datasets that map natural language queries to logic forms or intermediate representation, which could be used to train encoders and decoders. Recently, weakly supervised approaches like TaPas [20] that do not rely on annotating logic forms, can be trained on larger corpora, thus outperform state-of-the-arts. Our evaluation shows that our introspective synthesis approach that reconciles the power of symbolic reasoning and machine learning can significantly push the boundary of visualization queries.

9 Conclusion

We have proposed a new methodology for synthesizing programs from natural language and applied it to the problem of answering visualization queries. Starting with a few tentative answers obtained from an off-the-shelf statistical model, our approach first invokes an *abstract synthesizer* that generates a set of sketches that are consistent with the answers. Then we design an instance of optimal synthesis to complete one of the candidate sketches by satisfying common type constraints and maximizing the consistency among three parties, i.e., natural language, the visualization, and the candidate program.

We implement the proposed idea in a system called Poe that can answer visualization queries from natural language. Our method is fully automated and does not require users to know the underlying schema of the visualizations. We evaluate Poe on 629 visualization queries and our experiment shows that Poe outperforms state-of-the-arts by improving the accuracy from 44% to 59%.

Acknowledgments

We thank our shepherd Dana Drachsler Cohen and the anonymous reviewers for their helpful feedback. This work has been supported in part by NSF Award #1908494, DARPA HARDEN and a Google Faculty Research Award.

References

[1] Maaz Bin Safeer Ahmad and Alvin Cheung. 2016. Leveraging Parallel Data Processing Frameworks with Verified Lifting. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016 (EPTCS, Vol. 229)*, Ruzica Piskac and Rayna Dimitrova (Eds.). 67–83. https://doi.org/10.4204/EPTCS.229.7

[2] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and Abstraction: A Synergistic Approach for Analyzing Neural Network Robustness. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 731–744. https://doi.org/10.1145/3314221.3314614

[3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs. In *Proc. International Conference on Learning Representations*. OpenReview. https://doi.org/10.48550/arXiv.1611.01989

[4] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-Time Learning for Bottom-up Enumerative Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA (nov 2020). https://doi.org/10.1145/3428295

[5] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *PACMPL* 3, OOPSLA (2019), 168:1–168:27. https://doi.org/10.1145/3360594

[6] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Seattle, Washington, USA, 1533–1544. https://www.aclweb.org/anthology/D13-1160

[7] Steven Bird and Edward Loper. 2004. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*. Association for Computational Linguistics, Barcelona, Spain, 214–217. https://aclanthology.org/P04-3031

[8] James Bornholt and Emina Torlak. 2017. 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*. 467–481. https://doi.org/10.1145/3062341.3062353

[9] 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 (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

[10] Yanju Chen, Ruben Martins, and Yu Feng. 2019. 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*. 602–612. https://doi.org/10.1145/3338906.3338951

[11] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *Computer Aided Verification*, Shuvendu K Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 587–610. https://doi.org/10.1007/978-3-030-53291-8_30

[12] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. 2019. What Does BERT Look At? An Analysis of BERT's Attention. arXiv:1906.04341 [cs.CL]

[13] Philip Edmonds and Graeme Hirst. 2002. Near-Synonymy and Lexical Choice. *Computational Linguistics* 28, 2 (2002), 105–144. https://doi.org/10.1162/089120102760173625

[14] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to infer graphics programs from hand-drawn images. In *Advances in neural information processing systems*. 6059–6068. https://doi.org/10.48550/arXiv.1707.09627

[15] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proc. Conference on Programming Language Design and Implementation*. 420–435. https://doi.org/10.1145/3192366.3192382

[16] 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 *Proc. Conference on Programming Language Design and Implementation*. ACM, 422–436. https://doi.org/10.1145/3062341.3062351

[17] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 229–239. https://doi.org/10.1145/2737924.2737977

[18] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. Symposium on Principles of Programming Languages*. ACM, 317–330. https://doi.org/10.1145/1926385.1926423

[19] Zellig S Harris. 1954. Distributional Structure. *WORD* 10, 2-3 (1954), 146–162. https://doi.org/10.1080/00437956.1954.11659520

[20] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 4320–4333. https://doi.org/10.18653/v1/2020.acl-main.398

[21] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proc. International Conference on Software Engineering*. ACM/IEEE, 215–224. https://doi.org/10.1145/1806799.1806833

[22] Dae Hyun Kim, Enamul Hoque, and Maneesh Agrawala. 2020. Answering Questions about Charts and Generating Visual Explanations. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–13. https://doi.org/10.1145/3313831.3376467

[23] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 70–80. https://doi.org/10.1145/3093335.2993244

[24] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: an extensible synthesis framework for data science. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1914–1917. https://doi.org/10.14778/3352063.3352098

[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, C J C Burges, L Bottou, M Welling, Z Ghahramani, and K Q Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://doi.org/10.48550/arXiv.1310.4546

[26] Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing symmetric lenses. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–28. https://doi.org/10.1145/3341699

[27] Ines Montani, Matthew Honnibal, Matthew Honnibal, Sofie Van Landeghem, Adriane Boyd, Henning Peters, Paul O'Leary McCann, Maxim Samsonov, Jim Geovedi, Jim O'Regan, György Orosz, Duygu Altinok, Søren Lind Kristiansen, Roman, Explosion Bot, Leander Fiedler, Grégory Howard, Wannaphong Phatthiyaphaibun, Yohei Tamura, Sam Bozek, murat, Mark Amery, Björn Böing, Pradeep Kumar Tippa, Leif Uwe Vogelsang, Bram Vanroy, Ramanan Balakrishnan, Vadim Mazaev, and GregDubbin. 2021. *explosion/spaCy: v3.2.0: Registered scoring functions, Doc input, floret vectors and more.* https://doi.org/10.5281/zenodo.5648257

[28] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. https://doi.org/10.3115/v1/D14-1162

[29] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick S. H. Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander H. Miller. 2019. Language Models as Knowledge Bases?. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*. 2463–2473. https://doi.org/10.18653/v1/D19-1250

[30] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *Proc. Conference on Programming Language Design and Implementation* (2016), 522–538. https://doi.org/10.1145/2908080.2908093

[31] Oleksandr Polozov and Sumit Gulwani. 2015. 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. 107–126. https://doi.org/10.1145/2814270.2814310

[32] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1135–1144. https://doi.org/10.1145/2939672.2939778

[33] Ohad Rubin and Jonathan Berant. 2021. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies NAACL-HLT*. 311–324. https://doi.org/10.18653/v1/2021.spnlp-1.2

[34] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[35] Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. 2019. Program Synthesis and Semantic Parsing with Learned Code Idioms. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, Article 971, 11 pages. https://doi.org/10.48550/arXiv.1906.10816

[36] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 7762–7773.

[37] Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. 2019. Learning a Meta-Solver for Syntax-Guided Program Synthesis. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Syl8Sn0cK7

[38] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. 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*. 404–415. https://doi.org/10.1145/1168857.1168907

[39] Emina Torlak and Rastislav Bodík. 2014. 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*. 530–541. https://doi.org/10.1145/2594291.2594340

[40] W3C. 2017. Accessible Rich Internet Applications (WAI-ARIA) 1.1. https://www.w3.org/TR/wai-aria/. Accessed: 2021-11-14.

[41] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 7567–7578. https://doi.org/10.18653/v1/2020.acl-main.677

[42] Chenglong Wang, Yu Feng, Rastislav Bodík, Alvin Cheung, and Isil Dillig. 2020. Visualization by example. *Proc. ACM Program. Lang.* 4, POPL (2020), 49:1–49:28. https://doi.org/10.1145/3371117

[43] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program Synthesis using Abstraction Refinement. In *Proc. Symposium on Principles of Programming Languages*. ACM, 63:1–63:30. https://doi.org/10.1145/3158151

[44] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 63:1–63:26. https://doi.org/10.1145/3133887

[45] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. 440–450. https://doi.org/10.18653/v1/P17-1041

[46] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task. In *Proceedings of EMNLP*. Association for Computational Linguistics. https://doi.org/10.18653/v1/D18-1193

[47] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross- Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 3911–3921. https://doi.org/10.18653/v1/D18-1425

[48] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. In *arXiv:1709.00103[cs]*. https://doi.org/10.48550/arXiv.1709.00103

[49] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. 2019. An Inductive Synthesis Framework for Verifiable Reinforcement Learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 686–701. https://doi.org/10.1145/3314221.3314638